On the Solution Phase of Direct Solvers for Sparse Linear Systems with Multiple Sparse Right-Hand Sides

Gilles Moreau

Numéro National de Thèse : 2018LYSEN084

THÈSE de DOCTORAT DE L'UNIVERSITÉ DE LYON

opérée par

l'École Normale Supérieure de Lyon

*École Doctorale N°*512

École Doctorale en Informatique et Mathématiques de Lyon

Spécialité : Informatique

Soutenue publiquement le 10/12/2018, par

Gilles Moreau

On the Solution Phase of Direct Methods for Sparse Linear Systems with Multiple Sparse Right-Hand Sides

De la phase de résolution des méthodes directes pour systèmes linéaires creux avec multiples seconds membres creux

Devant le jury composé de :

Patrick AMESTOY	Professeur, INPT-ENSEEIHT-IRIT, Toulouse, France	*Co-encadrant*
Jocelyne ERHEL	Directrice de recherche, Inria Rennes, France	*Examinatrice*
John GILBERT	Professeur, University of California, Etats-Unis	*Rapporteur*
Laura GRIGORI	Directrice de recherche, Inria Paris, France	*Examinatrice*
Jean-Yves L'EXCELLENT	Chargé de recherche, Inria, Lyon, France	*Directeur de thèse*
Pierre RAMET	Maître de conférences, Université de Bordeaux, France	*Rapporteur*

Aknowledgements

J'aimerais d'abord remercier les membres du jury qui sont venus assister à ma soutenance, Jocelyne Erhel, Laura Grigori, et plus particulièrement John Gilbert et Pierre Ramet pour avoir lu attentivement mon manuscrit et pour leurs remarques constructives.

Ensuite, j'aimerais consacrer quelques mots à ceux qui m'ont aidé à la construction de ce manuscrit en y apportant une rigueur scientifique et des idées originales qu'il m'aurait été impossible de trouver seul. D'abord, merci à Daniil Shantsev et Jan Petter Morten de m'avoir accueilli en Norvège. Alfredo, Théo, François-Henry, Guillaume et Marie m'ont aussi apporté au cours de ces trois années une aide précieuse, ce fut un plaisir pour moi de collaborer avec eux. Je remercie aussi Patrick et tout particulièrement Jean-Yves de m'avoir transmis de nouvelles compétences qui j'en suis persuadé seront déterminantes.

Je remercie mes amis qui m'ont suivi au cours de ces trois ans, mes co-bureaus et aussi amis, Bertrand, Loïc et Issam, et mon amie, Ioanna, qui m'a accompagné au cours de cette dernière année. Pour finir, merci à mes parents de m'avoir soutenu et encouragé dans cette voie.

Abstract

We consider direct methods to solve sparse linear systems

$$AX = B,$$

where A is a sparse matrix of size $n \times n$ with a symmetric structure and X and B are respectively the solution and right-hand side matrices of size $n \times n_{rhs}$. A is usually factorized and decomposed in the form LDL^T, or LU, where L and U are respectively a lower and an upper triangular matrix, and D is diagonal. Then, the *solve* phase is applied through two triangular resolutions named the forward and the backward substitutions. Solving sparse linear systems arises in numerous fields of applications and for many years, the factorization has been the subject of much attention compared to the solve. It has rightly been considered as the most computationally intensive phase due to its higher complexity. Recent advances on the exploitation of *data sparsity* by low-rank approximations of matrices reduced this complexity but also increased the relative weight of the solve. In addition, for some applications, the very large number of right-hand sides (RHS) in B, $n_{rhs} \gg 1$, makes the solution phase the computational bottleneck. However, B is often sparse and its structure exhibits specific characteristics that may be efficiently exploited to reduce this cost.

The exploitation of sparsity in matrix B may be divided into two main features: *vertical* sparsity to reduce the number of operations by avoiding computations on rows that are entirely zero, and *horizontal* sparsity to go further by performing each elementary solve operation only on a subset of the RHS columns. We propose in this thesis to study the impact of the exploitation of this structural sparsity during the solve phase going through its theoretical aspects down to its actual implications on real-life applications. Although we mainly focus on the forward substitution, all the results obtained also apply to the backward substitution, when only part of the solution is required.

First, we investigate the asymptotic complexity, in the big-$\mathcal{O}$ sense, of the forward substitution when exploiting the RHS sparsity in order to assess its efficiency when increasing the problem size. In particular, we study on 2D and 3D regular problems the asymptotic complexity of the forward substitution both for traditional full-rank unstructured solvers and for the case when low-rank approximation is exploited. A significant asymptotic improvement is observed in the latter case. These complexity results are indeed more general, since they also provide a measure of the available parallelism of the solve phase in the dense RHS case.

Next, we extend state-of-the-art algorithms on the exploitation of RHS sparsity, and more particularly the horizontal feature, and also propose an original approach converging toward the optimal number of operations while preserving performance. For that, we propose a new

algorithm to build a permutation of the RHS columns, and then propose an original approach to split the RHS columns into a minimal number of blocks, while reducing the number of operations down to a given threshold. Both algorithms are motivated by geometric intuitions and designed using an algebraic approach, so that they can be applied to general systems.

Finally, we show the impact of the exploitation of sparsity in a real-life application. Controlled-Source ElectroMagnetism (CSEM) is a method of choice for oil and gas exploration that often requires the solution of sparse systems of linear equations with a large number of sparse right-hand sides. We explain why and how the algebraic properties described previously relate to the CSEM physical and geometrical context. We also adapt the parallel algorithms that were designed for the factorization to solve-oriented algorithms and describe performance optimizations particularly relevant for the very large numbers of right-hand sides of the CSEM application. The total CSEM simulation time can be divided by approximately a factor of 3 on all the matrices from our set (from 3 to 30 million unknowns, and from 4 to 12 thousands RHS).

We validate and combine the previous improvements using the parallel solver MUMPS, conclude on the contributions of this thesis and give some perspectives.

Contents

French summary

Introduction

Nous considérons la résolution de systèmes linéaires avec multiples seconds membres de la forme

$$AX = B, \tag{1}$$

où A est une matrice carrée, creuse, non singulière, de taille $n \times n$ avec une structure symétrique ou symétrisée, et où X et B sont respectivement la matrice de solutions et la matrice de seconds membres, toutes deux de taille $n \times n_{rhs}$. La résolution de systèmes linéaires creux se pose dans de nombreux domaines d'application et est un domaine de recherche actif. Parmi les applications industrielles nécessitant la résolution efficace de l'équation (1), on peut citer la mécanique des structures, la géophysique, l'électromagnétisme, l'analyse ou optimisation de données. Le besoin de précision numérique et la complexité croissante des simulations numériques entraînent des difficultés pour résoudre de grands systèmes linéaires avec des centaines de millions d'inconnues et font de la résolution des systèmes creux une véritable clé de voûte du calcul scientifique.

Du point de vue théorique et algorithmique, même si la question de la résolution de tels problèmes est bien comprise, il s'agit toujours d'un domaine de recherche très actif, comme cela sera illustré dans cette thèse. En outre, les progrès constants et impressionnants de la puissance informatique ainsi que l'évolution des architectures des supercalculateurs ont motivé un travail d'adaptation également constant. En effet, les supercalculateurs modernes dotés de milliers de nœuds de calcul constituent les nouveaux outils indispensables à la résolution de problèmes de plus en plus grands, mais au prix d'une difficulté accrue pour préserver la précision et atteindre une certaine efficacité dans l'utilisation de la mémoire et des processeurs.

Il existe deux classes principales de méthodes pour la résolution de systèmes linéaires creux de la forme de l'équation (1). Avec les *méthodes directes*, la matrice A est d'abord factorisée comme un produit de matrices dont la structure permet une résolution aisée (matrices diagonales, à permutation, triangulaires, orthogonales, par exemple). Dans notre contexte, nous considérerons la factorisation de A en un produit de deux matrices triangulaires L et U respectivement triangulaire inférieure et triangulaire supérieure. Même si, pour une question de stabilité numérique, cette factorisation LU peut également impliquer des matrices diagonales (pour la mise à l'échelle des lignes et des colonnes) et des matrices de permutation pour le pivotage numérique, nous supposerons, par souci de simplicité, que $A = LU$ après la phase de factorisation. La prochaine étape, appelée *phase de résolution* dans la suite de cette thèse,

effectue deux résolutions triangulaires, la *descente*, qui consiste à résoudre

$$LY = B, \tag{2}$$

pour obtenir la matrice Y de taille $n \times n_{rhs}$, suivie de la *remontée*

$$UX = Y. \tag{3}$$

D'une autre manière avec les *méthodes itératives*, une séquence de X_k qui converge, à une précision donnée, vers la solution X de notre système linéaire est construite. Les méthodes directes sont connues pour être numériquement plus stables mais plus exigeantes en termes de mémoire et de calculs, alors que les méthodes itératives qui pourraient être plus rapides, sont en général moins exigeantes en terme de mémoire mais numériquement moins robustes.

Dans cette thèse, nous nous concentrons sur les méthodes directes et plus particulièrement dans la résolution des équations (2) et (3) soit lorsque la matrice de seconds membres B est creuse, soit lorsque l'ensemble des solutions recherchées dans la matrice X est réduit. Ainsi, nous commençons par rappeler le contexte des méthodes directes en nous concentrant sur la phase de résolutions triangulaires. Deux applications de modélisation sismique et électromagnétique pour lesquelles la phase de résolutions triangulaires domine les temps de calcul seront utilisées pour illustrer nos discussions. Nous évaluons tout d'abord le comportement asymptotique de la phase de résolution dans le contexte de seconds membres creux et d'approximations de rang faible. Une amélioration des algorithmes existants permettant l'exploitation du creux dans les seconds membres est ensuite décrite et une approche originale permettant d'atteindre le nombre minimal d'opérations est introduite. Ensuite, nous présentons à nouveau des mécanismes permettant l'exploitation du creux mais cette fois-ci en terme de propriétés physiques liées à l'application. Des algorithmes pour améliorer les performances dans un environnement parallèle sur des applications réelles sont également proposés. Enfin, nous tentons de rassembler les contributions apportées des études précédentes puis concluons sur le travail effectué.

Contributions

Comme dit précédemment, nous nous intéressons plus particulièrement dans cette thèse à la phase de résolution soit lorsque les seconds membres (ou RHS pour Right-Hand Side) sont creux, soit lorsque seulement un sous-ensemble des entrées de la solution est requis. L'objectif de la première partie est de rappeler des travaux existants concernant l'exploitation de RHS creux et d'introduire quelques concepts nécessaires dans les études suivantes. En particulier, cette première partie fournit un formalisme qui sera utilisé dans l'étude traitant de l'amélioration de l'existant pour prouver de nouvelles propriétés. En plus des notions liées aux méthodes directes, notamment l'arbre d'élimination, nous caractérisons le creux *vertical* et *horizontal*, chacun associé à un outil permettant de l'exploiter. De plus, des notions telles que nœuds actifs/colonnes actives sont introduites. Nous illustrons également dans cette première partie l'importance de choisir une bonne permutation des colonnes de RHS.

Cette thèse s'articule ensuite autour de trois contributions principales (trois chapitres du manuscrit) brièvement résumées ci-après.

Calcul de complexité de la phase de résolution avec RHS creux multiple [JS1]. Nous soulignons que, dans le contexte d'un très grand nombre de RHS, la phase de résolution peut devenir le goulot d'étranglement de la simulation numérique complète. Dans ce chapitre nous étudions la complexité asymptotique de la phase de résolution en tenant aussi compte du fait que les matrices de facteurs peuvent être représentés avec des approximations de rang faible. Nous prouvons sur des problèmes réguliers 2D et 3D que l'exploitation des RHS creux dans le contexte des approximations de rang faible modifie significativement la complexité asymptotique des résolutions triangulaires.

Tout d'abord, l'utilisation de techniques d'approximations de rang faible ramène la complexité asymptotique de la phase de résolution, notée $\mathcal{C}(n)$, à une complexité linéaire $\mathcal{O}(n)$ (à un coefficient logarithmique près). Deuxièmement, le creux dans la matrice de seconds membres B peut être exploité pour réduire les coûts de la phase de descente et ainsi sa complexité asymptotique, notée $C^{ES}(n)$. En particulier, nous étudions sur des problèmes réguliers 2D et 3D les complexités asymptotiques à la fois pour les solveurs traditionnels non structurés de rang plein et pour le cas où des approximations de rang faible sont exploitées. Nous étudions particulièrement le rapport suivant :

$$\mathcal{G}^{ES}(n) = \frac{\mathcal{C}(n)}{\mathcal{C}^{ES}(n)}.$$

Une importante amélioration asymptotique est observée dans le cas de l'utilisation d'approximations de rang faible, pouvant aller jusqu'à $\Theta(n^{1/2})$[1]. Nous confirmons ces résultats théoriques d'abord sur des problèmes réguliers et ensuite sur un ensemble de matrices provenant d'applications réelles. Nous mentionnons que le résultat pourrait être étendu à l'ensemble de la phase de résolution lorsqu'une partie seulement de la solution est demandée.

De plus, cette étude de complexité fournit une mesure du parallélisme d'arbre disponible lors de la phase de résolution dans le cas de seconds membres denses. Une comparaison avec la complexité et le parallélisme de la factorisation montre des propriétés intéressantes de la phase de résolution à prendre en compte lors de la conception des algorithmes.

Amélioration des algorithmes existants sur l'exploitation du creux dans les RHS [W2, J1]. Dans ce chapitre, nous nous concentrons sur l'extension des algorithmes actuels pour l'exploitation des seconds membres creux. En nous basant sur une intuition géométrique en lien avec l'algorithme des dissections emboîtées, nous proposons tout d'abord une approche générique et plus efficace permettant de permuter les seconds membres et de réduire le coût de la phase de descente. Une deuxième contribution est la description d'un algorithme de blocage qui diminue encore ce coût en choisissant intelligemment des groupes de seconds membres pouvant être traités ensemble. Bien que les deux algorithmes soient motivés par des observations géométriques, ils sont conçus avec une approche algébrique, donnant une portée générale à ce travail. Les notions d'optimalité de nœud et d'indépendance de seconds membres sont introduites et formalisées, et des preuves théoriques justifiant l'efficacité des algorithmes proposés sont fournies.

[1] Par définition, $f(n) = \Theta(g(n))$ ssi $\exists C_1, C_2, n_0 > 0,\ \forall n > n_0, C_1 g(n) \leq f(n) \leq C_2 g(n)$. La notation devient nécessaire par la division dans $\mathcal{G}^{ES}$.

Les expériences confirment l'efficacité de la démarche proposée, notre permutation nommée "flat tree" réduit en moyenne de 13% de nombre d'opérations par rapport aux travaux précédents. L'algorithme de blocage réduit encore le nombre d'opérations. Basé sur une approche glouton, ce dernier tente de limiter le nombre de groupes nécessaires pour atteindre un nombre d'opérations qui peut être arbitrairement proche de la solution optimale. Dans un environnement séquentiel et sur nos applications réelles, nous comparons les performances de l'approche proposée avec des stratégies de blocage régulières et nous montrons la supériorité de notre approche de blocage.

Étude d'une application réelle dans un contexte de résolution parallèle [JS2, W1]. L'électromagnétisme à source contrôlée ("Controlled Source ElectroMagnetism" ou CSEM) est une méthode de plus en plus utilisée pour l'exploration du gaz et du pétrole. Dans ce contexte, l'inversion des données électromagnétiques (EM) pour des applications géophysiques à grande échelle nécessite souvent la résolution de systèmes d'équations linéaires avec un *grand* nombre de seconds membres *creux*, chacun correspondant à la position d'une source/d'un émetteur de l'application. Les solveurs creux directs sont très attrayants pour ce type de problèmes, surtout lorsqu'ils sont combinés avec des approximations de rang faible qui réduisent la complexité et le coût de la factorisation.

Nous montrons que l'exploitation des seconds membres creux et le calcul d'un sous-ensemble de la solution peuvent avoir un impact important sur les performances de la phase de résolutions triangulaires. Nous expliquons pourquoi et comment les propriétés algébriques et les outils introduits précédemment peuvent être utilisés pour accélérer le calcul dans le contexte CSEM.

Le premier objectif de ce chapitre est de proposer un autre point de vue sur les seconds membres creux qui ne nécessite pas d'être spécialiste des méthodes directes et/ou de la théorie des graphes, et de comprendre le potentiel applicatif de l'exploitation de ces seconds membre creux. Le deuxième objectif est d'adapter les algorithmes parallèles conçus pour la factorisation à la phase de résolution pour améliorer la performance de l'application, ce qui apparait particulièrement pertinent dans le cas d'un très grand nombre de seconds membres comme dans l'application CSEM. Alors que les précédents travaux ciblaient le nombre d'opérations, nous traitons plus particulièrement ici des problèmes de performances et de mémoire dans un environnement parallèle. Ce contexte motive l'amélioration des algorithmes pour mieux préserver le parallélisme de la phase de résolution. Nous montrons que le nombre d'opérations et le temps écoulé pour les résolutions triangulaires peuvent être considérablement réduits. La durée totale de la simulation CSEM peut être divisée par environ un facteur 3 sur tous les systèmes linéaires de notre ensemble (de 3 à 30 millions d'inconnues et de 4 000 à 12 000 RHS).

Conclusion et perspectives

Dans cette dernière section, nous résumons les contributions apportées puis dessinons quelques perspectives ou extensions du travail présenté dans ce manuscrit.

Ce travail se divise en trois contributions principales qui tentent d'évaluer théoriquement et d'améliorer techniquement les travaux existants exploitant les seconds membres creux. Le

premier aspect du travail est lié à l'établissement d'un formalisme théorique pour l'exploitation des seconds membres creux. Dans ce contexte, nous montrons théoriquement une réduction de la complexité asymptotique de la phase de descente (avec une extension à la phase de remontée lorsque la solution aussi est creuse), réduction d'autant plus importante que des approximations de rang faible sont utilisées. Ce résultat permet dans un premier temps de justifier l'importance de l'exploitation du creux dans les seconds membres, et dans un second temps de montrer des caractéristiques intéressantes de parallélisme d'arbre dans les résolutions triangulaires. La discussion se poursuit ensuite sur l'amélioration des algorithmes d'exploitation du creux existants. Nous concluons à la fin de cette seconde étude, pour les applications fournies, que la combinaison des deux algorithmes permet la réduction du nombre d'opérations jusqu'à une solution presque optimale tout en conservant de bonnes propriétés nécessaires à une résolution haute performance. Enfin, nous poursuivons par une étude naissant de la collaboration avec une entreprise travaillant sur une méthode numérique pour la recherche d'hydrocarbure. Nous y décrivons sous un autre point de vue l'exploitation du creux et proposons de nouveaux algorithmes adaptés à la phase de résolution.

En conclusion, nous avons essayé dans ce manuscrit de fournir une étude exhaustive de la phase de résolution avec seconds membres ou solution creuse. Théoriquement, nous définissons des notions utiles à la démonstration de propriétés mathématiques, nous continuons avec l'amélioration des algorithmes existants et enfin nous en étudions les effets sur une application réelle. En fin de manuscrit, nous rassemblons les résultats des différentes études, en combinant approximations de rang faible, nouveaux algorithmes d'exploitation du creux et algorithmes adaptés à la phase de résolution.

En perspective, tandis que cette thèse porte principalement sur la phase de résolution avec seconds membres multiples, dans de nombreuses applications, la phase de résolution s'effectue sur un seul second membre mais *de nombreuses fois*. C'est le cas par exemple des applications en régime instables où chaque second membre dépend de la solution du pas de temps précédent. Dans ce cas, le temps de résolution peut également devenir critique ou prédominant. Les algorithmes adaptés à la phase de résolution peuvent être appliqués, cependant, de nombreux autres travaux peuvent être menés pour adapter les algorithmes actuels qui sont orientés factorisation vers des algorithmes basés sur la phase de résolution. Par exemple, certains de nos résultats théoriques ont montré que la résolution présente plus de parallélisme d'arbre que la factorisation; cette propriété peut être utilisée pour piloter la conception de nouveaux algorithmes afin de mieux exploiter ce potentiel. Enfin, l'utilisation d'approximations de rang faible a modifié radicalement le comportement des algorithmes actuels dans des environnements parallèles (mémoire partagée et / ou distribuée). Des efforts algorithmiques ont été déployés pour exploiter efficacement les structures de rang faible au cours de la phase de factorisation et il reste beaucoup à faire pour améliorer les performances de la phase de résolution dans ce contexte.

Chapter 1

General introduction

We consider the solution of linear systems with multiple right-hand sides of the form

$$AX = B \tag{1.1}$$

where A is a large square nonsingular sparse matrix of size $n \times n$ with a symmetric or symmetrized structure, X and B are respectively the solution matrix and the matrix of right-hand sides of size $n \times n_{rhs}$. Solving sparse linear systems arises in numerous fields of applications and is an active field of research. Among the industrial applications that require the efficient resolution of Equation (1.1), we can cite structural mechanics, geophysics, electromagnetism, data analysis or optimization. The need for numerical precision and the increasing complexity of numerical simulations result in difficulties to solve large linear systems and makes the resolution of sparse systems a keystone in scientific computation.

From the theoretical and algorithmic point of view, even if the question of solving such problems is well understood, it is still quite an active research area, as will be illustrated in this thesis. Furthermore, the steady and impressive progress of computer power as well as the evolution of computer architectures have also motivated both algorithmic and more finalized work. Indeed modern supercomputers with thousands of computing nodes make the solution of increasingly large problems possible, but at the cost of an increased difficulty to preserve accuracy and to reach efficiency in memory and processor use.

There exists two main classes of methods for the resolution of large sparse linear systems of the form of Equation (1.1). With *direct methods*, matrix A is first factored as the product of easy to solve matrices (e.g. diagonal, permutation, triangular, orthogonal matrices). In our context we will consider the factorization of A into the product of two triangular matrices L and U respectively lower and upper triangular. Even if for numerical stability this so-called LU factorization might also involve diagonal matrices (for scaling the rows and the columns) and permutation matrices for numerical pivoting, we will assume, for the sake of simplicity, that $A = LU$ after the factorization phase. The next step, referred to as the *solve phase* in the remainder of this thesis, performs two triangular resolutions, the so-called *forward substitution*

$$LY = B, \tag{1.2}$$

obtaining the $n \times n_{rhs}$ matrix Y, followed by the so-called *backward substitution*

$$UX = Y. \tag{1.3}$$

With *iterative methods*, a sequence of X_k that converges, at a given precision, toward the solution X of our linear system is built. Direct methods are known to be numerically more stable but more demanding in terms of memory and computations, whereas iterative methods might be faster, are in general less memory demanding but are numerically less robust.

In this thesis, we focus on direct methods which share with graph theory many algorithmic aspects. This strong relation between graph and direct methods will be highlighted in this chapter, with a special focus on the forward and backward substitutions. The chapter is organized as follows. We first consider dense matrices in Section 1.1.1 and describe simplified algorithms for the factorization and solve phases. In Section 1.2, we approach the main steps and main notions involved in the processing of sparse matrices with direct methods. In Section 1.3, we study in more details the resolution of Equations (1.2) and (1.3) and consider the case where matrix B is sparse. In Section 1.4, we provide operation counts of the solve phase on regular problems and illustrate some properties that are different from the ones of the factorization. Low-rank formats are briefly introduced in Section 1.5. Finally, we give some remarks on the test cases and the computational environment in Section 1.6.

1.1 Solution of sparse linear systems

1.1.1 Dense matrices: LU factorization and solve phases

At each step of the LU factorization of a dense matrix, Gaussian elimination is applied on a reduced matrix to build part of the L and U matrices. The LU factorization algorithm does not impact the solution, giving an easy way to compute it. In the general case, depending on the properties of matrix A, Gaussian elimination can be used to factor the matrix A in the general form LU if A is unsymmetric, LDL^T with D diagonal (possibly with 2x2 diagonal blocks) if A is general symmetric and LL^T if A is symmetric definite positive.

Algorithm 1.1 Dense LU (Right-looking) factorization.

1: **Input:** a matrix A of order n
2: **Output:** A is replaced by its LU factors
3: **for** $k = 1$ **to** $n - 1$ **do**
4: **Factor:** $a_{k+1:n,k} \leftarrow a_{k+1:n,k}/a_{k,k}$
5: **Update:** $a_{k+1:n,k+1:n} \leftarrow a_{k+1:n,k+1:n} - a_{k+1:n,k}a_{k,k+1:n}$
6: **end for**

Algorithm 1.1 is a simplified sketch of the dense LU factorization based on Gaussian elimination in which the lower part of A is replaced by the L factor with implicit ones on the diagonal and the upper part is replaced by the U factor.

At each step k of the Gaussian elimination, the diagonal entry a_{kk} is used as a *pivot* to eliminate entries in column k of A and build column k of L. For the sake of simplicity, we

have assumed here that all pivots are large enough to preserve the numerical stability of the factorization so that columns of A can be processed in order without performing so-called numerical pivoting. At each step k of our proposed algorithm, firstly the column k of the L factor is computed, denoted as **Factor** in Algorithm 1.1. Secondly the trailing submatrix is modified through a rank-one update, denoted as **Update** in Algorithm 1.1. Please note that the order in which the entries of the L and U matrices are computed is not unique.

Following the steps in Algorithm 1.1, the operation count of the factorization is

$$\sum_{k=1}^{n-1} \sum_{i=k+1}^{n} (1 + \sum_{j=k+1}^{n} 1),$$

and is proportional to $\mathcal{O}(n^3)$ while the memory consumption is of the order $\mathcal{O}(n^2)$ [38]. Given the two factor matrices L and U, the solution of the linear system at Equation (1.1) requires two triangular resolutions: first the forward substitution which solves a lower triangular system using L, second the backward substitution which solves an upper triangular system using U. In Algorithm 1.2, the L factors are accessed by columns and the U factors are accessed by rows. This leads to a right-looking algorithm for the forward substitution and a left-looking algorithm for the backward substitution.

Algorithm 1.2 Dense triangular solution through forward and backward substitutions.

Solution of $Ly = b$
(forward substitution)

$y \leftarrow b$
for $j = 1$ **to** n **do**
  for $i = j + 1$ **to** n **do**
    $y_i \leftarrow y_i - l_{ij} \cdot y_j$
  end for
end for

Solution of $Ux = y$
(backward substitution)

$x \leftarrow y$
for $i = n$ **to** 1 **by** -1 **do**
  for $j = i + 1$ **to** n **do**
    $x_i \leftarrow x_i - u_{ij} \cdot x_j$
  end for
  $x_i \leftarrow x_i / u_{ii}$
end for

In Wilkinson's definition, a matrix is considered sparse when it is worth taking advantage of its nonzero structure. In physical applications, sparsity is often due to the fact that distant points in the physical domain do not interact with each other. Strictly speaking, the interaction between two nodes in the domain is represented by a nonzero entry (or a nonzero block) in the matrix A. In modern numerical computing, A can be extremely large (a few hundred millions of equations). It is thus critical to adapt dense algorithms to sparse objects, with the underlying objective to avoid storing the zero entries and thus spare useless operations and storage.

In the following sections we describe the main steps for the direct solution of sparse linear systems. We limit our description to the so-called three phase methods for which the resolution can be divided into three consecutive steps: the analysis, the factorization and the solve phases. The first two steps are briefly described in Section 1.2. The solution phase is described in more details in Section 1.3.

1.2 Sparse direct methods: analysis and factorization phases

1.2.1 The analysis phase

For the sake of simplicity, we assume that A has a symmetric structure and is factored in the form $A = LU$, as in Algorithm 1.1. The extension of the analysis phase to matrices with an unsymmetric structure is not straightforward and will be discussed at the end of this section.

In sparse direct methods, one key issue is the *fill-in* phenomenon. From the **Update** operation of Algorithm 1.1, fill-in appears when an entry a_{ij} in matrix A is initially zero and becomes nonzero. Indeed, an elementary **Update** operation can be written as

$$i, j > k, i \quad a_{ij} \leftarrow a_{ij} - a_{ik} \cdot a_{kj}$$

If $a_{ij} = 0$ but $a_{ik} \neq 0$ and $a_{kj} \neq 0$, then $a_{ij} \neq 0$ after the factorization phase if there is no numerical cancellation.

One should note that the order in which pivots are selected can influence the amount of fill-in during factorization. Furthermore, for a given sequence of pivots, one may also want to predict the fill-in that would occur during factorization. This will be referred to as the *the symbolic factorization* phase. The symbolic factorization is an important step that provides estimates of the cost in terms of number of operations and memory footprint of the factorization phase. The analysis phase most important steps are thus the preprocessing of the matrix to reduce the fill-in during factorization and the symbolic factorization to predict the main structures involved during the numerical phases: factorization and solve. Remark that the analysis phase may be performed only once when sequences of matrices with different numerical values but identical structure need to be factored. We will illustrate how a modelization based on graphs can be efficiently used to design the algorithms involved in the analysis phase and to model the factorization phase.

Adjacency graph

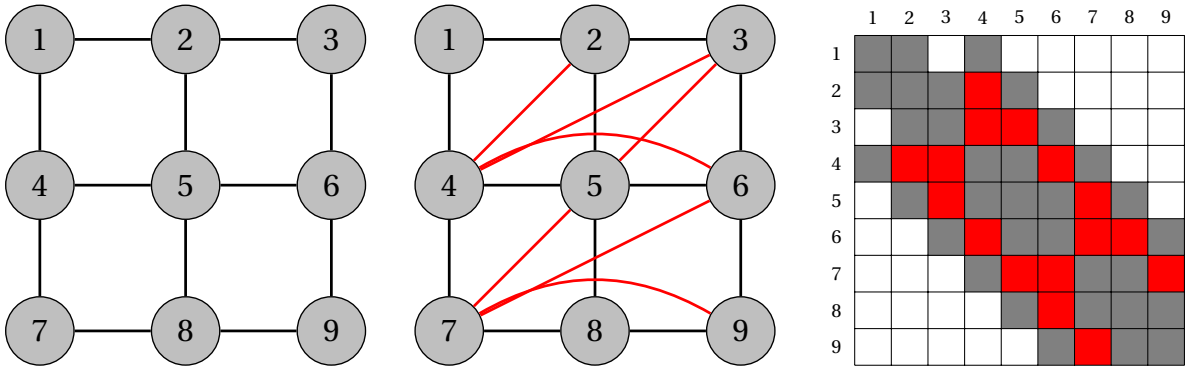
Graph formalism is introduced to analyze the properties of a sparse matrix A. The sparsity pattern of any sparse matrix A can be modeled by a so-called adjacency graph $\mathcal{G}(A)$.

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

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

If A is structurally symmetric (*i.e.* $a_{ij} \neq 0$ iff $a_{ji} \neq 0$), then one can also consider an undirected graph representation. When not explicitly stated, we will assume that the matrix is structurally symmetric and that its adjacency graph has undirected edges. We discuss the generalization to structurally unsymmetric matrices at the end of this section.

The example of Figure 1.1a represents a simple undirected adjacency graph that will be used to drive our discussion.



(a) Adjacency graph.

(b) Filled graph.

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

Figure 1.1: Symbolic factorization: predicting the sparsity pattern of the factors.

Symbolic factorization based on elimination graphs

We explain how a sequence of graphs, so-called elimination graphs, can be built to predict the structure of the L matrix and thus perform a symbolic factorization. Although simple and elegant, the elimination graphs are costly to build and to store and not used in practice to compute the symbolic factorization of a symmetric matrix. For an efficient way to compute the row and column structures of the L matrix we recommend the work of Gilbert, Ng and Peyton [37].

Starting from the adjacency graph of the matrix, we describe in the following how this graph can be updated to mimic the symbolic factorization of the matrix.

During symbolic factorization we want to simulate the effect of Gaussian elimination operations during the elementary **Update** operation

$$a_{i,j} \leftarrow a_{ij} - a_{ik} \cdot a_{kj},$$

to detect the position of the new entries a_{ij} in the updated matrix. We assume no numerical cancellation so that the new nonzero entries will remain nonzeros in the matrix of factors. In terms of graph, this translates into the fact that when vertex k is eliminated from the current adjacency graph, all its neighbors become interconnected, *i.e.* a clique is formed. The updated graph is referred to as the *elimination graph*. A clique is defined as a set of vertices that are pairwise connected. For example, in Figure 1.1a, if vertex 1 is eliminated, vertices 2 and 4 becomes interconnected, *i.e.* an edge $(2, 4)$ must be added (as done in red in Figure 1.1b).

We define the filled graph $\mathcal{G}(F)$ as the adjacency graph where all edges that were created during the symbolic factorization have been added (see illustration in Figure 1.1b). Since the new edges correspond to filled entries, the filled graph is the adjacency graph of the factors $F = L + U$ (or $L + L^T$ in our context). The sparsity pattern of the factor matrix L (resp. U) is reported in the lower triangular part (resp. upper triangular part) of Figure 1.1c.

The filled graph and the elimination graphs fully describe the structures that will be processed during factorization. As mentioned before, since they are costly to handle, we will

introduce a simpler tree structure called the elimination tree that can be used to represent in a compact way the mains steps and structures involved during the factorization.

Reordering and permutations

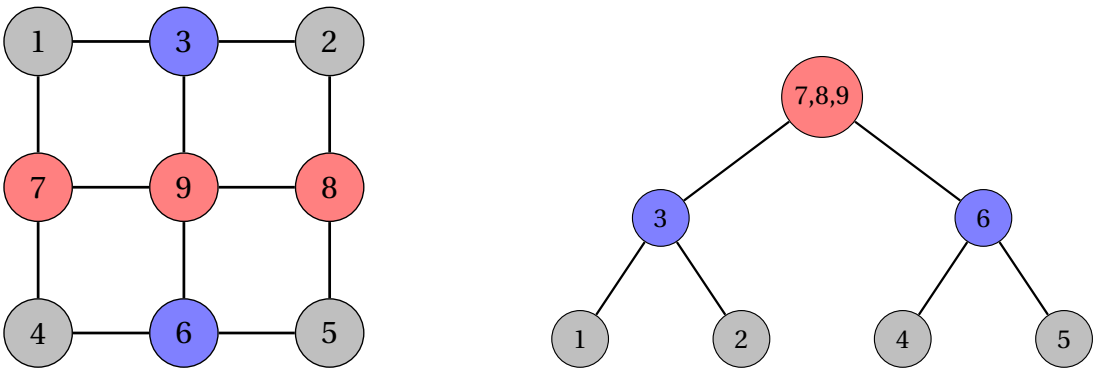
If a matrix can be factorized so that no fill-in occurs, then the elimination ordering on this graph is said to be *perfect*; the graph of F is one such matrix [29, 55]. However, it is usually not the case for the graph of A, $\mathcal{G}(A)$, so that the first step of the analysis phase is to find a good fill-reducing permutation of the pivot variables.

We show that the order in which the variables are eliminated, referred to as the ordering, can significantly influence the fill-in. Obviously, we want to minimize the amount of fill-in, since it increases the computational cost of the factorization and the storage of the matrix of factors; thus, finding a good ordering is a crucial issue to make sparse direct methods effective. For example, [27] shows that fill-in is catastrophic for random matrices which cause the sparse factorization to require $\mathcal{O}(n^2)$ memory and $\mathcal{O}(n^3)$ operations.

However, finding the ordering that minimizes the fill-in is an NP-complete problem [68]. Several heuristic strategies exist, whose effectiveness is matrix-dependent and whose objectives may differ (operation count, parallel performance, memory consumption). We can distinguish:

- Local heuristics, that successively eliminate vertices in an order depending on some local criterion: for example, the vertex of minimum degree (such as AMD [7] or MMD [45]), or the vertex that produces the minimum fill (such as AMF or MMF [50]).
- Global heuristics, such as nested dissection(ND) [34], that recursively partition the graph into subgraphs, or the (Reverse) Cuthill-McKee algorithm [25].
- Hybrid heuristics, which first use a global heuristic to partition the graph, and then apply local heuristics to each subgraph. This is the strategy implemented in several partitioning libraries, such as METIS [43] and SCOTCH [52].

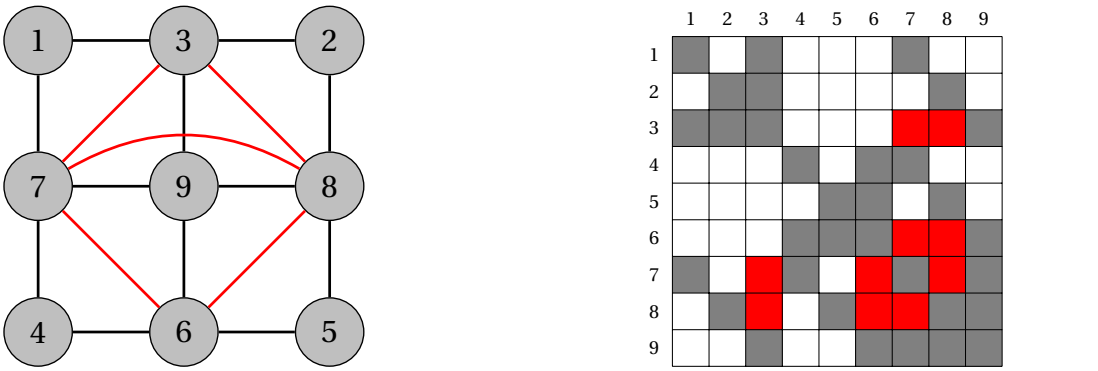
We now briefly review the nested dissection ordering that will be illustrated on a 3D regular mesh in Section 1.3 and analyzed in more details in Section 1.4. It is a fill-reducing ordering well-suited to matrices arising from the discretization of a problem with 2D or 3D geometry. It divides the adjacency graph into a given number of domain subgraphs (or subdomains) separated by a set of vertices called separator. In this section, we consider that the number of subgraphs/subdomains at each step is equal to two, but one can also consider 4 subdomains or 8 subdomains at each step for 2D and 3D regular problems, respectively, as we will do later in this thesis. As a consequence, the vertices of a given subdomain are only connected to other vertices in the same subdomain or in the separator, but not to other subdomains [56]. This way, the elimination of a vertex within a subdomain will not create any fill-in in the other subdomains and the vertices in the separator can be eliminated after the subdomains. The process is then recursively applied to the each subdomain created until the domain subgraphs become too small to be subdivided again. This generates a separator tree. This process is illustrated in Figure 1.2a on our illustrative example where at each iteration, separators divide the domain or subdomains in two subdomains of equal size. This generates an associated separator tree reported in Figure 1.2b.



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

(b) Associated separator tree.

Figure 1.2: Nested dissection on example mesh.



(a) Filled graph.

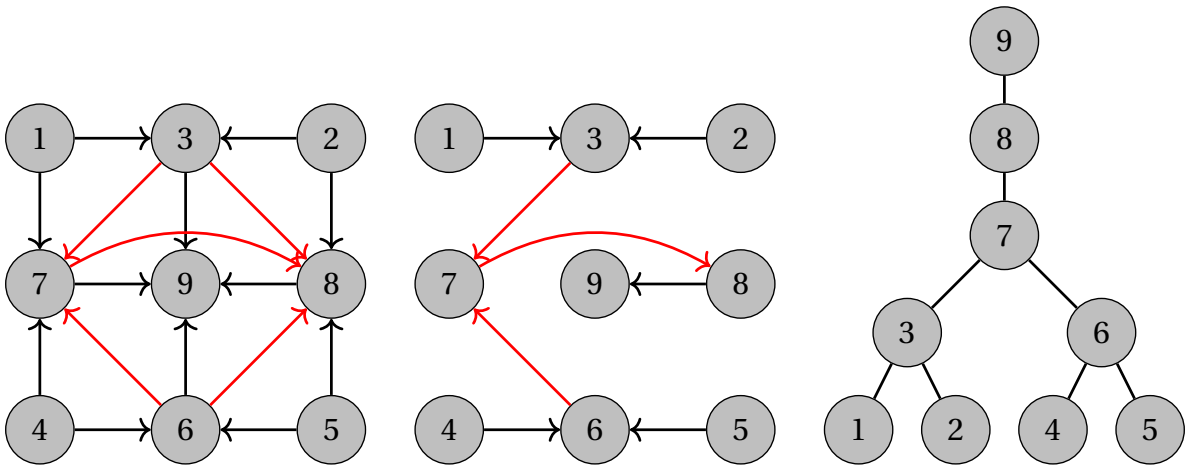
(b) Sparsity pattern of the factors (original nonzeros are in gray, fill-in in red, zeros in white).

Figure 1.3: Symbolic factorization after nested dissection reordering.

In Figure 1.3a and 1.3b, we describe the corresponding filled graph and factor sparsity pattern obtained by performing the symbolic factorization on the reordered matrix. The comparison with Figure 1.1 shows that the fill-in has decreased from 16 to 10 filled entries, which is a considerable improvement compared to the number of nonzeros in the original matrix.

Dependencies between variables and elimination tree

As said previously, the sparse structure of the matrix of factors L depends on the elimination sequence of the pivot variables. Furthermore, the elimination of one variable does not impact all other variables. The study of these dependencies is essential in sparse direct methods to manage the numerical phases. We want to characterize the fact that j depends on i if and only if the elimination of i modifies column j. This will naturally lead to introducing the *elimination tree* a crucial structure in sparse matrix factorization. A detailed study on the role and properties of the elimination tree can be found in [48].



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

Figure 1.4: Construction of the elimination tree.

Definition 1.2 (Vertex dependency)**.** *Let i, j be two vertices such that $i < j$. Vertex j depends on i (noted $i \to j$) if and only if $l_{ji} \neq 0$.*

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

Definition 1.3 (Directed filled graph)**.** *Let the undirected filled graph be $\mathcal{G}(F) = (V, E)$. Then, the directed filled graph $\vec{\mathcal{G}}(F)$ is the graph $(\vec{V}, \vec{E})$ such that:*

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

The directed filled graph thus characterizes the dependencies between vertices. By definition, $(i, j) \in \vec{E} \Rightarrow i < j$ and therefore it cannot have any cycle. It is therefore a directed acyclic graph (DAG) [2]. We can thus introduce the notion of *descendant* and *ancestor* between columns as follows: i descendant of $j \Leftrightarrow j$ ancestor of $i \Leftrightarrow i \to j$. The directed filled graph on our example is given on Figure 1.4a.

The directed filled graph contains however many redundant dependencies: in our example, $1 \to 7$ can be obtained from $1 \to 3$ and $3 \to 7$. We can thus obtain a more compact representation of the vertex dependencies by removing the redundant dependencies, *i.e.*, by computing the transitive reduction. The transitive reduction thus consists in removing edges which can be replaced by a path in the directed filled graph. Because the directed filled graph is a DAG, its transitive reduction is unique [1]. This is illustrated on Figure 1.4b.

The transitive reduction of $\mathcal{G}(F)$ is obviously still a DAG. One key observation here is that its undirected version is still acyclic, *i.e.* it is actually a spanning tree of the directed filled

graph, as illustrated in Figure 1.4c. This tree is referred to as the elimination tree [58] and we note it T.

The advantage of the elimination tree is that it provides a more compact representation of the directed filled graph and to some extent the structure of the factors. Liu [48] shows how to obtain the structure of L from the elimination tree and from the structure of A. The elimination tree also provides a way to express the parallelism inherent to the direct solution of sparse linear systems in the sense that two variables in different subtrees can be computed independently.

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

The unsymmetric case

In the previous sections we have assumed that the matrix is symmetric, or structurally symmetric. The extension to the unsymmetric case is not straightforward. It has been been studied by Gilbert and Liu in [36] who generalized the elimination tree structure. In this context, instead of a tree, the structures that can be used to characterize the dependencies between steps of the factorization are directed acyclic graphs, referred to as *elimination dags* or edags in [36]. The edags can be viewed as the transitive reduction of the directed graphs associated to the L and U matrices. Another possibility is to use the elimination tree structure introduced by Eisenstat and Liu (the theory of elimination trees for sparse unsymmetric matrices, [32]). In this work, the authors extend the notion of elimination tree to unsymmetric matrices. The existence of paths in graphs associated to the L and U factors is used to generalize the elimination tree.

A much simpler approach is to symmetrize artificially the original matrix using the structure of $A + A^T$. Note that in [15], only a partial symmetrization is performed, that makes the use of the elimination tree possible. In both cases, all previous results based on the symmetric structure are applicable. In many applications, the structures of A and A^T are similar and the overhead introduced by the artificial addition of nonzero entries can be limited. This approach was suggested by Duff and Reid [31] and is the one that we will consider in this thesis.

1.2.2 The factorization phase

To efficiently process sparse matrices, we have shown that the original matrix needs to be permuted to reduce the fill-in. The dependency between steps of the elimination process is then captured by the elimination tree that also gives information about the dynamic structures involve during factorization.

A first simple view of the factorization of a sparse matrix is to consider Algorithm 1.1, assume that the original matrix A has already been permuted and process the permuted matrix in order. Fill-in obtained during the update operation will be limited but will have to be stored.

Let us recall that each node of the elimination tree corresponds to the elimination of one pivot variable. Then exploiting the properties of the elimination tree, one can also view the factorization of a sparse matrix as a bottom to top traversal of the elimination tree respecting son to father dependencies. Each elimination of pivot, or equivalently, each elimination of a node of the elimination tree, is then divided into two consecutive operations: **Factor** to eliminate variable i, and **Update** to compute the contributions of this elimination, which are used to

update all ancestor nodes j such that $i \rightarrow j$. In our example, when node 1 is eliminated, nodes 3 and 7 must be updated; when node 3 is eliminated, nodes 7, 8, and 9 are updated.

The independence described by the elimination tree gives a degree of freedom to schedule the elimination of the pivot variables. Once the **Factor** operations are scheduled following the dependencies given by the elimination tree, there exists different ways to schedule the **Update** operations. This results in two different approaches: the *left-looking* and the *right-looking* algorithms. In the right-looking approach, the updates are performed as soon as possible: after the elimination of i, all ancestors j such that $i \rightarrow j$ are updated. On the contrary, in the left-looking approach, the updates are performed as late as possible: all the contributions coming from descendants i such that $i \rightarrow j$ are applied just before the elimination of j. A final possibility that we describe in more detail is the multifrontal method which derives from the right-looking method and make full use of the elimination tree structure to limit the number of direct dependencies between elimination steps.

The multifrontal method was introduced by Duff and Reid [30] as a generalization of the frontal method [42]. They present in this article the first formal and detailed description of the computations and data structures of all phases: analysis, factorization and solve phases. It is directly based on the elimination tree [47, 58] and relies on the following observation: if $i \rightarrow j$, then node j is an ancestor of node i in the elimination tree. As a consequence, a contribution from the elimination of a node i to a node j can be carried through the path from node i to node j in the elimination tree. The local contribution, right term of the Update operation, is computed during the processing of the node but will be passed on the tree, from child to parent up to its destination. The multifrontal method can be described in terms of operations on dense matrices. To each node of the elimination tree is associated a dense matrix, called *frontal matrix* or *front*. Each front is constituted of the variable to be eliminated and the variables that are updated and that correspond to ancestors in the elimination tree.

In practice, the nodes of the elimination tree are grouped together when their associated columns have the same sparsity structure in the reduced matrix. This is referred to as amalgamation and the resulting nodes are called *supernodes*. This can be relaxed to enable grouping nodes that have a similar structure leading to so-called *relaxed supernodes*. The resulting tree is then known as the *assembly tree*. This allows the fronts to have more than one pivot to eliminate at each node. Variables that can be eliminated at a node are referred to as *fully-summed variables* because all contributions from descendants variables associated to descendant nodes in the elimination tree have been summed. The elimination of the fully summed variables of a front takes the form of a partial dense factorization that makes great use of dense linear algebra kernels [44]. The variables (ancestors in the elimination tree) that are only updated during the process of node are called the non fully-summed variables of the corresponding frontal matrix.

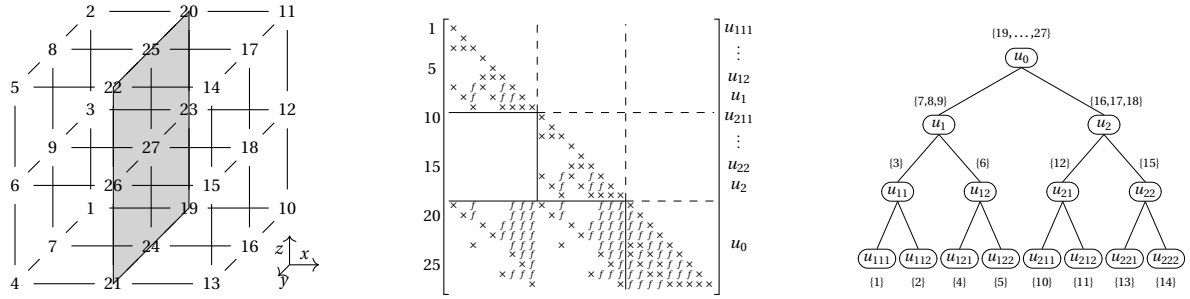
1.3 The solve phase

Once A is factorized, it follows the solve phase on which we give a special focus. The illustration will be based on a 3D example of Figure 1.5 as it lend itself more easily to the description of the exploitation of sparse right-hand sides (RHS) that comes later. Moreover, we now consider the general $A = LU$ decomposition.

Firstly, we discuss the solution phase with a particular focus on the forward substitution and in the general case of a dense RHS. Secondly, we give a detail study of available advances in the litterature concerning the exploitation of sparse RHS.

1.3.1 General case: dense RHS

We introduce a $3 \times 3 \times 3$ domain in Figure 1.5a on which we applied the nested dissection algorithm introduced in Section 1.2.1. The domain is first divided by a 3×3 constant-x plane



(a) $3 \times 3 \times 3$ regular mesh. (b) Structures of A and L. (c) Separator tree T.

Figure 1.5: (a) A 3D mesh with a 7-point stencil. Mesh nodes are numbered according to a nested dissection ordering. (b) Corresponding matrix with initial nonzeros ($\times$) in the lower triangular part of a symmetric matrix A and fill-in (f) in the L factor. (c) Separator tree, also showing the sets of variables to be eliminated at each node.

separator u_0 and each subdomain is then divided recursively. By ordering the separators after the subdomains, we obtain the *separator tree* of Figure 1.5c when choosing supernodes identical to separators[1]. The order of the tree nodes (u_{111}, u_{112}, u_{11}, u_{121}, u_{122}, u_{12}, ..., u_0), partially represented on the right of the matrix, is here a *postordering*: nodes in any subtree are processed consecutively. This order, suitable for memory issues [46], also defines the order in which the nodes will be processed during the forward substitution.

Considering a single RHS b and the decomposition $A = LU$, the solution of the triangular system $Ly = b$ (and $Ux = y$) relies on block operations at each node of the tree T following respectively a bottom-up and a top-down traversal of the tree for the forward and the backward substitutions. Figure 1.6 represents the L and U factors restricted to a given node u of T, where

[1] In this example, identifying supernodes with separators leads to relaxed supernodes: the sparsity in the interaction between u_1 and u_0 (and u_2 and u_0) is not exploited to benefit from larger blocks.

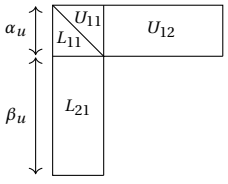


Figure 1.6: Structure of the factors associated to a node u of the tree.

the diagonal block is formed of the two lower and upper triangular matrices L_{11} and U_{11}, and the *update* matrices are L_{21} and U_{12}. The α_u variables are the ones of node (or separator) u, also called earlier the fully-summed variables, and the β_u variables correspond to the nonzero rows in the off-diagonal parts of the L factor restricted to node u (Figure 1.5b), that have been gathered together, called the non fully-summed variables. For example, node u_1 from Figure 1.5 corresponds to separator $\{7, 8, 9\}$, so that L_{11} and U_{11} are of order $\alpha_{u_1} = 3$ and there are $\beta_{u_1} = 9$ update variables $\{19, \ldots, 27\}$, so that L_{21} is of size 9×3 (and U_{12} is of size 3×9). Starting with $y \leftarrow b$, the active components of y are gathered into two temporary dense vectors y_1 of size α_u and y_2 of size β_u at each node u of T, where the triangular solve

$$y_1 \leftarrow L_{11}^{-1} y_1, \tag{1.4}$$

is performed, followed by the update operation

$$y_2 \leftarrow y_2 - L_{21} y_1. \tag{1.5}$$

y_1 and y_2 can then be scattered back into y, and y_2 will be used at higher levels of T. When the root is processed, y contains the solution of $Ly = b$. The backward substitution is very similar to the forward substitution except that the two operations are reversed. Starting from $x \leftarrow y$, the backward update operation at each node u is $x_1 \leftarrow x_1 - U_{12} x_2$ followed by a triangular solve $x_1 \leftarrow U_{11}^{-1} x_1$. x_1 and x_2 are partial dense solution vectors gathering the variables concerned by u and that can be scattered back into a global solution vector x for later use at lower levels of the tree. It is important to note that the dependencies are reversed between forward and backward phase. Although several variants of the solve algorithm may be defined, depending on the way parts of y or x are passed up and down the tree, possibly in a parallel environment [8, 59], those will not impact the study of sparse RHS. Because the matrix blocks in Figure 1.6 are considered dense, there are $\alpha_u(\alpha_u - 1)$ arithmetic operations for the triangular solution (1.4) and $2\alpha_u\beta_u$ operations for the update operation (1.5), leading to a total number of operations

$$\Delta = \sum_{u \in T} \delta_u, \tag{1.6}$$

where $\delta_u = \alpha_u \times (\alpha_u - 1 + 2\beta_u)$ is the number of arithmetic operations at node u.

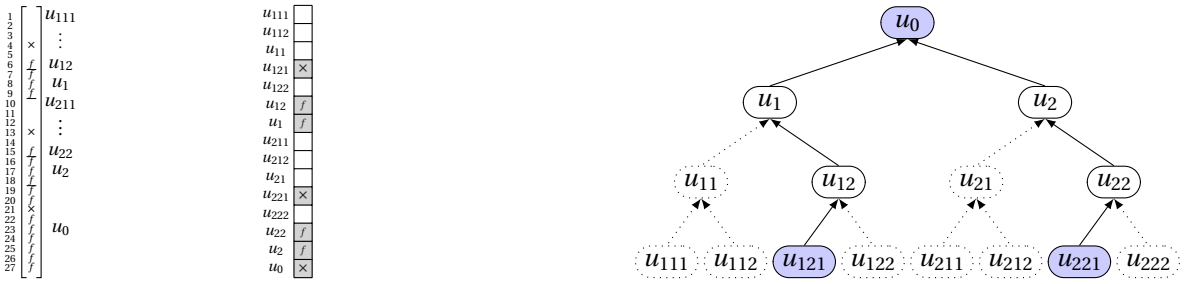
1.3.2 Extension to sparse RHS

As said precedently, we focus our attention on the forward substitution for which we recall the formula:

$$LY = B. \tag{1.7}$$

Structure prediction

In this section, we review two approaches to exploit sparsity in B when solving the triangular system (1.2). The first one exploits a formalism and properties established by Gilbert [35] and Gilbert and Liu [36], called tree pruning in [61], which consists in pruning the nodes at which only computations on zeros are performed. The second approach goes further by working on different sets of RHS columns at each node of the tree [11].



(a) Structure of $L^{-1}b$.

(b) Pruned tree $T_p(b)$.

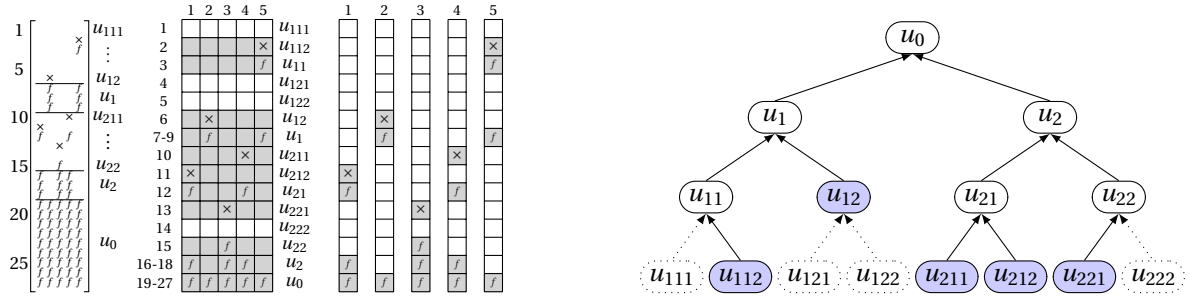
Figure 1.7: Illustration of Example 1.1. (a) Structure of $L^{-1}b$ with respect to matrix variables (left) and to tree nodes (right). $\times$ corresponds to original nonzeros and f to fill-in. In the right part, gray parts of $L^{-1}b$ correspond to the nodes involving computation. (b) Pruned tree $T_p(b)$: pruned nodes and edges are represented with dotted lines and nodes in V_b are filled.

Tree pruning Consider a non-singular $n \times n$ matrix A with a nonzero diagonal, and its directed graph $\mathcal{G}(A)$, with an edge from vertex i to vertex j if $a_{ij} \neq 0$, see Definition 1.1. Given a vector b, let us define $\text{struct}(b) = \{i, b_i \neq 0\}$ as its nonzero structure, and $\text{closure}_A(b)$ as the smallest subset of vertices of $\mathcal{G}(A)$ including $\text{struct}(b)$ without incoming edges. Gilbert [35, Theorem 5.1] characterizes the structure of the solution of $Ax = b$ by the relation $\text{struct}(A^{-1}b) \subseteq \text{closure}_A(b)$, with equality in case there is no numerical cancellation. In our context of triangular systems, ignoring such cancellation, $\text{struct}(L^{-1}b) = \text{closure}_L(b)$ is also the set of vertices reachable from $\text{struct}(b)$ in $\mathcal{G}(L^T)$, where edges have been reversed [36, Theorem 2.1]. Finding these reachable vertices can be done using the transitive reduction of $\mathcal{G}(L^T)$, which is a tree (the elimination tree) when L results from the factorization of a matrix with symmetric (or symmetrized) structure. Since we work with a tree T with possibly more than one variable to eliminate at each node (supernode), let us define V_b as the set of nodes in T including at least one element of $\text{struct}(b)$. The structure of $L^{-1}b$ is obtained by following paths from the nodes of V_b up to the root. The tree consisting of these paths is the *pruned tree* of b, and we denote it by $T_p(b)$. The number of operations Δ from Equation (1.6) now depends on b:

$$\Delta(b) = \sum_{u \in T_p(b)} \delta_u. \tag{1.8}$$

Example 1.1. *Let b be a vector with nonzeros at positions 4, 13, and 21. The corresponding tree nodes are given by $V_b = \{u_{121}, u_{221}, u_0\}$, see Figures 1.5 and 1.7. Following the paths in T from nodes in V_b to the root results in the pruned tree of Figure 1.7b. Compared to $\Delta = 288$ in the case of a dense right-hand side, $\Delta(b) = 228$ ($\delta_{u_{121}} = \delta_{u_{221}} = 6, \delta_{u_{12}} = \delta_{u_{22}} = 12, \delta_{u_2} = \delta_{u_1} = 60, \delta_{u_0} = 72$).*

We now consider the multiple RHS case of Equation (1.2), where RHS columns have different structures and we denote by B_i the columns of B, for $1 \leq i \leq n_{rhs}$. Rather than solving n_{rhs} systems each with a different pruned tree $T_p(B_i)$, we favor matrix-matrix computations by considering $V_B = \bigcup_{1 \leq i \leq n_{rhs}} V_{B_i}$, the union of all nodes in T with at least one nonzero from matrix B, and the pruned tree $T_p(B) = \bigcup_{1 \leq i \leq n_{rhs}} T_p(B_i)$ containing all nodes in T reachable



(a) Structures of $L^{-1}B$, $L^{-1}B_1, \ldots, L^{-1}B_5$.

(b) Pruned tree $T_p(B) = T_p(B_1) \cup \ldots \cup T_p(B_5)$.

Figure 1.8: Illustration of multiple RHS and tree pruning. $\times$ corresponds to an initial nonzero in B and f to "fill-in" that appears in $L^{-1}B$, represented in terms of original variables and tree nodes. Gray parts of $L^{-1}B$ (resp. of $L^{-1}B_i$) are the ones involving computations when RHS columns are processed in one shot (resp. one by one).

from nodes in V_B. The triangular and update operations (1.4) and (1.5) become $Y_1 \leftarrow L_{11}^{-1}Y_1$ and $Y_2 \leftarrow Y_2 - L_{21}Y_1$, leading to:

$$\Delta(B) = n_{rhs} \times \sum_{u \in T_p(B)} \delta_u. \tag{1.9}$$

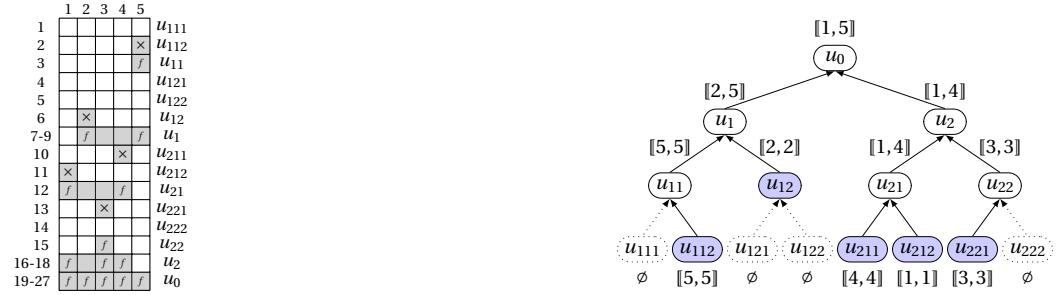
Example 1.2. *Figure 1.8a shows a RHS matrix* $B = [\{B_{11,1}\}, \{B_{6,2}\}, \{B_{13,3}\}, \{B_{10,4}\}, \{B_{2,5}\}]$ *in terms of original variables (1 to 27) and in terms of tree nodes* ($V_B = \{u_{212}, u_{12}, u_{221}, u_{211}, u_{112}\}$). *In Figure 1.8a,* $\times$ *corresponds to an initial nonzero in* B *and* f *corresponds to "fill-in" that appears in* $L^{-1}B$ *during the forward substitution on the nodes that are on the paths from nodes in* V_B *to the root (see Figure 1.8b). We have* $\Delta(B) = 5 \times 264 = 1320$ *and* $\Delta(B_1) + \Delta(B_2) + \ldots + \Delta(B_5) = 744$.

At this point, we exploit tree pruning, or *vertical sparsity*, but perform extra operations by considering globally $T_p(B)$ instead of each individual pruned tree $T_p(B_i)$. Processing B by smaller blocks of columns would further reduce the number of operations at the cost of more traversals of the tree and a smaller arithmetic intensity, with a minimal number of operations $\Delta_{min}(B) = \sum_{i=1,n_{rhs}} \Delta(B_i)$ reached when B is processed column by column, as in Figure 1.8a(right). We note that performing this minimal number of operations while traversing the tree only once (and thus accessing the L factor only once) may require performing complex and costly data manipulations at each node u with copies and indirections to work only on the nonzero entries of $L^{-1}B$ at u.

We now present a simpler approach which exploits the notion of intervals of columns at each node $u \in T_p(B)$. This approach to exploit what we call *horizontal sparsity* in B was introduced in another context [11].

Working with column intervals at each node Given a matrix B, we associate to a node $u \in T_p(B)$ its set of active columns

$$Z_u = \{j \in \{1, \ldots, n_{rhs}\} \mid u \in T_p(B_j)\}. \tag{1.10}$$



(a) Structure of $L^{-1}B$.

(b) Pruned tree $T_p(B)$ with intervals.

Figure 1.9: Column intervals for the RHS of Figure 1.8: in gray (a) and above/below each node (b). Taking for instance u_{21}, there are nonzeros in columns 1 and 4, so that $Z_{u_{21}} = \{1, 4\}$. Instead of performing the solve operations on $n_{rhs} = 5$ columns at u_{21}, computation is limited to the $\theta(Z_{u_{21}}) = 4$ columns of interval $[\![1, 4]\!]$ (and to a single column at, *e.g.*, node u_{221}). Overall, $\Delta(B)$ is reduced from 1320 to 948 (while $\Delta_{min}(B) = 744$).

The interval $[\![\min(Z_u), \max(Z_u)]\!]$ includes all active columns, and its length is

$$\theta(Z_u) = \max(Z_u) - \min(Z_u) + 1.$$

Z_u is sometimes defined for an ordered or partially ordered subset R of the columns of B, in which case we will use the notation $Z_u|_R$ and $\theta(Z_u|_R)$. For u in $T_p(B)$, Z_u is non-empty and $\theta(Z_u)$ is different from 0. As illustrated in Figure 1.9, the idea is then to perform the operations (1.4) and (1.5) on the $\theta(Z_u)$ contiguous columns $[\![\min(Z_u), \max(Z_u)]\!]$ instead of the n_{rhs} columns of B, leading to

$$\Delta(B) = \sum_{u \in T_p(B)} \delta_u \times \theta(Z_u). \tag{1.11}$$

Example 1.3. *In Example 1.2, there are nonzeros in columns 1 and 4 at node u_{21} so that $Z_{u_{21}} = \{1, 4\}$ (see Figure 1.9). Instead of performing the solve operations on all 5 columns at node u_{21}, we limit the computations to the $\theta(Z_{u_{21}}) = 4$ columns of interval $[\![1, 4]\!]$ (and to a single column at,* e.g.*, node u_{221}). Overall, $\Delta(B)$ is reduced from 1320 to 948 (while $\Delta_{min}(B) = 744$).*

It is clear from Example 1.3 that $\theta(Z_u)$ and $\Delta(B)$ strongly depend on the order of the columns in B.

In the next Section, we formalize the problem of permuting the columns of B and evaluate the application of the postorder. Chapter 3 will propose a new permutation and an adapted blocking technique to further decrease the number of operations by identifying and extracting "problematic" columns.

Permuting RHS columns

We showed in Section 1.3.2 that *horizontal sparsity* can be exploited thanks to column intervals. The number of operations to solve (1.2) then depends on the permutation of the columns of B. We express the corresponding minimization problem as:

$$\begin{aligned}&\text{Find a permutation } \sigma \text{ of } \{1,\dots,n_{rhs}\} \text{ that minimizes } \Delta(B,\sigma) = \textstyle\sum_{u\in T_p(B)} \delta_u \times \theta(\sigma(Z_u)),\\ &\text{where } \sigma(Z_u) = \{\sigma(i) \mid i \in Z_u\}\,, \text{ and}\\ &\qquad \theta(\sigma(Z_u)) \text{ is the length of the permuted interval } [\![\min(\sigma(Z_u)), \max(\sigma(Z_u))]\!].\end{aligned} \tag{1.12}$$

If we assume that we work with a balanced tree in terms of computational cost, we reduce the problem to the minimization of the sum of all interval lengths in $T_p(B)$ such that:

We first define the notion of node optimality.

Definition 1.4. *Given a node u in $T_p(B)$, and a permutation σ of $\{1,\dots,n_{rhs}\}$, we say that we have node optimality at u, or that σ is $u-$optimal, if and only if $\theta(\sigma(Z_u)) = \#Z_u$, where $\#Z_u$ is the cardinality of Z_u. Said differently, $\sigma(Z_u)$ is a set of contiguous elements.*

$\theta(\sigma(Z_u)) - \#Z_u$, the number of columns (or padded zeros) on which extra computation is performed, is 0 if σ is $u-$optimal.

Example 1.4. *Consider the RHS structure of Figure 1.9a and the identity permutation. We have node optimality at u_0 because $\#Z_{u_0} = \#\{1,2,3,4,5\} = 5 = \theta(Z_{u_0})$. We do not have node optimality at u_1 and u_2 because the numbers of padded zeros are $\theta(Z_{u_1}) - \#Z_{u_1} = 2$ and $\theta(Z_{u_2}) - \#Z_{u_2} = 1$, respectively. Our aim is thus to find a permutation σ that reduces the difference $\theta(\sigma(Z_u)) - \#Z_u$.*

The postorder permutation In Figure 1.5, the sequence $[u_{111}, u_{112}, u_{11}, u_{121}, u_{122}, u_{12}, u_1, u_{211}, u_{212}, u_{21}, u_{221}, u_{222}, u_{22}, u_2, u_0]$ used to order the matrix follows a postordering.

Definition 1.5. *Consider a postordering of the tree nodes $u \in T$, and a RHS matrix $B = [B_j]_{j=1\dots n_{rhs}}$ where each column B_j is represented by one of its associated nodes $u(B_j) \in V_{B_j}$ (see below). B is said to be postordered if and only if: $\forall j_1, j_2, 1 \le j_1 < j_2 \le n_{rhs}$, we have either $u(B_{j_1}) = u(B_{j_2})$, or $u(B_{j_1})$ appears before $u(B_{j_2})$ in the postordering. In other words, the order of the columns B_j is compatible with the order of their representative nodes $u(B_j)$.*

The postordering has been applied [10, 61, 67] to build regular chunks of RHS columns with "nearby" pruned trees, thereby limiting the accesses to the factors or the amount of computation. It was also experimented together with node intervals [11] to RHS with a single nonzero per column, although it was then combined with an interleaving mechanism for parallel issues.

In Figure 1.9a, B has a single nonzero per column. The initial natural order of the columns (`INI`) induces computation on explicit zeros represented by gray empty cells and we had $\Delta(B) = \Delta(B,\sigma_{\texttt{INI}}) = 948$ and $\Delta_{min}(B) = 744$ (see Figure 1.9). On the other hand, the postorder permutation, $\sigma_{\texttt{PO}}$, reorders the columns of B so that the order of their representative nodes u_{112}, u_{12}, u_{211}, u_{212}, u_{221} is compatible with the postordering. In this case, there are

(a) RHS with one nonzero per column.

(b) RHS with multiple nonzeros per column.

Figure 1.10: Illustration of the postordering of two RHS with one or more nonzeros per column.

no gray empty cells (see Figure 1.10a) and $\Delta(B, \sigma_{\texttt{PO}}) = \Delta_{min}(B)$. More generally, it can be shown that the postordering induces no extra computations for RHS with a single nonzero per column [11].

For applications with multiple nonzeros per RHS, each column B_j may correspond to a set V_{B_j} with more than one node, among which a representative node should be chosen. We describe two strategies.

The first one, called `PO_1`, chooses as representative node the one corresponding to the first nonzero found in B_j (in the natural order associated to the physical problem). The second one, called `PO_2`, chooses as representative node in V_{B_j} the one that appears first in the sequence of postordered nodes of the tree. A comparison of the two postorders with the initial natural order is provided in Table I, for a subset of four problems presented in Table III of Section 1.6. Note that the initial order depends on the physical context of the application and has some geometrical properties.

Table I: Comparison of the number of operations ($\times 10^{13}$) between postorder strategies `PO_1` and `PO_2`.

Δ	`INI`	`PO_1`	`PO_2`	Δ_{min}
H0	.086	.076	**.070**	.050
H3	2.48	1.69	**1.47**	.95
5Hz	.44	.44	**.36**	.22
7Hz	1.46	1.48	**1.21**	.69

Table I shows that the choice of the representative node has a significant impact. The superiority of `PO_2` over `PO_1` is clear and is larger when the number of nonzeros per RHS column is large (problems 5Hz and 7Hz). Indeed, `PO_1` is even worse than the initial order on problem 7Hz.

Example 1.5. *Let* $B = [B_1, B_2, B_3, B_4, B_5, B_6] = [\{B_{1,1}, B_{10,1}, B_{19,1}\}, \{B_{4,2}\}, \{B_{13,3}, B_{15,3}\}, \{B_{2,3}\}, \{B_{5,4}, B_{14,4}, B_{22,4}\}, \{B_{10,5}\}]$ *be the RHS represented in Figure 1.10b. In terms of tree nodes, we have:* $V_{B_1} = \{u_{111}, u_{211}, u_0\}$, $V_{B_2} = \{u_{121}\}$, *etc. Because the rows of* B *have already*

been permuted according to the postordering of the tree, the representative nodes for strategies `PO_1` *and* `PO_2` *are in both cases the nodes* u_{111}, u_{121}, u_{221}, u_{112}, u_{122}, u_{211} *(cells with a bold contour), for columns* $B_1, B_2, B_3, B_4, B_5, B_6$, *respectively. The postorder permutation yields* $\sigma_{PO}(B) = [B_1, B_4, B_2, B_5, B_6, B_3]$, *which reduces the number of gray cells and the volume of computation with respect to the original column ordering:* $\Delta(B) = 1368$ *becomes* $\Delta(B, \sigma_{PO}) = 1242$. *Computations on padded zeros still occur, for example at nodes* u_{211} *and* u_{21} *where* $\theta(\sigma_{PO}(Z_{u_{211}})) = \theta(\sigma_{PO}(Z_{u_{21}})) = 5$ *whereas* $\#Z_{u_{211}} = \#Z_{u_{21}} = 2$.

The quality of $\sigma_{\texttt{PO}}$ depends on the tree postordering. If u_{111} and u_{112} were exchanged in the original postordering, B_1 and B_4 would be swapped, further reducing Δ. One drawback of the postorder permutation is that, since the position of a column is based a single representative node, some information is unused. Other permutations exist that further decrease the number of operations with respect to $\sigma_{\texttt{PO}}$. This motivates a deeper study, that will be the object of Chapter 3.

1.4 Operation count for the solve phase on regular problems with nested dissection

In this section, we provide an analytical formula of the operation count for the solve phase on regular problems when nested dissection is applied. We consider a single right-hand side. We will extend these results to the computation of the asymptotic complexity for sparse right-hand sides and for low-rank representations in Chapter 2. This section relies a lot on the work of George [34] who introduced the nested dissection ordering (see Section 1.2.1) for regular meshes. In particular, we use the same formalism based on meshes and elements, although notation slightly differs.

In [34], George proposed a new ordering strategy for which the number of operations for the factorization of a sparse matrix of order n resulting from an $N \times N$ 2D mesh requires $O(n)$ operations for the factorization and $O(n\ log(n))$ nonzeros in the L factor. The number of operations for the solve phase, directly related to the number of nonzeros in the factors, is thus also of the order of $O(n\ log(n))$. For an $N \times N \times N$ 3D mesh, the number of operations for the factorization is $O(n^2)$ and the number of nonzeros in the L factor (and number of operations for the solve) is $O(n^{4/3})$.

To prove this result, the author draws an interesting parallel between the matrix and the so-called *finite element mesh* in the process of the elimination of variables during the factorization. Using this model, it is possible to compute precisely the number of operations either for the factorization or for the solve phase. We thus propose here to describe the method and then extend it to the computation of formulas on the critical path of the separator tree, with the objective to assess some intrinsic properties of the solve phase.

Consider a 2D mesh of $N \times N$ elements, with the strong assumption that $N = 2^l$, for $l \geq 2$. Any nodal point i corresponds to an unknown x_i while any area delimited by edges of the mesh corresponds to an *element*. This leads to a matrix of order $n = (N + 1)^2$. Furthermore, we consider as [34] a 9-point stencil to connect nodal points. This means that each nodal point in an element is pairwise connected with all other nodal points of that element. In Figure 1.11,

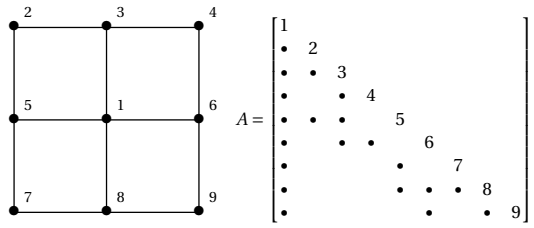


Figure 1.11: 2D mesh and matrix A representing connections between variables.

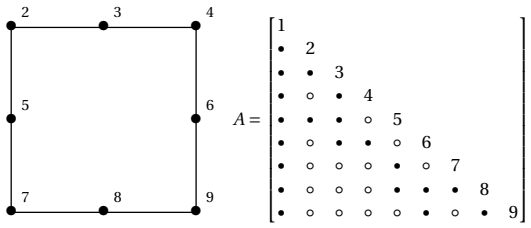


Figure 1.12: Effect of the elimination of variable x_1 on the "fill in" in matrix A.

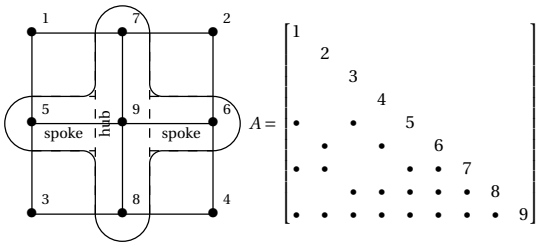


Figure 1.13: Numbering of variables using cross-shaped separators.

we have a mesh composed of $n = 9$ nodal points/unknowns and $N^2 = 4$ elements. In this mesh point of view, two variables are connected if and only if they belong to the same element. In particular, x_1 belongs to all elements so that it is connected to all variables $x_2, \ldots, x_9$ in Figure 1.11 (dense first column of matrix A). Note that the mesh is included in but different from the graph $\mathcal{G}(A)$ introduced in Section 1.2.1 which would, for example, include the edges from x_1 to all other variables.

Instead of the graph view of Section 1.2.1 where edges were added to the graph $\mathcal{G}(A)$ in order to obtain the filled graph of the factors $\mathcal{G}(F)$, the elimination process is now modeled in terms of mesh operations: the elimination of one variable induces the removal of the associated nodal point *and* all its connected edges, forming a new element. For example, Figure 1.12 shows the effect of the elimination of variables x_1 on the fill-in. The elimination of variable x_1 implies that all variables belong now to the same element and thus become pairwise connected, resulting in the matrix structure of Figure 1.12.

The nested dissection process as presented by [34] is a divide-and-conquer strategy, in which separators take the form of crosses built with two "spokes" and one "hub", so that there are four subdomains per separator, as shown in Figure 1.13. The variables corresponding to the subdomains created by the separator are numbered first (corner variables x_1, x_2, x_3, x_4). The numbering of the separator variables orders first the variables of the spokes and then the variables of the hub. When subdomains are large enough, the process is repeated recursively within each separator.

We now mention and illustrate in Figures 1.14 and 1.15 two different sequences to eliminate separator variables in corner elements. We note that a corner is composed of $2^k \times 2^k$ nodal points ($2 < k < l$, with $k = 1, 2$ treated as special cases), that the separator cannot divide the corner in four exactly equal parts, and that it is better to have $2^{(k-1)} \times 2^{(k-1)}$ nodal points in the bottom-left corner at the next level (and only $(2^{(k-1)} - 1) \times (2^{(k-1)} - 1)$ on the opposite side).

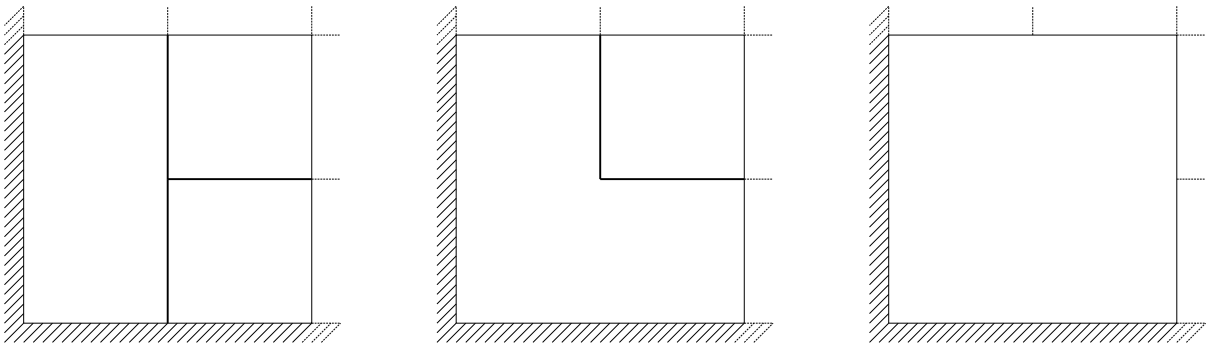


Figure 1.14: Elimination sequence of spokes in [34] for a bottom left corner of a 2D mesh. Extern spokes are eliminated first.

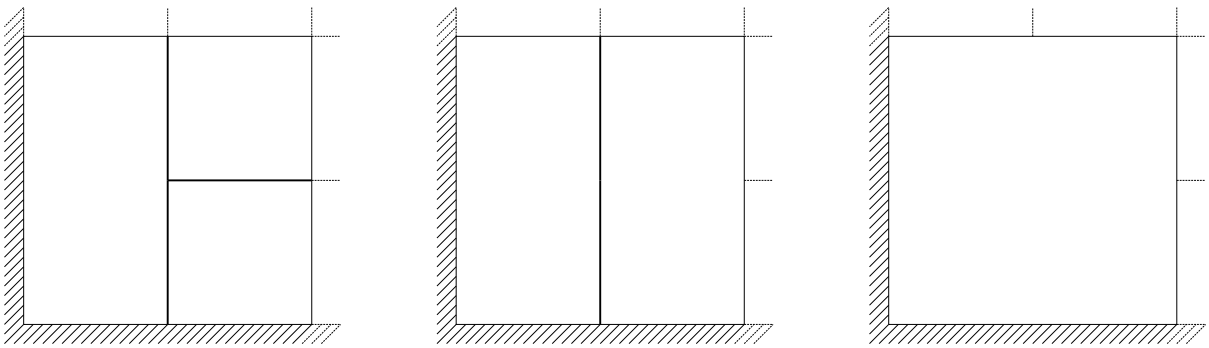


Figure 1.15: Elimination sequence of spokes in a classical nested dissection algorithm.

Figure 1.14 corresponds to the sequence presented in [34]. It numbers first the two spokes that are closest to the border. Figure 1.15 corresponds to the classical sequence of elimination usually implemented in nested dissection ordering packages and numbers horizontal spokes before the vertical spoke, although this leads to slightly more operations. In the following, the first one will be referred to as the "George" sequence and the second one as the "classical" sequence[2].

Computation of the number of operations

With such a construction, George gives in [34, Lemma 2.3] the number of operations and the factor storage associated to the elimination of a spoke or a hub Q. During the elimination process, such a spoke or hub forms part of an element and thus forms a clique, and it is fully connected to the other variables of the elements it belongs too, called R. Focusing on storage, the part of the L factor associated to the corresponding q eliminated variables and update of r eliminated unknowns is: $s(q,r) = \frac{q(q+2r-1)}{2}$. We note that this formula corresponds to the number of entries in the L factor represented in Figure 1.6, with $q = \alpha_u$ and $r = \beta_u$. It leads to a number of operations for the solve equal to $\delta_u = \alpha_u \times (\alpha_u - 1 + 2\beta_u)$, considering one addition and one multiplication per non-diagonal factor entry in L, as discussed at the end of Section 1.3.1. (In a multifrontal context, $\alpha_u = q$ is also the number of fully summed variables, and β_u the number of non-fully summed variables at a given node of the elimination tree.) Remark that Lemma 2.3 of [34] is applied first to the spokes and then to the hub. It is not applied directly on the unknowns of an entire cross-shaped separator because those are not fully connected.

Example 1.6. *Consider the elimination of the cross-shaped separator of Figure 1.13. Variables x_1, x_2, x_3 and x_4 already eliminated. Then, the first spoke x_5 is connected to x_7, x_8 and x_9. As a consequence, the number of off-diagonal entries in the column of L associated to x_5 is $s(1,3) = 3$ and the number of operations is $\delta_u(1,3) = 6$. The same quantities are obtained for the second spoke. The variables of the hub are connected to no other variables, leading to $\delta_u(3,0) = 4$.*

The same approach can be used to compute the operation count on any separator. Thanks to the strong assumption on the mesh size, $N = 2^l$, one may determine exactly the operation count on any separator. The regularity of the mesh and the recursivity of the nested dissection create sets of separators with equal shapes for which the number of operation can be efficiently computed (after dealing with separators located on the borders of the domains), leading to an analytical expression of the total number of operations Δ as a function of the mesh size N. All materials for a formal proof can be found in [34] and we provide here only the formulas. Because we identified two elimination sequences ("George" and "classical"), we provide in the following two sets of equations:

[2]For elements not in a corner but on a border of the mesh, George eliminates first the spoke closest to the border, then the one on the other side and finally the one parallel to the border. We consider the same is true for the "classical sequence", as this leads to slightly less operations.

Table II: Ratio $\frac{\Delta^{Classical}(N)}{\Delta_c^{Classical}(N)}$ when N tends to ∞.

$\frac{\Delta}{\Delta_c}$	Facto	Solve
$N \times N$ 2D-mesh	13.74	$2.98 \log_2 N$
$N \times N \times N$ 3D-mesh	3.95	7.07

Theorem 1.1. *Let $N = 2^l$ be the 2D-mesh size, with $l \geq 2$. Then, the number of operations (multiplications and additions) to perform the solve phase is*

$$\Delta^{George}(N) = \frac{31}{2}\log_2(N)N^2 - \frac{146}{3}N^2 + 48\log_2(N)N + 6N + 8\log_2(N) + \frac{176}{3} \quad (1.13)$$

in the case of George's elimination sequence, and

$$\Delta^{Classical}(N) = \frac{31}{2}\log_2(N)N^2 - 48N^2 + 48N\log_2(N) + 6N + 8\log_2(N) + 48 \quad (1.14)$$

in the case of a classical elimination sequence.

The two formulas have been computed using the formal computation tool Maple. We note that the correct elimination of the corners is useful to slightly decrease the total number of operations, but this only affects the lower order terms. In the rest of this thesis, we used a classical nested dissection ordering when dealing with regular problems.

Computation of the critical path

We are also interested in the number of operations on the critical path noted Δ_c of the assembly/separator tree and we have extended this work for its computation.

We identify the critical path as the path in the separator tree containing most of the operations. As said precedently, using finite element mesh representation of [34], the critical path is composed of sets of spokes and hubs. To identify them, we chose the ones located on the interior of the domain, for which their number r of connected variables in R is bigger than for the spokes and hubs on the borders. Using this approach and Maple again, we obtain for $l \geq 2$:

$$\Delta_c^{Classical}(N) = \frac{125}{24}N^2 - 10N + 4\log_2(N) + \frac{38}{3}. \quad (1.15)$$

We now divide the total amount of work by the work on the critical path. Since we also computed formulas for the 3D case (with a 27-point stencil instead of a 9-point stencil in this case) and for the factorization, we provide in Table II the result when N tends to infinity for the different cases, including the constants.

We distinguish two different ways to interpret the ratio reported in Table II:

- first, as a theoretical speed-up and thus a metric for tree parallelism;
- second, as a lower bound of the gain that one would obtain when exploiting RHS sparsity.

With the first interpretation, the ratio represents the speed-up of the solve phase when using an infinite number of processors, when using tree parallelism only. It thus provides a simple measure of the available tree parallelism. The second assertion will be discussed and pushed further together with the use of low-rank approximations in Chapter 2. As a brief introduction, a sparse RHS, as described in Section 1.3.2, may be represented by a pruned tree T_p that is sufficiently narrow to be considered as a branch of the tree T, at least asymptotically.

First, we see in Table II that the ratio (theoretical speed-up and gain due to RHS sparsity) is always higher for the solve than for the factorization phase, and may even be asymptotically proportional to $\log_2 N$ in the 2D case. Second, from 2D to 3D regular problems, we observed a decrease of the ratio, in other words, a loss of tree parallelism. Indeed, 3D problems tend to have larger top nodes (size proportional to N^2 rather than N in the 2D case) that become predominant.

We now provide a short introduction to low-rank approximations, that we will use in Chapter 2 to compute asymptotic complexities of the solve phase in both full-rank and low-rank settings, for different models of sparse RHS.

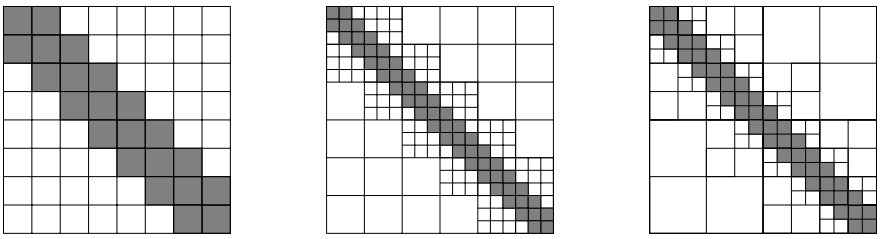
1.5 Low-rank matrix formats

In many applications requiring the solution of a dense linear system $Ax = b$, such as the solution of discretized PDEs, the matrix A has been shown to have a low-rank property [19]: its off-diagonal blocks have low numerical rank, i.e., they can be well approximated by matrices of small rank r.

Several formats have been proposed to exploit this property depending on how the block partitioning of the matrix is computed. Let us consider a dense matrix S of order m. In our context the dense matrix S will be related to the dense frontal matrix processed at each step of the mulitfrontal factorization. The simplest format is the block low-rank (BLR) format [3]. It partitions the dense matrix S with a flat, 2D blocking and approximates its off-diagonal blocks by low-rank submatrices, as illustrated in Figure 1.16a. Compared with the quadratic $\mathcal{O}(m^2)$ cost of storing S as a full-rank (FR) matrix, storing its BLR representation only requires $\mathcal{O}(m^{3/2}r^{1/2})$ entries [5]. One may find in [49, 64] an exhaustive description of the method.

More advanced formats are based on a hierarchical partitioning of the matrix: matrix S is partitioned with a 2×2 blocking and the two diagonal blocks are recursively refined, as illustrated in Figure 1.16c. Different hierarchical formats can be defined depending on whether the off-diagonal blocks are directly approximated (so-called weakly-admissible formats) or further refined (so-called strongly-admissible formats). The most general of the hierarchical formats is the strongly-admissible $\mathcal{H}$-matrix format [39]; the HODLR format [17] is its weakly-admissible counterpart. These hierarchical formats have a near-linear storage complexity $\mathcal{O}(mr \log m)$. The log factor can be removed by using a so-called nested-basis structure. The strongly-admissible $\mathcal{H}^2$-matrix format [22] and the weakly-admissible HSS [66] format exploit such nested basis structures to achieve linear complexity $\mathcal{O}(mr)$.

More recently, a multilevel BLR (MBLR) format [16] has been proposed to bridge the gap between flat and hierarchical formats. As illustrated in Figure 1.16b, it aims at finding a compromise between the simplicity of the BLR format and the low complexity of the hierarchical



(a) BLR partitioning. (b) MBLR partitioning. (c) $\mathcal{H}$ partitioning

Figure 1.16: Illustration of different low-rank formats of a dense matrix S of order m. Gray blocks are stored in full-rank whereas white ones are approximated by low-rank matrices.

ones. By setting the number of levels used in the block hierarchy to a given constant parameter $\ell \geq 2$, its storage complexity can be easily controlled and is equal to $\mathcal{O}(m^{(\ell+2)/(\ell+1)}r^{\ell/(\ell+1)})$.

In this thesis, we will show the impact of low-rank formats on the complexity of the solve phase with sparse RHS in Chapter 2, and present a few results using a block-low-rank solve implementation in Chapter 5.

1.6 Motivations and experimental environment

1.6.1 Experimental environment and computational systems

We first describe computational systems on which we have run our experiments:

- `EOS`: CALMIP supercomputer EOS[3], which is a BULLx DLC system composed of 612 computing nodes, each composed of two Intel Ivybridge processors with 10 cores (total 12 240 cores) running at 2.8 GHz, with 64 GBytes of memory per node.

- `brunch`: machine from the LIP laboratory which has 4 "Broadwell" CPUs (24 cores each) running at 2.2GHz and 1.5TB of memory.

Moreover, all experiments are performed with the MUMPS[4] solver [9, 12]. The MUMPS solver is a parallel direct solver that implement the multifrontal method. It relies on the elimination tree of the matrix to be factored; in case A is unsymmetric, the pattern of $A + A^T$ is used. We refer the reader to the MUMPS User's Guide for more details on the available functionalities.

Regarding parallelism, MUMPS was initially designed for distributed-memory systems but also uses multithreaded BLAS and OpenMP in shared memory environments and on clusters of shared-memory nodes. Then, it takes advantage of both tree and node parallelism: first, it uses the inherent parallelism coming from the properties of the elimination tree; second, it also

[3] https://www.calmip.univ-toulouse.fr/
[4] www.http://mumps.enseeiht.fr

Table III: Characteristics of the $n \times n$ matrix A and $n \times n_{rhs}$ matrix B for different test cases. $D(A) = nnz(A)/n$ and $D(B) = nnz(B)/n_{rhs}$ represent the average column densities for A and B, respectively.

application	matrix	$n(\times 10^6)$	$D(A)$	sym	n_{rhs}	$D(B)$
seismic modeling	5Hz	2.9	24	no	2302	567
	7Hz	7.2	25	no	2302	486
	10Hz	17.2	26	no	2302	486
electro-magnetism modeling	H0	.3	13	yes	8000	9.8
	H3	2.9	13	yes	8000	7.5
	H17	17.4	13	yes	8000	6
	H116	116.2	13	yes	8000	6
	S3	3.3	13	yes	12340	19.7
	S21	20.6	13	yes	12340	9.5
	S84	84.1	13	yes	12340	8.6
	D30	29.7	23	yes	3914	7.6

distributes sufficiently large nodes following a 1D row-wise partitionning: the so-called *master* holds the fully-summed rows to be eliminated while the off-diagonal block is distributed among *slave* processes. The adaptation of distributed-memory parallelism to target the performance of the solve phase (rather than the factorization phase) will be illustrated in Chapter 4.

1.6.2 Applications

As said previously, the solve phase with a large number of RHS can be a bottleneck. It is even more the case when low-rank approximation are used because the reduction of the operation count is larger for the factorization than for the solve. In the following, we describe two types of applications (geophysics and electromagnetism) which offer the characteristics discussed previously: many RHS and sparse RHS. We first give a description of these two applications and then motivate our work.

The two applications we used were composed of systems of different sizes and different numbers of RHS. The characteristics of these systems are gathered in Table III.

The first application corresponds 3D seismic modeling based on the Full-waveform Inversion [63] and that is intensively used in the oil industry as part of a seismic imaging workflow. The matrices were provided by the the Geoscience Azur laboratory and are used in the SEISCOPE Consortium. Each matrix corresponds to the finite-difference discretization of the Helmholtz equation at a given frequency (5, 7, and 10 Hz). It has been shown, in a collaboration between the MUMPS group and the SEISCOPE consortium, that low-rank approximation can be efficiently used in this context to speed up the factorization [4]. We mention that the exploitation of sparsity in [4] is used to speed up the forward substitution.

The second application which will be the subject of an extensive study in Chapter 4 is a 3D electromagnetic modeling applied to marine Controlled-Source Electromagnetism (CSEM)

Table IV: Resolution times (analysis has been ommitted) on one matrix from each model presented (EMGS and SEISCOPE). 90 MPI $\times$ 10 OMP on the EOSsupercomputer.

	T_f	T_s	T_{tot}
10Hz	267 (38%)	439 (62%)	706
S21	476 (6%)	7819 (94%)	8295

surveying, a widely used method for detecting hydrocarbon reservoirs and other resistive structures embedded in conductive formations. Matrices were provided by the EMGS company (Norway). They are built through a finite-difference discretization of frequency-domain Maxwell equations. During data acquisition from CSEM surveying, transmitters and receivers are placed on a large surface above the seafloor and thus induce the resolution of large sparse linear systems with multiple RHS, up to 10000. [60] showed the efficiency of the application of a BLR solver to reduce the factorization time but increased the importance of the solve time.

We now motivate the contributions of this thesis by showing in Table IV the times for factorization (T_f) and solve (T_s) and their proportion on the whole resolution. In Table IV, sparsity is not exploited during the solution phase, and the runs were done with a version of MUMPS exploiting low-rank compression during the factorization phase but not during the solve phase. For each application, the BLR accuracy was chosen to match the application requirements.

We clearly see that the solve phase is the bottleneck in such applications since it takes up to 93% of the total resolution time.

1.6.3 Outline of the thesis

This thesis focuses on the reduction of the cost of the solve phase in the presence of sparse, multiple, right-hand sides. In Chapter 2, we evaluate the theoretical implication of exploiting RHS sparsity. Then we improve state-of-the-art algorithms to order and organize computations on multiple sparse RHS in Chapter 3 and finally concentrate on the practical impact of RHS sparsity in parallel environments on the CSEM application mentioned above in Chapter 4.

Chapters 2, 3 and 4 can be read independently from eachother.

We will also see that in these chapters, although the results obtained most often focus on the exploitation of RHS sparsity in the forward substitution, they can all be transposed to the backward substitution when only part of the solution is requested.

Chapter 2

On the complexity of the solution phase with sparse right-hand sides

2.1 Introduction

In this chapter, we consider applications with single and multiple sparse right-hand sides (RHS) on regular problems of size $n = N \times N$ in 2D and $n = N \times N \times N$ in 3D, where N is the mesh size. As originally presented in [36] and discussed in the introduction of this thesis, RHS sparsity can be exploited to reduce the cost of the forward substitution of sparse direct solvers. We will also present new practical approaches to exploit sparsity of multiple RHS in Chapters 3 and 4, but focus here on complexity issues.

Specifically, the question we aim at answering is whether exploiting the RHS sparsity improves the asymptotic complexity of the solution phase or leaves it unchanged. Table II in Section 1.4 provides a first partial answer, in the case of a full-rank solver (for one RHS with a single nonzero). However, we consider a more general setting in this chapter, and include the case of compressed factor storage thanks to low-rank representations (see Section 1.5).

If $\mathcal{C}(n)$ and $\mathcal{C}^{ES}(n)$ are respectively the complexities of the solve phase with and without the exploitation of RHS sparsity, then we examine in this chapter the asymptotic behavior of the gain noted $\mathcal{G}^{ES}(n)$ and expressed as:

$$\mathcal{G}^{ES}(n) = \frac{\mathcal{C}(n)}{\mathcal{C}^{ES}(n)}. \tag{2.1}$$

In this chapter, we sometimes express these quantities as a function of N and also use the notation $\mathcal{G}^{ES}(N) = \frac{\mathcal{C}(N)}{\mathcal{C}^{ES}(N)}$, with $N = n^{\frac{1}{2}}$ or $N = n^{\frac{1}{3}}$ for 2D and 3D problems, respectively. Because of the division, we need asymptotic expressions bounded by above and below, and use for that the big-Θ notation, rather than the big-$\mathcal{O}$ notation[1]. The exploitation of RHS sparsity was presented in Section 1.3.2 and is shortly discussed in Section 2.2.2 for our purpose. In this chapter, assuming a given storage complexity for dense matrices that depends on the full-rank or low-rank format, we first compute $\mathcal{C}^{ES}(n)$ for *one* RHS and extend the results to the case of

[1] By definition, $f(n) = \Theta(g(n))$ iff $\exists C_1, C_2, n_0 > 0,\ \forall n > n_0, C_1 g(n) \leq f(n) \leq C_2 g(n)$.

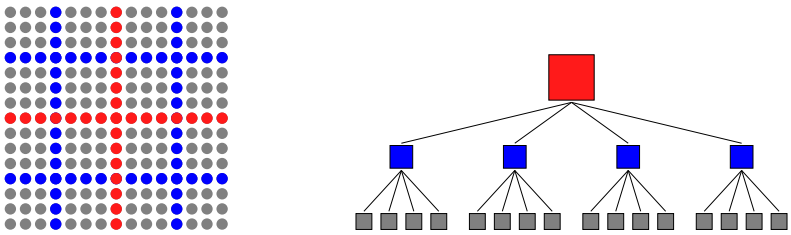


Figure 2.1: Nested dissection with cross-shaped separators and its corresponding separator tree.

multiple RHS with *multiple* nonzeros to give a realistic model of the complexity and properties of the Equation (1.2).

This chapter is organized as follows. In Section 2.2, we present general complexity formulas of the solve phase, recall how RHS sparsity can be exploited and present model problems that will be used to validate our theoretical results. We prove in Section 2.3 that $\mathcal{G}^{ES}$ is constant or nearly constant in the case of a full-rank unstructured solver, whereas it increases with n in the case of a low-rank solver. Specifically, the gain due to the RHS sparsity is of order $\Theta(\log n)$ in 2D and $\Theta(1)$ in 3D in the full-rank case, whereas in the low-rank case, it can be as high as $\Theta(n^{1/2})$ in 2D and $\Theta(n^{1/3})$ in 3D. In Section 2.4, we then illustrate these complexity results with some practical experiments on two real-life applications. In Section 2.5, we consider the interpretation of the results obtained (Equation (2.1)) as a metric for tree parallelism so that all demonstrated results also give insights on the inherent parallel properties of the solve phase. Finally, we provide our concluding remarks in Section 2.6; in particular, we discuss how our theoretical results can also apply to the backward substitution when only part of the solution is needed, i.e., when only a sparse subset of X is required.

2.2 Preliminaries

2.2.1 Nested dissection and complexity formulas

As discussed in Sections 1.2.1 and 1.4, a widely used approach to limit the size of the factors (which is proportional to the number of operations in the solve phase), is the nested dissection ordering [34]. In terms of graph, we recall that it divides the adjacency graph associated with A into s domain subgraphs separated by a separator subgraph. The process is then recursively applied to the s domain subgraphs until they become too small to be subdivided again. This generates a separator tree, as illustrated in Figure 2.1 with $s = 4$ in the case of a 2D regular domain.

Both the factorization and solution phases then consist in a traversal of the separator tree where, at each node of the tree, dense operations are performed on the unknowns associated with the corresponding separator. To be specific, the forward and backward substitutions take the form of a bottom-up and top-down traversals, respectively. Since at each node, the dense operation that is performed is a triangular solve, the complexity of the solution phase will be directly derived from the complexity $\Theta(m^\alpha)$ of storing a given separator of m unknowns. If the separator is stored in full-rank (FR) format, its storage complexity is $\Theta(m^2)$. If it is

Table I: Complexity of the solution phase for a sparse system of $N \times N$ (2D case) or $N \times N \times N$ (3D case) unknowns, assuming a storage complexity $\Theta(m^\alpha)$.

$\mathcal{C}_{2D}(N)$		$\mathcal{C}_{3D}(N)$	
$\alpha > 2$	$\Theta(N^\alpha)$	$\alpha > 3/2$	$\Theta(N^{2\alpha})$
$\alpha = 2$	$\Theta(N^2 \log N)$	$\alpha = 3/2$	$\Theta(N^3 \log N)$
$\alpha < 2$	$\Theta(N^2)$	$\alpha < 3/2$	$\Theta(N^3)$

approximated by a low-rank matrix format, its complexity depends on which format is used and has been given in Section 1.5 of Chapter 1.

Then, the overall complexity of the solution phase is the sum of the storage complexities over all separators. This in turn depends on the shape and size of the physical domain.

- For a two-dimensional (2D) problem of $n = N \times N$ unknowns, the separators are crosses whose size begins at $2N$ at the root of the tree and is then divided by two at each level. Separators at level k of the tree are therefore of size $\Theta(N/2^k)$. Moreover, each separator subdivides the domain into $s = 4$ subdomains and thus there are 4^k nodes at level k.
- Similarly, for a three-dimensional (3D) problem of size $n = N \times N \times N$ unknowns, there are 8^k hypercross separators of size $\Theta(N^2/4^k)$ at level k of the separator tree.

When eliminating a cross-shaped separator, spokes are eliminated first and the hub is eliminated last (see description of George's algorithm in Section 1.4), exploiting some sparsity in the separator. Since we are interested here in the asymptotic complexity (without the constants), it is sufficient to consider for each separator the factorization of a dense matrix of size proportional to that of the entire cross separator. Therefore, the complexity of the solution phase is given by

$$\mathcal{C}_{2D}(N, \alpha) = \sum_{k=0}^{K_{2D}} \Theta\left(4^k \left(N/2^k\right)^\alpha\right), \tag{2.2}$$

$$\mathcal{C}_{3D}(N, \alpha) = \sum_{k=0}^{K_{3D}} \Theta\left(8^k \left(N^2/4^k\right)^\alpha\right), \tag{2.3}$$

where $K_{2D} = \Theta(\log_2 N) = \Theta(\log_4 N^2) = K_{3D}$ denote the total number of levels in the separator tree in 2D and 3D, and where α defines the storage complexity of the format used to represent the separators. (2.2) and (2.3) are geometric series of common ratio $q = 2^{2-\alpha}$ and $q = 2^{3-2\alpha}$, respectively. Using

$$\sum_{k=0}^{K} q^k = \begin{cases} \Theta(K) \text{ if } q = 1, \\ \frac{1-q^{K+1}}{1-q} = \begin{cases} \Theta(1) \text{ if } q < 1, \\ \Theta(q^{K+1}) = \Theta(q^K) \text{ if } q > 1, \end{cases} \end{cases}$$

we can easily compute the 2D and 3D solution complexities depending on the value of α. We report the result in Table I.

Note that when considering low-rank matrix formats, the value of α depends on the asymptotic dependence of the rank r with respect to n. For example, consider a given format with

Table II: Complexity of the solution phase of a sparse system of $N \times N$ (2D case) or $N \times N \times N$ (3D case) unknowns, depending on which of the FR, BLR, MBLR, and hierarchical formats is used and depending on the rank bound r.

	$r = \Theta(1)$			$r = \Theta(m^{1/2})$		
	α	$\mathcal{C}_{2D}(N)$	$\mathcal{C}_{3D}(N)$	α	$\mathcal{C}_{2D}(N)$	$\mathcal{C}_{3D}(N)$
FR	2	$\Theta(N^2 \log N)$	$\Theta(N^4)$	2	$\Theta(N^2 \log N)$	$\Theta(N^4)$
BLR	$3/2$	$\Theta(N^2)$	$\Theta(N^3 \log N)$	$7/4$	$\Theta(N^2)$	$\Theta(N^{7/2})$
MBLR	$(\ell+2)/(\ell+1)$	$\Theta(N^2)$	$\Theta(N^3)$	$(3\ell/2+2)/(\ell+1)$	$\Theta(N^2)$	$\Theta(N^{(3\ell+4)/(\ell+1)})$
Hier.	1	$\Theta(N^2)$	$\Theta(N^3)$	$3/2$	$\Theta(N^2)$	$\Theta(N^3 \log N)$

storage complexity $\Theta(m^\beta r^\gamma)$. If $r = \Theta(1)$ does not depend on m (e.g., a Poisson problem), then $\alpha = \beta$. However, if $r = \Theta(m^{1/2})$ (e.g., a Helmholtz problem), then $\alpha = \beta + \gamma/2$.

The complexity of the solution phase is given for two types of rank bounds and for the FR, BLR [5], MBLR [16], and hierarchical formats in Table II. Note that throughout this chapter, we do not need to distinguish between the different types of hierarchical low-rank formats ($\mathcal{H}$, $\mathcal{H}^2$, HSS, etc.) since the logarithmic terms in m in the storage complexity do not impact the complexity of the sparse solution phase.

2.2.2 Exploiting the RHS sparsity

Along with the multiplication of RHS, electromagnetic or seismic applications often feature RHS characterized by their locality, in other words, their nonzero structure could be very sparse (tens or few hundreds of nonzero per RHS column). We recall from Chapter 1 how to exploit this sparsity and how this translates into the computation of a single branch of the separator tree.

For $Ax = b$ where b is a sparse vector, it is shown in [35] that the nonzero structure of b after the forward substitution can be predicted a priori, and demonstrated in [36] that we can reduce the computation to the set of meaningful variables that are on paths of the separator tree from the nodes corresponding to initial nonzero variables to the root node. This was also referred to as *tree pruning* in [61], or *vertical sparsity* in Chapter 1. Therefore, if the number of nonzero variables is limited, the exploitation of RHS sparsity amounts to the traversal of a sufficiently small ($\Theta(1)$) number of branches of the separator tree.

For $AX = B$, where B is a sparse matrix, the computation of $\mathcal{C}^{ES}$ in the previous context is not straightforward. Indeed, the optimality that we had in the case of a single RHS may vanish in the case of multiple RHS. This is due to the definition of tree pruning, see Chapter 1. In other words, the complexity $\mathcal{C}^{ES}(N, n_{rhs})$ may not be equal to n_{rhs} times the complexity of one branch $\mathcal{C}^{ES}(N)$. Indeed, the worst case scenario would be to solve a problem where the pruned tree would be equal to the initial tree. Then, we would have $\mathcal{C}^{ES}(N, n_{rhs}) = \mathcal{C}(N, n_{rhs})$. To overcome this problem, one optimal approach is the so-called *sequential* approach, in which we sequentially process the RHS one by one. Only then, we may conclude that $\mathcal{C}^{ES}(N, n_{rhs}) = n_{rhs} \times \mathcal{C}^{ES}(N)$. Because we also have $\mathcal{C}(N, n_{rhs}) = n_{rhs} \times \mathcal{C}(N)$, we may conclude the

following:

$$\mathcal{G}^{ES}(N, n_{rhs}) = \frac{\mathcal{C}(N, n_{rhs})}{\mathcal{C}^{ES}(N, n_{rhs})} = \frac{n_{rhs} \times \mathcal{C}(N)}{n_{rhs} \times \mathcal{C}^{ES}(N)} = \frac{\mathcal{C}(N)}{\mathcal{C}^{ES}(N)} = \mathcal{G}^{ES}(N) \tag{2.4}$$

This approach however does not exploit BLAS-3 operations, which are typically an order of magnitude faster than their BLAS-2 counterpart. We thus wish to process multiple RHS by groups of enough columns (say, 256 or 512), although this introduces operations on zeros. To limit the number of operations on zeros and stay close to the minimal number of operations, one should exploit horizontal sparsity, permutations (e.g. postorder) or blocking techniques, as discussed in Chapter 1. In Chapter 3, new RHS permutations and blocking techniques will be shown to lead to a number of operations that can be arbitrarily close to that of the sequential approach, so that Equation (2.4) will hold for multiple right-hand sides. We will develop our complexity study for increasingly general models of sparse RHS in Section 2.3.

2.2.3 Model problems and experimental setting

For the complexity experiments described in the next section, we have used standard Poisson and Helmholtz operators. The resulting matrices exhibit low-rank properties and will be used to give an experimental validation of the complexity results obtained.

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

$$\Delta u = f$$

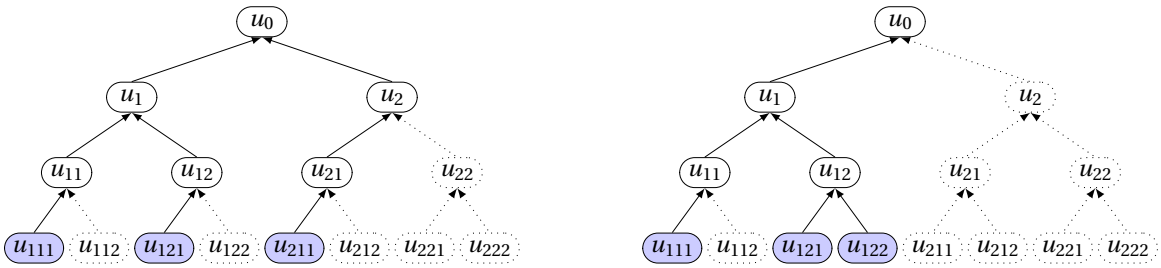
on a cubic domain with Dirichlet boundary conditions. We perform the computations in double-precision arithmetic. In case of BLR representation, it may be shown that the rank r of the approximated blocks resulting from the partition is asymptotically equal to $r = \Theta(1)$ [5].

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

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

where ω is the angular frequency, $v(x)$ is the seismic velocity field, and $u(x, \omega)$ is the time-harmonic wavefield solution to the forcing term $s(x, \omega)$. The matrix A is built for an infinite medium. This implies that the input grid is augmented with PML absorbing layers. Frequency is fixed and equal to 4 Hz. The grid interval h is computed such that it corresponds to 4 grid points per wavelength. Computations are done in single-precision arithmetic. In case of BLR representation, it may be shown that the rank r of the approximated blocks resulting from the partition is asymptotically equal to $r = \Theta(m^{1/2})$.

In our experiments, we use the multifrontal solver MUMPS, in which the BLR format has been integrated [6]. We compute the low-rank approximations to the off-diagonal blocks by computing their QR factorization with column pivoting and truncating it after a given threshold ε has been reached (i.e., we stop the factorization after the diagonal element r_{kk} of R falls below ε). We refer to ε as the low-rank threshold.



(a) MAXOPS (b) MINOPS

Figure 2.2: Example of construction of a RHS with 3 nonzeros for strategies MAXOPS and MINOPS. The 3 nonzeros are placed in the 3 filled nodes.

2.3 Complexity analysis

In this section, we provide our complexity analysis of the forward substitution phase when RHS sparsity is exploited. Since the RHS structure can be arbitrary, one must rely on a particular structure in order to compute complexity models. We first present in Section 2.3.1 some realistic models of sparse RHS structures, then provide in Sections 2.3.2 to 2.3.4 our complexity analysis for these increasingly realistic models.

2.3.1 Models for sparse RHS

Let us first model one sparse RHS with only one nonzero. As studied earlier, the associated computations in the pruned tree will be represented as a branch of the separator tree, starting from the active node, or equivalently the node containing the nonzero variable, up to the root. We have chosen the position of this nonzero to be located on the leaf that belongs to the critical path of the separator tree. This corresponds to the worst case scenario. As a consequence, any RHS structure with a single nonzero will have a complexity bounded by this worst case scenario. This corresponding theoretical complexity is discussed in Section 2.3.2.

As a second more general model we consider RHS structures with multiple nonzeros per column. We note nnz the number of nonzeros and consider that it can either be constant, or growing with N. To do so, we make the following assumptions on the structure of such RHS:

- only one nonzero per active node (this is indeed enough because multiple nonzeros per node would not change the operation count);
- the active node is a leaf or is at depth $\Theta(log(N))$; in other terms we do not assume any specific property that would imply that the nonzeros are associated to the top-level separators;
- the dependence over N of nnz may not be superior to $\Theta(N)$.

Based on these assumptions and in order to cover a wide range of cases one may model two types of right-hand sides:

- MAXOPS: it corresponds to a RHS structure that maximizes the number of operations during the solve phase. For that, we identify the first layer, called the *target layer* of the tree containing at least nnz nodes in the tree. We pick nnz nodes in this layer and pick a leaf in the nnz associated subtrees that we define as active, that is, on which we place a nonzero, as shown in Figure 2.2a. Geometrically, this would translate as a RHS whose set of nonzeros is almost uniformly spread in a large part of the domain.

- MINOPS: it corresponds to a RHS structure that would be gathered inside a smaller subtree, thus limiting the number of operations. We still pick the nonzeros in the leaves, but choose the smallest subtree containing at least nnz leaves, see Figure 2.2b. Geometrically, this would translate in nonzeros that are all localized in a small part of the domain.

We will see later that in the context of typical real-life applications, RHS are sometimes assimilated to *sources* geometrically localized in the domain. This is why we thought the MINOPS strategy could provide a reasonable model.

2.3.2 Ideal setting: one RHS with one nonzero

In this ideal setting, the forward substitution only requires the traversal of a single branch in the separator tree. Thus, exploiting the RHS sparsity amounts to dropping the term corresponding to the number of nodes at level k in the complexity formulas (2.2) and (2.3):

$$\mathcal{C}_{2D}^{ES}(N,\alpha) = \sum_{k=0}^{K_{2D}} \Theta\left(\left(N/2^k\right)^\alpha\right) = \Theta(N^\alpha), \tag{2.5}$$

$$\mathcal{C}_{3D}^{ES}(N,\alpha) = \sum_{k=0}^{K_{3D}} \Theta\left(\left(N^2/4^k\right)^\alpha\right) = \Theta(N^{2\alpha}), \tag{2.6}$$

We now measure the asymptotic gain obtained by exploiting the RHS sparsity, defined as

$$\mathcal{G}^{ES}(N,\alpha) = \frac{\mathcal{C}(N,\alpha)}{\mathcal{C}^{ES}(N,\alpha)}. \tag{2.7}$$

For the FR unstructured format, we obtain $\mathcal{G}_{2D}^{ES}(N) = \Theta(N^2 \log N)/\Theta(N^2) = \Theta(\log N)$ and $\mathcal{G}_{3D}^{ES}(N) = \Theta(N^4)/\Theta(N^4) = \Theta(1)$. Without the use of low-rank approximations, the gain due to the exploitation of the RHS sparsity is thus constant or nearly constant (see also the last column of Table II, for 9-point/27-point stencils in 2D/3D, respectively).

Interestingly, this changes when considering low-rank formats. For example, with the BLR format and assuming $r = \Theta(1)$ (i.e., $\alpha = 3/2$), we obtain $\mathcal{G}_{2D}^{ES}(N) = \Theta(N^2)/\Theta(N^{3/2}) = \Theta(N^{1/2})$. The asymptotic gain $\mathcal{G}_{2D}^{ES}$ is thus rapidly increasing with the number of unknowns. Similar results hold for the other formats and for other values of r; they are reported in Table III. The asymptotic gain can be as high as $\Theta(N)$ for hierarchical formats.

In Figure 2.3, we validate these theoretical results with numerical experiments performed with 2D and 3D Poisson and Helmholtz problems, and use the same fitting method as in [5]. We compare the asymptotic complexity fit of the forward substitution flops depending on whether

Table III: Asymptotic gain due to the exploitation of the RHS sparsity obtained for the forward substitution phase of a sparse system of $N \times N$ (2D case) or $N \times N \times N$ (3D case) unknowns, depending on which of the FR, BLR, MBLR, and hierarchical formats is used and depending on the rank bound r.

	$r = \Theta(1)$ (Poisson)		$r = \Theta(m^{1/2})$ (Helmholtz)	
	$\mathcal{G}_{2D}^{ES}(N)$	$\mathcal{G}_{3D}^{ES}(N)$	$\mathcal{G}_{2D}^{ES}(N)$	$\mathcal{G}_{3D}^{ES}(N)$
FR	$\Theta(\log N)$	$\Theta(1)$	$\Theta(\log N)$	$\Theta(1)$
BLR	$\Theta(N^{1/2})$	$\Theta(\log N)$	$\Theta(N^{1/4})$	$\Theta(1)$
MBLR	$\Theta(N^{\ell/(\ell+1)})$	$\Theta(N^{(\ell-1)/(\ell+1)})$	$\Theta(N^{\ell/(2\ell+2)})$	$\Theta(1)$
Hierar.	$\Theta(N)$	$\Theta(N)$	$\Theta(N^{1/2})$	$\Theta(\log N)$

we use a FR unstructured or BLR matrix format, and on whether the RHS sparsity is exploited or not. For the results with BLR, we use a low-rank threshold $\varepsilon = 10^{-6}$ for Poisson and $\varepsilon = 10^{-3}$ for Helmholtz. Finally, we mention that we have chosen the branch (or the RHS) to be the critical path of the solve phase.

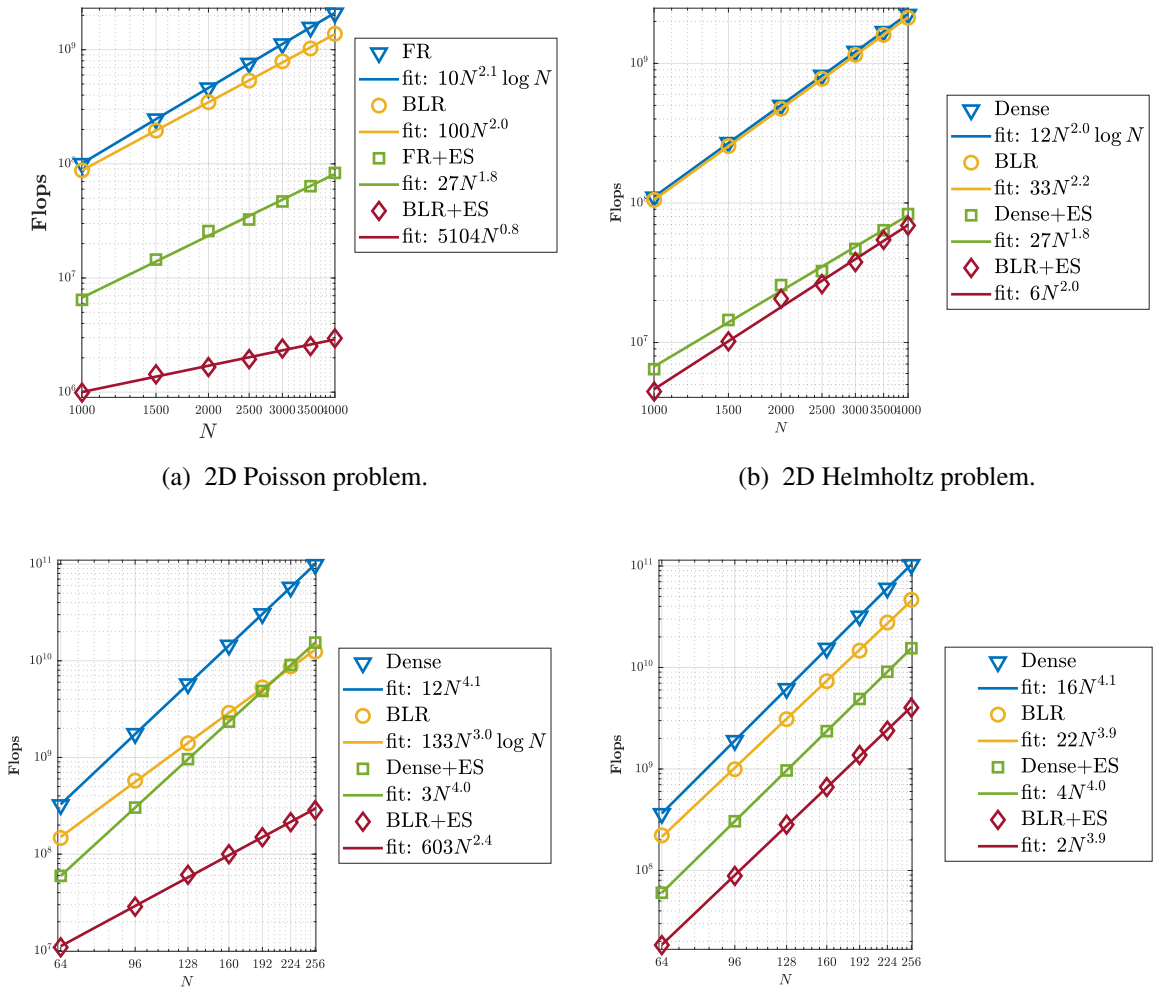
These experimental results are in relative good agreement with the theoretical gains reported in Table III. In fact, the experimental values of the asymptotic gain $\mathcal{G}_{ES}(N)$ obtained with these problems and the fitting method used are always better and, sometimes, even much better than their theoretical bounds. Although it would be interesting to be able to use larger problems, possibly avoiding the smallest problems, in order to see the impact on the fitting, the experimental results go in the expected direction.

For the 2D Poisson problem, we obtain a $\Theta(N^{0.3} \log N)$ asymptotic gain in the case of the FR format, whereas in the BLR case we obtain a much larger gain of $\Theta(N^{1.2})$, thereby confirming our theoretical finding that RHS sparsity benefits much more from the low-rank solver. In the 3D case, almost no asymptotic gain is achieved by the FR format, as predicted by theory; the BLR gain is however much larger than its theoretical prediction $\Theta(\log N)$, reaching $\Theta(N^{0.6} \log N)$.

For the Helmholtz problem, we obtain only a constant $\mathcal{G}_{ES}$ gain in the 3D case, as expected, whereas in the 2D case, both the FR and BLR formats benefit from a small asymptotic gain. Interestingly, when the RHS sparsity is not exploited, the BLR format does not succeed in significantly reducing the global number of flops (triangle and circle curves are indistinguishable in Figure 2.3b), which is dominated by the processing of nodes at the bottom of the separator tree, whereas a significant BLR reduction is achieved (diamond curve is well below square one) when RHS sparsity is exploited.

2.3.3 Generalization to one RHS with multiple nonzeros

In real-life applications, the RHS typically have more than one nonzero. Let us now consider the case of RHS with nnz nonzeros. If $nnz = \Theta(1)$, then the theoretical results of the previous section obviously still hold, since exploiting the RHS sparsity then amounts to traverse $\Theta(1)$ branches of the separator tree. nnz will now be considered as a parameter depending on N.



(a) 2D Poisson problem.

(b) 2D Helmholtz problem.

(c) 3D Poisson problem.

(d) 3D Helmholtz problem.

Figure 2.3: Experimental fit of the forward substitution complexity for the FR and BLR matrix formats, depending on whether RHS sparsity is exploited or not.

Table IV: Complexity of the solution phase with a non-constant number of nonzeros per RHS of a sparse system of $N \times N$ (2D case) or $N \times N \times N$ (3D case) unknowns, assuming a storage complexity $\Theta(m^\alpha)$, when using the MAXOPS model.

$\mathcal{C}_{2D}^{ES}(N)$		$\mathcal{C}_{3D}^{ES}(N)$	
$\alpha > 2$	$\Theta(N^\alpha)$	$\alpha > 3/2$	$\Theta(N^{2\alpha})$
$\alpha = 2$	$\Theta(N^2 \log nnz)$	$\alpha = 3/2$	$\Theta(N^3 \log nnz)$
$\alpha < 2$	$\Theta(N^\alpha \times nnz^{(2-\alpha)})$	$\alpha < 3/2$	$\Theta(N^{2\alpha} \times nnz^{(3-2\alpha)})$

In fact, in most applications, RHS do have a $\Theta(1)$ number of nonzeros. Nonetheless, it is interesting to evaluate the behavior of $\mathcal{C}^{ES}$ when the number of nonzeros is not constant as it gives insights on the limits of the application of this complexity study.

Using the MAXOPS model

The first modelisation represents the worst case scenario where the RHS structure induces a rapid increase in terms of operations since the pruned tree contains the upper and largest nodes of the separator tree. The application of tree pruning may thus not be enough to preserve the results of Section 2.3.2.

We express here the complexity of the forward substitution for the 2D case with the sparse structure built with the MAXOPS strategy as a function of the nnz nonzeros. Let $K_1 = \log_4(nnz)$ be the depth of the target layer, see Section 2.3.1. In the first K_1 levels of the tree, the computation of $\mathcal{C}_{2D}^{ES}$ remains unchanged compared to $\mathcal{C}_{2D}$, and below layer K_1 we compute on nnz branches so that $\mathcal{C}_{2D}^{ES}$ becomes

$$\mathcal{C}_{2D}^{ES}(N, \alpha, nnz) = \sum_{k=0}^{K_1} \Theta\left(4^k \left(N/2^k\right)^\alpha\right) + \sum_{k=K_1+1}^{K_{2D}} nnz \times \Theta\left(\left(N/2^k\right)^\alpha\right). \tag{2.8}$$

The first term can be expressed as $N^\alpha \times \sum_{k=0}^{K_1} \Theta\left(2^{(2-\alpha)k}\right)$ so that it depends on α:

1. Case $\alpha > 2$: $A = N^\alpha$
2. Case $\alpha = 2$: $A = N^2 \log_4 nnz$
3. Case $\alpha < 2$: $A = N^\alpha \times \Theta(2^{(2-\alpha)K_1}) = \Theta\left(N^\alpha nnz^{1-\alpha/2}\right)$, since $K_1 = \log_4 nnz$.

The second term may be expressed as $N^\alpha \times nnz \times \sum_{k=K_1+1}^{K_{2D}} \Theta\left(\left(1/2^\alpha\right)^k\right)$. Using the fact that $\sum_{k=K_1}^{K_{2D}} q^k = \frac{q^{K_1} - q^{K_{2D}}}{1-q} = \Theta(q^{K_1})$, we obtain:

$$N^\alpha \times nnz \times \Theta\left(2^{-\alpha K_1}\right) = \Theta\left(N^\alpha nnz^{1-\alpha/2}\right)$$

We can conclude that the order of $\mathcal{C}_{2D}^{ES}$ corresponds to the order of the first term. The results are gathered in Table IV.

The consequence of the results of Table IV is that, using the MAXOPS model, the complexity directly depends on nnz and we might thus lose part of the asymptotic gain observed in the previous section. For example, we may choose a model of nonzero such as $nnz = \Theta(N^\gamma)$ with $0 \leq \gamma \leq 1$. Then, we see that for $\alpha = 2$ (full-rank case), $\mathcal{C}_{2D}^{ES}(N) = \Theta(N^2 \log N) = \mathcal{C}_{2D}(N)$ so that the gain is lost. However, the low-rank approximation with $\alpha = 3/2$ limits the loss in the sense that as long as $\gamma < 1$, we still have an asymptotic gain. It also proves that, in this scenario corresponding to RHS nonzeros spread in the domain, having a non-constant nnz induces a direct loss on $\mathcal{C}^{ES}$.

Using the MINOPS model

In typical applications from geosciences, the RHS nonzeros represent physical locations surrounding a given point referred to as *source*, in other words, these nonzeros are geometrically clustered. First, if the number of nonzeros does not depend on N ($nnz = \Theta(1)$, as in some applications), the asymptotic complexity results are identical to the ones with a single nonzero from Section 2.3.2. Then, with such a configuration of nonzero structure, if nnz is not constant, our objective is to understand under which condition the theoretical results from Section 2.3.2 may still hold.

This locality of RHS matches the construction of the nnz nonzeros using the MINOPS strategy. The forward substitution consists in processing the entire subtree and then traversing the branch from the root of the subtree to the root of the global separator tree. We mention that considering the entire subtree whereas there may be slightly less nonzeros than the number of leaves in that subtree (see the Example of Figure 2.2b) does not change the asymptotic complexity.

We first focus on the 2D case. Let K_1 be the number of levels of the subtree, and $K_2 = K_{2D} - K_1$ be the number of levels in the remaining branch. We can compute K_1 using

$$nnz = \sum_{k=0}^{K_1} 4^k 2^{K_1 - k} \Rightarrow K_1 = \Theta(\log_4 nnz).$$

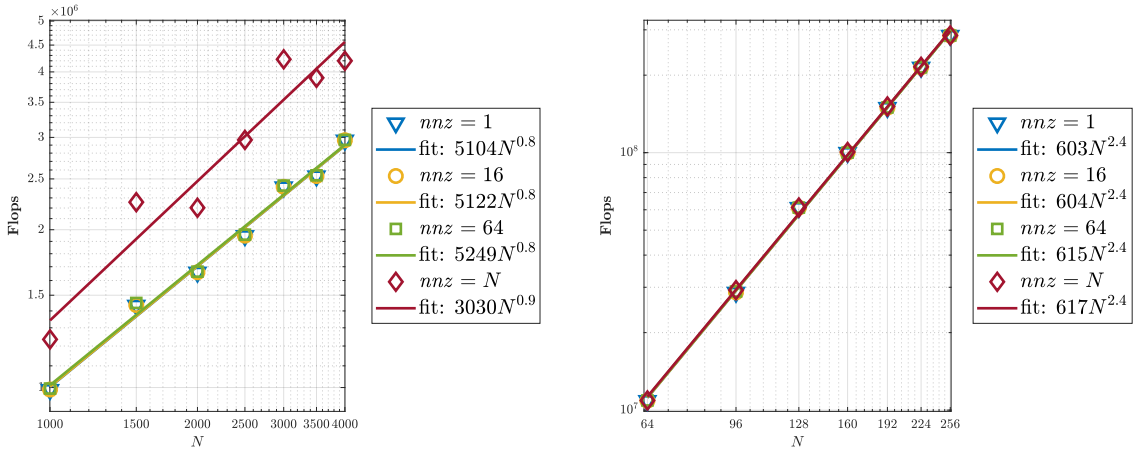
The root of the subtree is thus of size $\Theta(N/2^{K_2}) = \Theta(2^{K_1}) = \Theta(nnz^{1/2})$. We can therefore compute the complexity of the forward substitution with sparse RHS with nnz nonzeros as

$$\mathcal{C}_{2D}^{ES}(N, \alpha, nnz) = \sum_{k=0}^{K_2} \Theta\left(\left(N/2^k\right)^\alpha\right) + \sum_{k=0}^{K_1} \Theta\left(4^k \left(nnz^{1/2}/2^k\right)^\alpha\right). \tag{2.9}$$

The first term is equal to $\mathcal{C}_{2D}^{ES}(N, \alpha) = \Theta(N^\alpha)$, while the second term is of order $\Theta(nnz)$ if $\alpha < 2$. Thus, as long as $nnz \leq \Theta(N^\alpha)$, having multiple nonzeros does not increase the overall asymptotic complexity of the forward substitution; nnz can thus potentially be nonconstant. We can prove a similar result in the 3D case:

$$\mathcal{C}_{3D}^{ES}(N, \alpha, nnz) = \mathcal{C}_{3D}^{ES}(N, \alpha) + \sum_{k=0}^{\log_8 nnz} \Theta\left(8^k \left(nnz^{2/3}/4^k\right)^\alpha\right), \tag{2.10}$$

from which we conclude that, for $\alpha < 3/2$, $\mathcal{C}_{3D}^{ES}(N, \alpha)$ remains of the same asymptotic order as long as $nnz \leq \Theta(N^{2\alpha})$.



(a) 2D Poisson problem.

(b) 3D Poisson problem.

Figure 2.4: Experimental fit of the forward substitution complexity using the BLR matrix format and exploiting the RHS sparsity, depending on the number of nonzeros of the RHS.

We illustrate this fact in Figure 2.4 for the 2D and 3D Poisson problems. We mention that the RHS were built upon the MINOPS(m)odel with the additionnal characteristic that one of the nonzeros is located on the critical path. Regardless of the value of nnz, the experimental complexity of the BLR forward substitution remains very close to $\Theta(N^{0.8})$ in 2D and $\Theta(N^{2.4})$ in 3D, although some instability is observed for $nnz = N$ in 2D in our experimental environment.

2.3.4 Generalization to multiple RHS (with multiple nonzeros)

As mentioned in the introduction, we are interested in the case where we have a multiple number of RHS n_{rhs}.

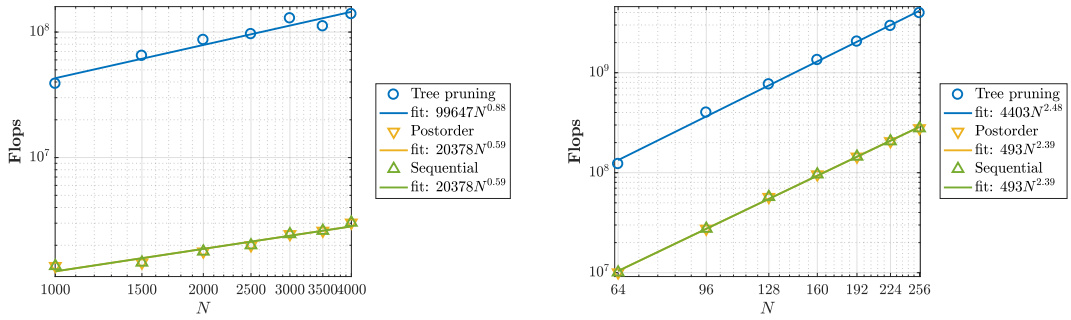
If the RHS are processed sequentially then the complexity of the forward substitution is straightforward to compute and equal to

$$\mathcal{C}^{ES}(N, \alpha, nnz, n_{rhs}) = n_{rhs} \times \mathcal{C}^{ES}(N, \alpha, nnz), \tag{2.11}$$

which is the best result we can expect since it is linear with respect to n_{rhs}.

However, as explained in Section 2.2.2, this approach does not allow us to exploit BLAS-3 operations. We therefore wish to process the RHS by groups, of a size $b_{rhs} > 1$.

In Figure 2.5, we compare the impact of tree pruning, of the postorder and of the sequential (RHS processed one-by-one) strategies on 2D and 3D Poisson problems with a RHS structure following the MINOPS model, where we chose a random subtree for each RHS. We mention that RHS have a different nonzero structure, that is, for each RHS is associated a different subtree containing the nnz chosen nonzeros, see Section 2.3.1. We plot the cost of the BLR forward substitution normalized by the number of RHS, which is here set to $n_{rhs} = b_{rhs} = 256$.



(a) 2D Poisson problem. (b) 3D Poisson problem.

Figure 2.5: Experimental fit of the forward substitution complexity using the BLR matrix format and exploiting the RHS sparsity, depending on the strategy to process multiple RHS (here, $n_{rhs} = b_{rhs} = 256$ and $nnz = 64$).

In Figure 2.5, the cost is divided by almost 50 (in 2D) and 15 (in 3D) when using the improved strategy, and we even observe that the application of the Postorder strategy on the RHS built with the MINOPS model gives the optimal solution, that is the Sequential and Postorder curves of Figure 2.5 overlap. This confirms the effectiveness of the Postorder permutation on such models (*i.e.*, MINOPS).

Thus, in our setting and with our MINOPS model to define the nonzero structure of each RHS, the number of operations with the Postorder permutation is optimal even when using blocks of RHS columns and the asymptotic complexity gain $\mathcal{G}^{ES}$ is preserved. However, we may argue that the different models from Section 2.3.1 do not suffice to describe real-life applications that do not always reach the optimality of the Postorder showed in Figure 2.5.

2.4 Experimental validation on real-life applications

In the following, we observe and discuss the behavior the gain on real-life applications considering both the number of operations and the time of the forward substitution. To do so, we took a restricted set of matrices from Table III of Section 1.6, namely matrices 5Hz, 10Hz from the seismic application and H3, H17 and S3, S21 from the electromagnetic application. The results are depicted in Tables V and VI. Each couple of matrices derives from the same model problem so that their matrices differ only by their size (e.g, H17 is larger than H3). We are thus able to assess the evolution of the gain as the problem size increases.

We used for these results a geometric nested dissection ordering, and a recent version of the MUMPS solver that includes a Block-Low Rank feature for both the factorization and the solve stages. We used a 2D block cyclic factorization on the root node (from ScaLAPACK), so that we do not include flops and timings for that node for the purpose of the comparison. Furthermore, these results benefit from the work of Chapter 4 (e.g., concerning the organization of the RHS by blocks and within each block).

Table V: Number of operations (OPS$\times 10^{12}$) and associated gains of the forward substitution phase in FR and BLR from SEISCOPE and EMGS. OPS do not include the root node where ScaLAPACK is used.

OPS	FR	BLR 5Hz	$\mathcal{G}^{BLR}$	FR	BLR 10Hz	$\mathcal{G}^{BLR}$
ES Off	16.9	12.6	1.3	201.6	114.9	1.8
ES On	3.3	1.2	2.8	39.4	9.0	4.3
$\mathcal{G}^{ES}$	5.1	10.5		5.1	11.7	
		H3			H17	
ES Off	69.7	34.5	2.0	777.0	404.9	1.9
ES On	10.7	4.2	2.6	101.9	10.2	10.0
$\mathcal{G}^{ES}$	6.5	8.2		7.6	40.0	
		S3			S21	
ES Off	131.4	37.7	3.5	1521.3	432.3	3.5
ES On	21.2	8.3	2.5	208.9	35.0	6.0
$\mathcal{G}^{ES}$	6.2	4.5		7.3	12.4	

Table VI: Time (s) and associated gains of the forward substitution phase in FR and BLR for matrices from the SEISCOPE and EMGS sets. 90 MPI$\times$1 OMP on `EOS`. Times do not include the root node where ScaLAPACK is used.

Time	FR	BLR 5Hz	$\mathcal{G}^{BLR}$	FR	BLR 10Hz	$\mathcal{G}^{BLR}$
ES Off	50	42	1.2	458	250	1.8
ES On	26	18	1.4	201	92	2.2
$\mathcal{G}^{ES}$	1.9	2.3		2.3	2.7	
		H3			H17	
ES Off	377	273	1.4	5449	3097	1.8
ES On	166	119	1.4	1339	630	2.1
$\mathcal{G}^{ES}$	2.3	2.3		4.1	4.9	
		S3			S21	
ES Off	641	443	1.4	6719	3104	2.2
ES On	404	255	1.6	3748	1386	2.7
$\mathcal{G}^{ES}$	1.6	1.7		1.8	2.2	

We mention that, for each matrix, the comparison of the values from column $\mathcal{G}^{BLR}$ gives insights on the performance of the low-rank compression when exploiting sparsity or not. Similarly, values of row $\mathcal{G}^{ES}$ assesses the performance of the exploitation of sparsity when using low-rank approximations or not.

We recall that the conclusions from Section 2.3 were first that the acceleration of the forward substitution increases with N when exploiting sparsity (at least in regular 3D problems) and second that this acceleration was more significant with the use of BLR approximations. Indeed, we are here interested in the evolution of $\mathcal{G}^{ES}$ and $\mathcal{G}^{BLR}$ as the problem size increases, that is their evolution on each set of couple matrices. Considering $\mathcal{G}^{ES}$ in FR, the increase exists but seems moderate (going from 6.5 to 7.6 for matrices H3 and H17 in Table V). However, it is noticeably improved in the BLR case (going from 8.2 to 40.0) and the result remains true for any other set of matrices. As a consequence, $\mathcal{G}^{ES}$ increases more rapidly with BLR approximations. This is coherent with the theoretical results of Section 2.3.

Concerning the timings, we observe a similar tendency in Table VI, although less pronounced: the gain due to exploiting sparsity increases more a BLR setting than a FR setting.

2.5 Extension to tree parallelism

We now consider the solve phase in the general case of dense RHS on which we propose to interpret the previous asymptotic study in this context.

As indicated in Chapter 1, tree parallelism arises from the fact that two nodes from different subtrees can be computed in parallel. A qualitative measure of tree parallelism is the computation of the so-called *theoretical speed up* associated with the solution of sparse systems. The

Table VII: Theoretical speed up obtained for the factorization phase of a sparse system of $N \times N$ (2D case) or $N \times N \times N$ (3D case) unknowns, depending on which of the FR, BLR, MBLR, and hierarchical formats is used and depending on the rank bound r.

	$r = \Theta(1)$		$r = \Theta(m^{1/2})$	
	$\mathcal{S}_{2D}(N)$	$\mathcal{S}_{3D}(N)$	$\mathcal{S}_{2D}(N)$	$\mathcal{S}_{3D}(N)$
FR	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
BLR	$\Theta(\log N)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Hierar.	$\Theta(N)$	$\Theta(N)$	$\Theta(N^{1/2})$	$\Theta(\log N)$

former metric, noted $\mathcal{S}$ supposes an infinite number of processors and it is defined as the ratio between the workload on the whole separator tree and on its critical path, that was already introduced in Section 1.4 and that was noted $\frac{\Delta}{\Delta_c}$. On an infinite number of processors, the solve time corresponds to the time to compute the critical path Δ_c. $\mathcal{S}$ is thus the maximal quantity of tree parallelism present in the tree. As a consequence, the larger (resp. thinner) is the tree, the better (resp. worse) the tree parallelism.

Yet, the speed-up is similar to the definition of $\mathcal{G}^{ES}$ when defining $\mathcal{C}^{ES}$ as the complexity to process one branch of the separator tree (assuming a balanced tree, which is the case on regular problems). As a consequence, the results previously obtained for the case of a single nonzero and a single RHS apply to the metric $\mathcal{S}$ but are now to be interpreted differently. Indeed, we may now conclude from Table III that tree parallelism increases asymptotically with N, and that this is even more true when low-rank approximation are used.

The comparison with the factorization stresses one other main difference between the two algorithms. Indeed, a close comparison of Tables III and VII shows that the solve algorithm exhibits more tree parallelism than the factorization. This result gives a valuable insight to drive the design of algorithms to efficiently take into account the inherent properties (tree parallelism) of the solve phase. We will discuss again the different intrinsic properties of the factorization and the solve phase to drive parallel algorithms dedicated to the solve phase in Chapter 4.

2.6 Conclusion

In this chapter we have investigated the asymptotic complexity of the solution phase of sparse linear systems $AX = B$ with multiple right-hand sides, focusing on the forward substitution $LY = B$.

In the case of traditional full-rank solvers, exploiting the sparsity of B only leads to a constant or nearly constant gain for 3D problems, thus leaving the asymptotic complexity of the forward substitution unchanged. However, this is no longer true for solvers based on low-rank formats, such as BLR or hierarchical formats, for which a significant asymptotic gain is obtained. Specifically, our theoretical computations prove that exploiting the RHS sparsity improves the complexity of the BLR forward substitution by a factor of order $\Theta(n^{1/4})$ in 2D (significantly larger than the $\Theta(\log n)$ obtained using traditional solvers) and $\Theta(\log n)$ in 3D.

The factor of improvement is even larger for multilevel or hierarchical formats. Our numerical experiments support these bounds and show that in practice even higher gains are obtained.

Due to this result, the forward substitution becomes dominated by the backward substitution and thus of negligible cost for large enough problems. Therefore, exploiting the RHS sparsity divides the cost of the solution phase by a factor of two. Importantly, in some applications, only part of the solution is needed; then, X is also a sparse matrix and the analysis performed in this chapter also applies to the backward substitution. In that case, the asymptotic complexity of the overall solution phase could itself be improved by exploiting the sparsity in both B and X.

With the RHS model MINOPS described in Section 2.3.1, we observed that in the presence of multiple right-hand sides, the number of operations is optimal, that is, equal to those one would obtain when treating the RHS columns sequentially, one-by-one. This is indeed due to the fact that RHS nonzeros were localized within a subtree. However, geometrical locality could lead to some RHS that are not fully part of low-level subtrees (e.g. if they are cut by high-level separators) and we will see why in such cases, the postorder is no longer optimal. One may also have more general RHS structures that may question the optimality of the Postorder permutation obtained with the model used in Section 2.3.4, and the validity of Equation (2.4). In the next chapter, we will propose techniques to limit this number of operations down to an arbitrarily threshold above the optimum, while keeping large-enough blocks. This will allow the theoretical complexity results of this chapter can effectively hold in the case of general multiple RHS.

Chapter 3

On the exploitation of right-hand side sparsity

We still consider the direct solution of sparse systems of linear equations

$$AX = B, \tag{3.1}$$

where A is an $n \times n$ sparse matrix with a symmetric structure and B is an $n \times n_{rhs}$ matrix of right-hand sides (RHS). We consider the decomposition $A = LU$ or $A = LDL^T$ with a sparse direct method [28], and we focus on the efficient solution of the forward system

$$LY = B, \tag{3.2}$$

where the unknown Y and the right-hand side B are $n \times n_{rhs}$ matrices. We will see in this study that the ideas developed for Equation (3.2) are indeed more general and can be applied in a broader context. In particular, they can be applied to the backward substitution phase, in situations where the system $UX = Y$ must be solved for a subset of the entries in X [10, 61, 69, 70]. The work presented in this chapter was motivated by electromagnetism, geophysics or imaging applications that can lead to systems with sparse multiple right-hand sides for which the solution phase is significantly more costly than the factorization phase [4, 60].

This chapter assumes familiarity with the notions presented in Chapter 1. It relies a lot on the separator tree (noted T) and on the pruned tree $T_p(B)$, which allows to exclude RHS rows for which computation can be fully avoided (*vertical* sparsity). We refer the reader to Section 1.3.2 for an introduction of how RHS sparsity can be exploited and for the other concepts and notation used in this chapter. We recall that the efficiency of *horizontal sparsity* strongly depends on the way the columns of B are ordered, as this impacts the sizes of the intervals Z_u (defined in Equation (1.10)) of RHS columns on which computations are performed for a given node $u \in T$. There are δ_u operations per RHS column processed at node u of the tree, and our minimization problem (already presented in (1.12)) is:

$$\begin{aligned}&\text{Find a permutation } \sigma \text{ of } \{1, \ldots, n_{rhs}\} \text{ that minimizes } \Delta(B, \sigma) = \textstyle\sum_{u \in T_p(B)} \delta_u \times \theta(\sigma(Z_u)),\\ &\text{where } \sigma(Z_u) = \{\sigma(i) \mid i \in Z_u\}\,, \text{ and}\\ &\qquad \theta(\sigma(Z_u)) \text{ is the length of the permuted interval } [\![\min(\sigma(Z_u)), \max(\sigma(Z_u))]\!].\end{aligned} \tag{3.3}$$

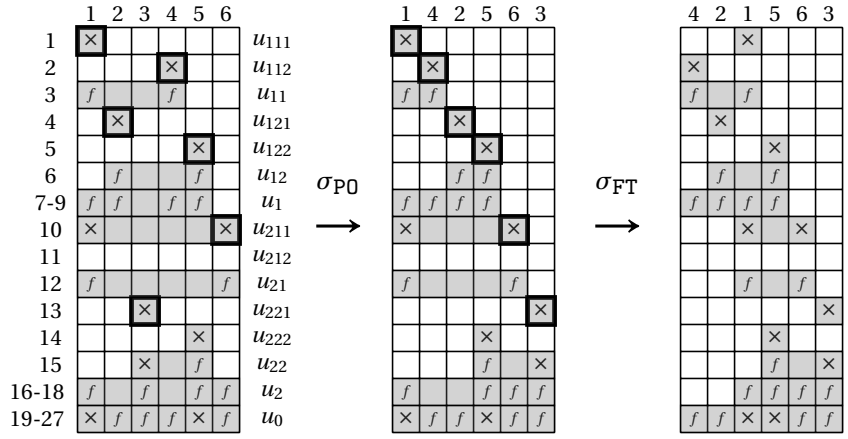


Figure 3.1: An RHS matrix B with multiple nonzeros per column (left), its postorder permutation (σ_{FT}, middle), and a more efficient permutation (σ_{FT}, right).

Restarting from Example 1.5 of Section 1.3.2, we present in Figure 3.1(left) the structure of a RHS composed of 6 columns and leading to an amount of computation represented by the gray cells (see also Figure 1.10 and the details on the operation count in the corresponding text). With the postorder permutation σ_{PO} from Definition 1.5, the number of gray cells, and the number of operations, is reduced. One of the contributions of this chapter is to introduce a new permutation, represented on the right of Figure 3.1(right), that further reduces the number of operations with respect to the postorder. The algorithm will first be introduced for a nested dissection ordering and for a regular mesh using geometric intuitions, then generalized to arbitrary elimination trees.

Computation can then be further reduced by dividing the RHS into blocks. However, instead of enforcing a constant number of columns per block, our objective is to minimize the number of blocks created. If $\Delta_{min}(B)$ represents the number of operations to solve (3.2) when processing the RHS columns one by one, we show on real applications that our blocking algorithm can approach $\Delta_{min}(B)$ within a tolerance of 1% while creating a small number of blocks. Please note that RHS sparsity limits the amount of tree parallelism because only a few branches are traversed in the elimination tree. Therefore, whenever possible, our heuristics also aim at choosing the approach that maximizes tree parallelism.

This Chapter is organized as follows. In Section 3.1, we introduce a new permutation to reduce the size of such intervals and thus limit the number of operations, first using geometrical considerations for a regular nested dissection ordering, then with a pure algebraic approach that can then be applied in a general case and for arbitrary right-hand sides. We call it the *Flat Tree* algorithm (hence the name σ_{FT} in Figure 3.1) because of the analogy with the ordering that one would obtain when "flattening" the tree. In Section 3.2, an original blocking algorithm is then introduced to further improve the flat tree ordering. It aims at defining a limited number of blocks of right-hand sides to minimize the number of operations while preserving parallelism. Section 3.3 gives experimental results on a set of systems coming from two geophysics applications relying on Helmholtz or Maxwell equations. Section 3.4 discusses adaptations of the

nested dissection algorithm to further decrease computation and Section 3.5 shows why this work has a broader scope than solving Equation (3.2) and presents possible applications.

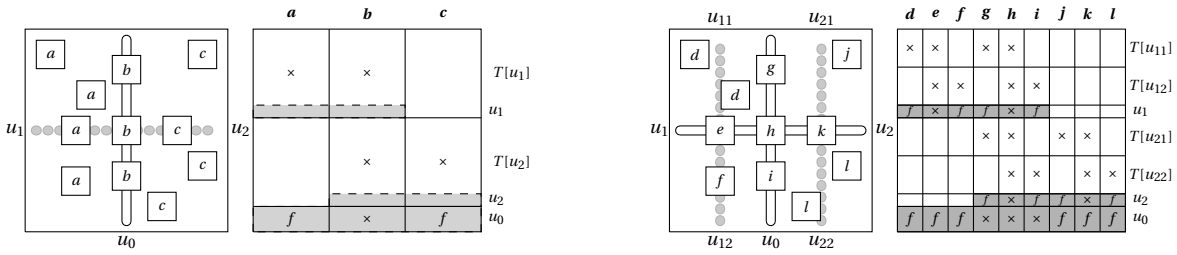
3.1 The flat tree permutation

With the aim of satisfying node optimality (see Definition 1.4), we present another algorithm to compute the permutation σ by first illustrating its geometric properties and then extending it to rely only on algebraic properties.

3.1.1 Geometrical intuition

As said previously, the variables of a separator u are the ones of the corresponding node u in the tree T. We use the same approach to represent a domain: for $u \in T$, the domain associated with u is defined by the subtree rooted at u and is noted $T[u]$. The set of variables in $T[u]$ corresponds to a subdomain created during the nested dissection algorithm. As an example, the initial 2D domain in Figure 3.2a (left) is $T[u_0]$ and its subdomains created by dividing it with u_0 are $T[u_1]$ and $T[u_2]$. In the following, $T[u]$ will equally refer to a subdomain or a subtree.

We do the strong assumption here that the nonzeros in an RHS column correspond to geometrically contiguous nodes in a regular domain on which a perfect nested dissection has been performed. For instance, all separators are in the same direction at each level of the tree.



(a) Flat tree step 1.

(b) Flat tree step 2.

Figure 3.2: Flat tree geometrical illustration. In (a) and (b), the figure on the left represents different types of RHS, and the one on the right the permuted RHS matrix. $\times$ or f in a rectangle indicate the presence of nonzeros in the corresponding submatrix, parts of the matrix filled in gray are fully dense and blank parts only contain zeros.

The *flat tree* algorithm relies on the evaluation of the position of each RHS column compared to separators. The name flat tree comes from the fact that, given a parent node with two child subtrees, the algorithm orders first RHS columns included in the left subtree, then RHS columns associated to the parent (because they intersect both subtrees), and finally, as in an inorder, RHS columns included in the right subtree.

Figure 3.2a shows the first step of the algorithm: it starts with the root separator u_0 which divides $T = T[u_0]$ into $T[u_1]$ and $T[u_2]$. The initial RHS columns may be *identified* by three different types noted a, b and c according to their positions and nonzero structures. An RHS

column is of type a when its nonzero structure is included in $T[u_1]$, c when it is included in $T[u_2]$, and b when it is *divided* by u_0. First, we group the RHS according to their type (a, b, or c) with respect to u_0 which leads to the creation of submatrices/subsets of RHS columns noted $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$. Second, we make sure to place $\boldsymbol{b}$ between $\boldsymbol{a}$ and $\boldsymbol{c}$. We thus achieve operation reduction by guaranteeing node optimality at u_1 and u_2: since all RHS in $\boldsymbol{a}$ and $\boldsymbol{b}$ have at least one nonzero in $T[u_1]$, u_1 belongs to the pruned tree of all of them, hence the dense area filled in gray in the RHS structure. The same is true for $\boldsymbol{b}$ and $\boldsymbol{c}$ and u_2. By permuting B as $[\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}]$ ($[\boldsymbol{c}, \boldsymbol{b}, \boldsymbol{a}]$ would also be possible), $\boldsymbol{a}$ and $\boldsymbol{b}$, and $\boldsymbol{b}$ and $\boldsymbol{c}$, are contiguous. Thus, $\theta(Z_{u_1}) = \#Z_{u_1}$, $\theta(Z_{u_2}) = \#Z_{u_2}$ and we have u_1- and u_2-optimality.

The algorithm proceeds recursively on each submatrix created to obtain *local* node optimality. First, $\boldsymbol{d}, \boldsymbol{e}, \boldsymbol{f}$ (resp. $\boldsymbol{j}$, $\boldsymbol{k}$, $\boldsymbol{l}$) form subsets of the RHS of $\boldsymbol{a}$ (resp. $\boldsymbol{c}$) based on their position/type with respect to u_1 (resp. u_2), see Figure 3.2b. Second, thanks to the perfect nested dissection assumption, u_1 and u_2 can be combined to form a single separator that subdivides the RHS of $\boldsymbol{b}$ into three subsets $\boldsymbol{g}$, $\boldsymbol{h}$ and $\boldsymbol{i}$. During this second step, B is permuted as $[\boldsymbol{d}, \boldsymbol{e}, \boldsymbol{f}, \boldsymbol{g}, \boldsymbol{h}, \boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k}, \boldsymbol{l}]$. The algorithm stops when the tree is fully processed or the RHS sets contain a single RHS.

This draws the outline of the algorithm introduced with geometrical considerations. The permutation fully results from the position of each RHS with respect to separators. However, the algorithm relies on strong assumptions regarding the ordering algorithm and the RHS structure. Without them, it is difficult or impossible to discriminate RHS columns in many cases (for example, when they are separated by several separators).

In order to overcome these limitations and enlarge the application field, we now extend these geometrical considerations with a more general approach.

3.1.2 Algebraic approach

Let us consider the columns of B as an initially unordered *set* of RHS columns that we denote $R_B = \{B_1, B_2, \ldots, B_{n_{rhs}}\}$. $R \subset R_B$ is a subset of the columns of B and $r \in R$ is a generic element of R (one of the columns B_j). A permuted submatrix of B can be expressed as an ordered *sequence* of RHS columns. For two subsets of columns R and R', $[R, R']$ denotes a sequence of RHS columns in which the RHS from the subset R are ordered before those from R', without the order within R and R' to be necessarily defined. We found this framework of RHS sets and subsets better adapted to formalize our algebraic algorithm than matrix notation with complex index permutations.

We now characterize the geometrical position of a RHS using the notion of *pruned layer*: for a given depth d in the tree, and for a given RHS $r \in R_B$, we define the *pruned layer* $L_d(r)$ as the set of nodes at depth d in the pruned tree $T_p(r)$. In the example of Figure 3.2a, $L_1(r) = \{u_1\}$ for all $r \in \boldsymbol{a}$, $L_1(r) = \{u_2\}$ for all $r \in \boldsymbol{c}$, and $L_1(r) = \{u_1, u_2\}$ for all $r \in \boldsymbol{b}$. The notion of pruned layer formally identifies sets of RHS with common characteristics in the tree, without geometric information. This is formalized and generalized by Definition 3.1.

Definition 3.1. *Let $R \subset R_B$ be a set of RHS, and let U be a set of nodes at depth d of the tree T. We define $R[U] = \{r \in R \mid L_d(r) = U\}$ as the subset of RHS with pruned layer U.*

We have for example, see Figure 3.2: $R[\{u_1\}] = \boldsymbol{a}, R[\{u_2\}] = \boldsymbol{c}$ and $R[\{u_1, u_2\}] = \boldsymbol{b}$ at depth $d = 1$.

The algebraic recursive algorithm is depicted in Algorithm 3.1. Its arguments are R, a set of RHS and d, the current depth. Initially, $d = 0$ and $R = R_B = R[u_0]$. At each recursion step, the algorithm builds the distinct pruned layers $U_i = L_{d+1}(r)$ for the RHS r in R. Then, instead of looking for a permutation σ to minimize $\sum_{u \in T_p(R_B)} \delta_u \times \theta(\sigma(Z_u))$, it orders the $R[U_i]$ by considering the *restriction* of (1.12) to R and to nodes at depth $d + 1$ of $T_p(R)$. Furthermore, with the assumption that T is balanced, all nodes at a given level of $T_p(R)$ are of comparable size. δ_u may thus be assumed *constant* per level and needs not be taken into account. The algorithm is a greedy top-down algorithm, where at each step a local optimization problem is solved. This way, priority is given to the top of the tree, which is in general more critical because factor matrices are larger.

Algorithm 3.1 Flat Tree

procedure FLATTREE(R, d)
`1) Build the set of children` $C(R)$
`1.1) Identify the distinct pruned layers (pruned layer = set of nodes)`
$\mathcal{U} \leftarrow \emptyset$
for all $r \in R$ **do**
$\mathcal{U} \leftarrow \mathcal{U} \cup \{L_{d+1}(r)\}$
end for
`1.2)` $C(R) = \{R[U] \mid U \in \mathcal{U}\}$
`2) Order children` $C(R)$ as $[R[U_1], \ldots, R[U_{\#C(R)}]]$:
return [FLATTREE($R[U_1], d + 1$),. . .,FLATTREE($R[U_{\#C(R)}], d + 1$)]
end procedure

The recursive structure of the algorithm can be represented by a recursion tree T_{rec} defined as follows: each node R of T_{rec} represents a set of RHS, $C(R)$ denotes the set of children of R and the root is R_B. By construction of Algorithm 3.1, $C(R)$ is a partition of R, *i.e.*, $R = \dot{\bigcup}_{R' \in C(R)} R'$ (disjoint union). Note that all $r \in R$ such that $L_{d+1}(r) = \emptyset$ belong to $R[\emptyset]$, which is also included in $C(R)$. In this special case, $R[\emptyset]$ can be added at either extremity of the current sequence without introducing extra computation and the recursion stops for those RHS, as will be illustrated in Figures 3.3 and 3.4a. With this construction, each leaf of T_{rec} contains RHS with indistinguishable nonzero structures, and keeping them contiguous in the final permutation avoids introducing extra computations. Assuming that for each $R \in T_{rec}$ the children $C(R)$ are ordered, this induces an ordering of all the leaves of the tree, which defines the final RHS sequence. We now explain how the set of children $C(R)$ is built and ordered at each step:

1) Building the set of children The set of children of $R \in T_{rec}$ is built by first identifying the pruned layers U of all RHS $r \in R$. The different pruned layers are stored in $\mathcal{U}$ and we have for example (Figure 3.2, first step of the algorithm), $\mathcal{U} = \{\{u_1\}, \{u_2\}, \{u_1, u_2\}\}$. We define $C(R) = \{R[U] \mid U \in \mathcal{U}\}$ (Definition 3.1), which forms a partition of R. One important

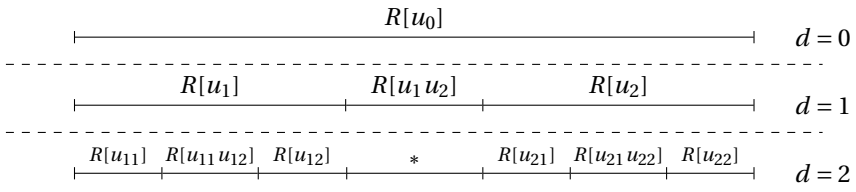


Figure 3.3: A layered sequence built by the flat tree algorithm on a binary tree. Sets $R[\emptyset]$ (not represented) could be added at either extremity of the concerned sequence (e.g., right after $R[u_2]$ for a RHS included in u_0). With the strong assumptions of Figure 3.2, * = $R[u_{11}u_{21}]$, $R[u_{11}u_{12}u_{21}u_{22}]$, $R[u_{12}u_{22}]$. Otherwise, * is more complex.

property is that all $r \in R[U]$ have the same nonzero structure at the corresponding layer so that numbering them contiguously prevents the introduction of extra computation.

2) Ordering the children The children sequence $[R[U_1], \ldots, R[U_{\#C(R)}]]$ at depth $d+1$ should minimize the size of the intervals for nodes u at depth $d+1$ of $T_p(R)$. The order inside each $R[U_i]$ does not impact the size of these intervals (it will only impact lower levels). For any node u at depth $d+1$ in $T_p(R)$, we have

$$\theta(Z_u|_R) = \max(Z_u|_R) - \min(Z_u|_R) + 1 = \sum_{i=i_{\min}(u)}^{i_{\max}(u)} \#R[U_i],$$

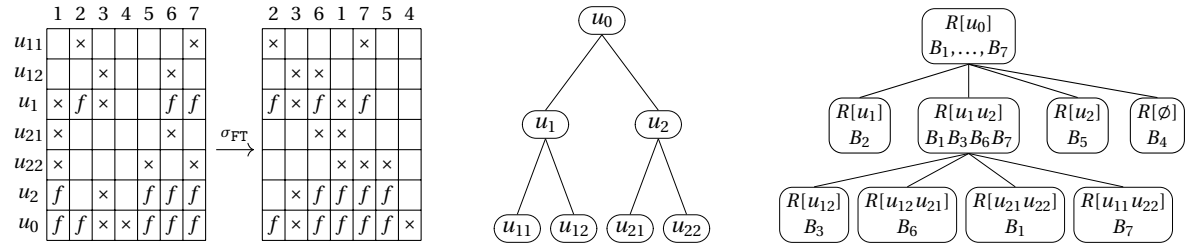
where $Z_u|_R$ is the set of permuted indices representing the active columns restricted to R, and $i_{\min}(u) = \min\{i \in \{1, \ldots, \#C(R)\} \mid u \in U_i\}$ (resp. $i_{\max}(u) = \max\{i \in \{1, \ldots, \#C(R)\} \mid u \in U_i\}$) is the first (resp. last) index i such that $u \in U_i$.

Proof. In the sequence $[R[U_1], \ldots, R[U_{\#C(R)}]]$, $\min(Z_u|_R)$ (resp. $\max(Z_u|_R)$) corresponds to the index of the first (resp. last) column in $R[U_{i_{min}}]$ (resp. $R[U_{i_{max}}]$). Since all columns from $R[U_{i_{min}}]$ to $R[U_{i_{max}}]$ are numbered consecutively, we have the desired result. □

Finally, we minimize the local cost function (sum of the interval sizes for each node at depth $d+1$):

$$\text{cost}([R[U_1], \ldots, R[U_{\#C(R)}]]) = \sum_{\substack{u \in T_p(R) \\ depth(u) = d+1}} \sum_{i=i_{min}(u)}^{i_{max}(u)} \#R[U_i] \tag{3.4}$$

To build the ordered sequence $[R[U_1], \ldots, R[U_{\#C(R)}]]$, we use a greedy algorithm that starts with an empty sequence, and at each step $k \in \{1, \ldots, \#C(R)\}$ inserts a RHS set $R[U]$ picked randomly in $C(R)$ at the position that minimizes (3.4) on the current sequence. To do so, we simply start from one extremity of the sequence of size $k-1$ and compute (3.4) for the new sequence of size k for each possible position $0 \ldots k$; if several positions lead to the same minimal cost, the first one encountered is chosen. In case $u-$optimality is obtained for each node u considered, then the permutation is said to be *perfect* and the cost function is minimal, locally inducing no extra operations on those nodes.



(a) RHS structure. (b) Separator tree T. (c) Recursion tree T_{rec}.

Figure 3.4: Illustration of the algebraic Flat Tree algorithm on a set of 7 RHS. Small example with (a) a RHS structure, (b) the corresponding tree and (c) the recursion tree built by the application of the Flat Tree algorithm on a set of 7 right-hand sides.

Figure 3.3 shows the recursive structure of the RHS sequence after applying the algorithm on a binary tree. We refer to this representation as the *layered sequence*. For simplicity, the notation for pruned layers has been reduced from, e.g., $\{u_1\}$ to u_1, and from $\{u_1, u_2\}$ to u_1u_2. From the recursion tree point of view, $R[u_1], R[u_1u_2], R[u_2]$ are the children of $R[u_0]$ in T_{rec}, $R[u_{11}], R[u_{11}u_{12}], R[u_{12}]$ the ones of $R[u_1]$, etc.

Example 3.1. *Let $B = [B_1, B_2, B_3, B_4, B_5, B_6, B_7]$ be a RHS matrix with the structure presented in Figure 3.4a. Although we still use a binary tree, we make no assumption on the RHS structure, the domain, or the ordering in the example of Figure 3.4. We have $R_B = R[u_0] = \{B_1, B_2, B_3, B_4, B_5, B_6, B_7\}$. The set of pruned layers corresponding to $R[u_0]$ is $\mathcal{U} = \{u_1u_2, u_1, \emptyset, u_2\}$, so that $C(R[u_0]) = \{R[u_1u_2], R[u_1], R[\emptyset], R[u_1u_2]\}$. As can be seen in the non-permuted RHS structure, $R[\emptyset] = B_4$ at depth 1 induces extra operations at nodes descendants of u_0, which disappear when placing $R[\emptyset]$ at one extremity of the sequence. We choose to place it last and obtain the sequence $[R[u_1], R[u_1u_2], R[u_2], R[\emptyset]]$. A recursive call is done on the identified sets, as illustrated in Figure 3.4c. Since $R[u_1]$, $R[u_2]$ and $R[\emptyset]$ contain a single RHS, we focus on $R = R[u_1u_2]$, whose set of pruned layers is $\mathcal{U} = \{u_{21}u_{22}, u_{12}, u_{12}u_{21}, u_{11}u_{22}\}$. The sequence $[R[U_1], R[U_2], R[U_3], R[U_4]]$, where $U_1 = u_{12}$, $U_2 = u_{12}u_{21}$, $U_3 = u_{21}u_{22}$, and $U_4 = u_{11}u_{22}$ is a perfect sequence which gives local optimality. However, taking the problem globally, we see that $\theta(Z_{u_{11}}) \neq \#Z_{u_{11}}$ in the final sequence $[B_2, B_3, B_6, B_1, B_7, B_5, B_4]$.*

Although not relying on geometric assumptions, particular RHS structures or binary trees, computations on explicit zeros (for example zero rows in column $\boldsymbol{f}$ and subdomain $T[u_{11}]$ in Figure 3.2b), may still occur with the flat tree algorithm. This will also be illustrated in Section 3.3, where $\Delta(B, \sigma_{\text{FT}})$ is 39% larger than $\Delta_{min}(B)$, in the worst case. A blocking algorithm is now introduced to further reduce $\Delta(B, \sigma_{\text{FT}})$.

3.2 Towards a minimal number of operations using blocks

In this section, we identify the causes of the remaining extra operations and provide an efficient blocking algorithm to reduce them efficiently while creating a small number of blocks. The algorithm relies on a property of independence of right-hand sides that is first illustrated, and then formalized.

3.2.1 Geometrical intuition

The use of blocking techniques may fulfill different objectives. In terms of operation count, optimality ($\Delta_{min}(B)$) is obtained when processing the columns of B one by one, which implies the creation of n_{rhs} blocks. However, this requires processing the tree n_{rhs} times and will typically lead to a poor arithmetic intensity (and likely a poor performance). On the other hand, the algorithms of Section 3.1 only use one block, which allows a higher arithmetic intensity but leads to extra operations. In the dense case, blocks are also often used to improve the arithmetic intensity. In the sparse RHS case, blocking techniques with regular blocks of columns have been associated to tree pruning to either limit the access to the factors [10], or limit the number of operations [67]. They were either based on a preordering of the columns or on hypergraph models. In this section, to give as much flexibility as possible to the underlying algorithms and avoid unnecessary constraints, our objective is to create a minimal number of (possibly large) blocks while reducing the number of extra operations by a given amount. In particular, we allow blocks to be irregular and assume node intervals are exploited within each block.

On the one hand, two RHS or sets of RHS included in two different domains exhibit interesting properties, as can be observed for sets $\boldsymbol{a} \in T[u_1]$ and $\boldsymbol{c} \in T[u_2]$ from Figure 3.2a. No extra operations are introduced between them: $\Delta([\boldsymbol{a}, \boldsymbol{c}]) = \Delta(\boldsymbol{a}) + \Delta(\boldsymbol{c})$. We say that $\boldsymbol{a}$ and $\boldsymbol{c}$ are *independent sets* and they can be associated together. On the other hand, a set of RHS intersecting a separator (such as set $\boldsymbol{b}$) has zeros and nonzeros in rows common to their adjacent RHS sets ($\boldsymbol{a}$ and $\boldsymbol{c}$) which will likely introduce extra computation. In Figure 3.2b, we have for example $\Delta([\boldsymbol{a}, \boldsymbol{b}]) = \Delta([\boldsymbol{d}, \boldsymbol{e}, \boldsymbol{f}, \boldsymbol{g}, \boldsymbol{h}, \boldsymbol{i}]) > \Delta([\boldsymbol{d}, \boldsymbol{e}, \boldsymbol{f}]) + \Delta([\boldsymbol{g}, \boldsymbol{h}, \boldsymbol{i}]) = \Delta(\boldsymbol{a}) + \Delta(\boldsymbol{b})$ and $\Delta([\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}]) > \Delta(\boldsymbol{a}) + \Delta(\boldsymbol{b}) + \Delta(\boldsymbol{c})$. We say that $\boldsymbol{b}$ is a set of *problematic* RHS. Figure 1.10 (right) gives another example where extracting the problematic RHS B_1 and B_5 from $[B_4, B_2, B_1, B_5, B_6, B_3]$ suppresses all extra operations: $\Delta([B_4, B_2, B_6, B_3]) + \Delta([B_1, B_5]) = \Delta_{min} = 1056$.

To give further intuition on the Blocking algorithm, consider the RHS structure of Figure 3.2b. Problematic RHS $\boldsymbol{e}$ and $\boldsymbol{k}$ in $[\boldsymbol{d}, \boldsymbol{e}, \boldsymbol{f}, \boldsymbol{j}, \boldsymbol{k}, \boldsymbol{l}]$ can be extracted to form two blocks, or *groups*, $[\boldsymbol{e}, \boldsymbol{k}]$ and $[\boldsymbol{d}, \boldsymbol{f}, \boldsymbol{j}, \boldsymbol{l}]$. The situation is slightly more complicated for $[\boldsymbol{g}, \boldsymbol{h}, \boldsymbol{i}]$, where $\boldsymbol{h}$ indeed intersects two separators, u_1 and u_2. In this case, $\boldsymbol{h}$ should be extracted to form the groups $[\boldsymbol{g}, \boldsymbol{i}]$ and $[\boldsymbol{h}]$. We note that the amount of extra operations will likely be much larger when the separator intersected is high in the tree.

Situations where no assumption on the RHS structure is made are more complicated and require a general approach. For this, we formalize the notion of *independence*, which will be the basis for our blocking algorithm.

3.2.2 Algebraic formalization

In this section, we give a first version of the Blocking algorithm. It is based on a sufficient condition allowing to group together sets of RHS without introducing extra computation. We assume the matrix B to be flat tree ordered and the recursion tree T_{rec} to be built and ordered. Using the notations of Definition 3.1, we give an algebraic definition of the independence property:

Definition 3.2. *Let U_1, U_2 be two sets of nodes at a given depth of a tree T, and let $R[U_1], R[U_2]$ be the corresponding sets of RHS. $R[U_1], R[U_2]$ are said to be* independent *if and only if $U_1 \cap U_2 = \emptyset$.*

With Definition 3.2, we are able to formally identify independent sets that can be associated together. Take for example $\boldsymbol{a} = R[u_1]$ and $\boldsymbol{c} = R[u_2]$ (Figure 3.2a), $R[u_1]$ and $R[u_2]$ are independent and $\Delta([R[u_1], R[u_2]]) = \Delta(R[u_1]) + \Delta(R[u_2])$. On the contrary, when $R[U_1], \ldots, R[U_n]$ are not pairwise independent, we group together independent sets of RHS, while forming as few groups as possible. This problem is equivalent to a graph coloring problem, where $R[U_1], \ldots, R[U_n]$ are the vertices and an edge exists between $R[U_i]$ and $R[U_j]$ if and only if $U_i \cap U_j \neq \emptyset$. Several heuristics exist for this problem, and each color will correspond to one group. The blocking algorithm as depicted in Figure 3.5 traverses T_{rec} from top

Algorithm 3.2 Blocking algorithm

for $d = 0$ **to** d_{max} **do**
  $j \leftarrow 0$ /* #groups at depth $d+1$ */
  for all groups g_i^d at detph d **do**
    $(g_{j+1}^{d+1} \ldots g_{j+k}^{d+1}) \leftarrow$ BUILDGROUPS$(g_i^d, d+1)$
    /* k new groups have been created */
    $j \leftarrow j + k$
  end for
end for

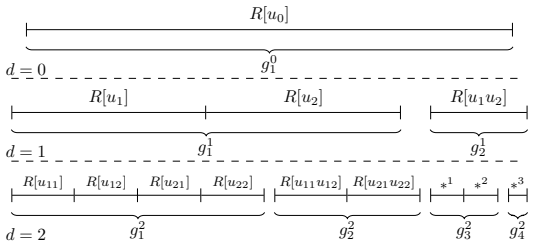


Figure 3.5: A first version of the blocking algorithm (left). It is illustrated (right) on the layered sequence of Figure 3.3. With the geometric assumptions of Figure 3.2, $*^1 = R[u_{11}u_{21}], *^2 = R[u_{12}u_{22}]$, and $*^3 = R[u_{11}u_{12}u_{21}u_{22}]$.

to bottom. At each depth d, each intermediate group g_i^d verifies the following properties: (i) g_i^d can be represented by a sequence $[R[U_1], \ldots, R[U_n]]$, and (ii) the sequence respects the flat tree order of T_{rec}. Then, BUILDGROUPS$(g_i^d, d+1)$ first builds the sets of RHS at depth $d+1$, which are exactly the children of the $R[U_j] \in g_i^d$ in T_{rec}. Second, BUILDGROUPS$(g_i^d, d+1)$ solves the aforementioned coloring problem on these RHS sets and builds the k groups $(g_{j+1}^{d+1}, \ldots, g_{j+k}^{d+1})$.

In Figure 3.5(right), there is initially a single group $g_1^0 = [R[u_0]]$ with one set of RHS, which may be expressed as the ordered sequence $[R[u_1]R[u_1u_2]R[u_2]]$. g_1^0 does not satisfy the independence property at depth 1 because $u_1 \cap u_1u_2 \neq \emptyset$ or $u_2 \cap u_1u_2 \neq \emptyset$. BUILDGROUPS$(g_1^0,$

1) yields $g_1^1 = [R[u_1], R[u_2]]$ and $g_2^1 = [R[u_1u_2]]$. The algorithm proceeds until a maximal depth d_{max}: $(g_1^2, g_2^2) = \text{BUILDGROUPS}(g_1^1, 2)$, $(g_3^2, g_4^2) = \text{BUILDGROUPS}(g_2^1, 2)$, etc. To illustrate the interest of property (ii), let us take sets $\boldsymbol{d} = R[u_{11}]$, $\boldsymbol{f} = R[u_{12}]$, $\boldsymbol{j} = R[u_{21}]$ and $\boldsymbol{l} = R[u_{22}]$ from Figure 3.2b. One can see that $\Delta([\boldsymbol{d}, \boldsymbol{f}, \boldsymbol{j}, \boldsymbol{l}]) = \Delta(\boldsymbol{d}) + \Delta(\boldsymbol{f}) + \Delta(\boldsymbol{j}) + \Delta(\boldsymbol{l}) < \Delta([\boldsymbol{d}, \boldsymbol{j}, \boldsymbol{f}, \boldsymbol{l}])$. Compared to $[\boldsymbol{d}, \boldsymbol{f}, \boldsymbol{j}, \boldsymbol{l}]$ which respects the global flat tree ordering and ensures u_1- and u_2-optimality, $[\boldsymbol{d}, \boldsymbol{j}, \boldsymbol{f}, \boldsymbol{l}]$ does not and would thus increase $\theta(Z_{u_1})$ and $\theta(Z_{u_2})$. Furthermore, Algorithm 3.2 ensures the property, that the independent sets of RHS grouped together do not introduce extra operations:

Property 3.1. *For any group $g^d = [R[U_1], \ldots, R[U_n]]$ created through Algorithm 3.2 at depth d, we have $\Delta([R[U_1], \ldots, R[U_n]]) = \sum_{i=1}^{n} \Delta(R[U_i])$.*

Proof. For $d \geq 1$, let $g^d = [R[U_i^d]_{i=1,\ldots,n^d}]$ be a group at depth d created through Algorithm 3.2 (we use superscripts d in this proof to indicate the depth without ambiguity). Let us split nodes above (A) and below (B) layer d in the pruned tree $T_p(g^d)$. The number of operations to process g^d is:

$$\Delta(g^d) = \sum_{u \in T_p(g^d)} \delta_u \times \theta(Z_u|_{g^d}) = \overbrace{\sum_{u \in A} \delta_u \times \theta(Z_u|_{g^d})}^{\Delta_A} + \overbrace{\sum_{u \in B} \delta_u \times \theta(Z_u|_{g^d})}^{\Delta_B}, \tag{3.5}$$

where $A = \{u \in T_p(g^d) \mid depth(u) < d\}$ and $B = \{u \in T_p(g^d) \mid depth(u) \geq d\}$.
(i) We first consider the term Δ_B. Let $B_i = \{u \in T_p(R[U_i^d]) \mid depth(u) \geq d\}$. Thanks to the independence property of the $R[U_i^d]$ forming g^d, the pruned layers U_i^d in T are disjoint and since T is a tree, we have $B_i \cap B_j = \emptyset$ for all $i \neq j$. Hence, $B = \dot{\bigcup}_{i=1}^{n^d} B_i$, where $\dot{\bigcup}$ denotes the disjoint union. Therefore,

$$\Delta_B = \sum_{u \in \dot{\bigcup}_{i=1}^{n^d} B_i} \delta_u \times \theta(Z_u|_{[R[U_j^d]_{j=1,\ldots n^d}]}) = \sum_{i=1}^{n^d} \sum_{u \in B_i} \delta_u \times \theta(Z_u|_{[R[U_j^d]_{j=1,\ldots n^d}]}).$$

We recall that a RHS r is said to be active at node u if $u \in T_p(r)$. In the inner sum, the only possible active RHS in B_i are the ones that belong to $R[U_i^d]$ (independence of the $R[U_j^d]$), so that for all $u \in B_i$, we have $\theta(Z_u|_{[R[U_i^d]_{i=1,\ldots n^d}]}) = \theta(Z_u|_{R[U_i^d]})$. Therefore, $\Delta_B = \sum_{i=1}^{n^d} \sum_{u \in B_i} \delta_u \times \theta(Z_u|_{R[U_i^d]})$.
(ii) We now consider the term Δ_A. Similarly to (i), we define $A_i = \{u \in T_p(R[U_i^d]) \mid depth(u) < d\}$. We have $A = \bigcup_{i=1}^{n_d} A_i$ but the union is no longer disjoint. Let $T_{rec}(g^d)$ be the restriction to g^d of the recursion tree T_{rec} associated to the flat-tree algorithm applied to R_B (see Section 3.1.2 for the definition of T_{rec}). $T_{rec}(g^d)$ is obtained by excluding at each node of T_{rec} the right-hand sides that are not part of g^d, then by pruning all empty nodes. We also restrict Definition 3.1 to g^d and thus note $R[U] = \{r \in g^d \mid L_d(r) = U\}$. In particular, the root of $T_{rec}(g^d)$ is $R[u_0] = g^d$.

By construction of Algorithm 3.2 (Figure 3.5), we know that any layer at depth $d' < d$ of the group g^d consists of independent sets $R[U_j^{d'}]$ of RHS. Therefore, $\forall u \in A$, $\exists! R[U] \in T_{rec}(g^d)$ such that $u \in U$. This means that the only active columns at node u are those in this unique

$R[U]$ and, since the RHS in $R[U]$ are all contiguous in g^d thanks to the global flat tree ordering, we have $\theta(Z_u|_{R[U]}) = \theta(Z_u|_{g^d}) = \#R[U]$.

Furthermore, by construction of the recursion tree (children nodes form a partition of each parent node), the RHS in $R[U]$ are the ones in the disjoint union of $R[U_i^d] \subset R[U]$, the sets of right-hand sides at layer d that are descendants of $R[U]$ in $T_{rec}(g^d)$. Therefore, $\#R[U] = \sum_{R[U_i^d] \subset R[U]} \#R[U_i^d]$. Furthermore, since the $R[U_i^d]$ such that $R[U_i^d] \subset R[U]$ are contiguous sets in g^d and are all active at node u, we also have $\theta(Z_u|_{R[U_i^d]}) = \#R[U_i^d]$. It follows:

$$\theta(Z_u|_{g^d}) = \sum_{R[U_i^d] \subset R[U]} \theta(Z_u|_{R[U_i^d]}).$$

We define $\xi_i(u) = 1$ if $R[U_i^d] \subset R[U]$ (with $R[U]$ derived from u as explained above), and $\xi_i(u) = 0$ otherwise. The condition $R[U_i^d] \subset R[U]$ means that u is an ancestor of U_i^d nodes in T. Thus, $\xi_i(u) = 1$ for $u \in A_i$ and $\xi_i(u) = 0$ for $u \notin A_i$. We can thus write $\sum_{R[U_i^d] \subset R[U]} \theta(Z_u|_{R[U_i^d]}) = \sum_{i=1}^{n^d} \xi_i(u)\theta(Z_u|_{R[U_i]})$ and redefine Δ_A as:

$$\begin{aligned}
\Delta_A = \sum_{u \in A} \delta_u \times \theta(Z_u|_{g^d}) &= \sum_{u \in A} \delta_u \times \sum_{i=1}^{n^d} \xi_i(u)\theta(Z_u|_{R[U_i^d]}) \\
&= \sum_{i=1}^{n^d} \sum_{u \in A} \delta_u \times \xi_i(u)\theta(Z_u|_{R[U_i^d]}) \\
&= \sum_{i=1}^{n^d} \sum_{u \in A_i} \delta_u \times \theta(Z_u|_{R[U_i^d]}).
\end{aligned}$$

Joining the terms Δ_A and Δ_B, we finally have:

$$\Delta(g^d) = \Delta_A + \Delta_B = \sum_{i=1}^{n^d} \sum_{u \in B_i} \delta_u \times \theta(Z_u|_{R[U_i]}) + \sum_{i=1}^{n^d} \sum_{u \in A_i} \delta_u \times \theta(Z_u|_{R[U_i]}) = \sum_{i=1}^{n^d} \Delta(R[U_i])$$

□

Interestingly, Property 3.1 can be used to prove, in the case of a single nonzero per RHS, the optimality of the flat tree permutation.

Corollary 3.1. *Let R_B be a set of RHS such that $\forall r \in R_B, \#V_r = 1$. Then the flat tree permutation is optimal: $\Delta(R_B) = \Delta_{min}(R_B)$.*

Proof. Since $\forall r \in R_B, \#V_r = 1$, $T_p(r)$ is a branch of T. As a consequence, any set of RHS $R[U]$ built through the flat tree algorithm is represented by a pruned layer U containing a single node u. At each step of the flat tree algorithm (Algorithm 3.1), the RHS sets identified are thus all independent from each other. When applying Algorithm 3.2, a unique group R_B is then kept until the bottom of the tree. Blocking is thus not needed and Property 3.1 applies at each level of the recursion. $\Delta(R_B)$ is thus equal to the sum of the $\Delta(R[U])$ for all leaves $R[U]$ of the recursion tree T_{rec}. Since $\Delta(R[U]) = \Delta_{min}(R[U])$ on those leaves (all RHS in $R[U]$ involve the exact same nodes and operations), we conclude that $\Delta(R_B) = \Delta_{min}(R_B)$. □

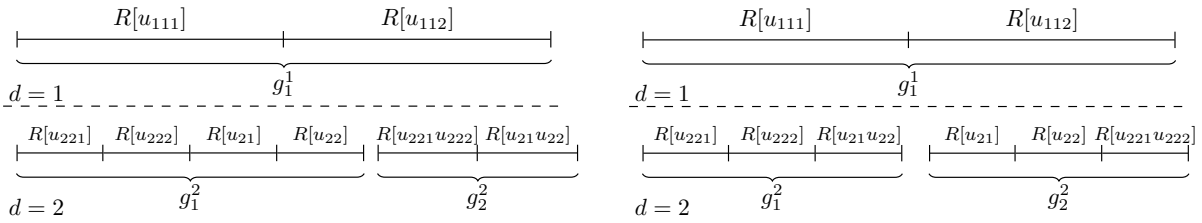


Figure 3.6: Two strategies to build groups: `CritPathBuildGroups` (left) and `RegBuildGroups` (right).

This proof is independent of the ordering of the children at step 2 of Algorithm 3.1. Corollary 3.1 is thus more general: any top-down recursive ordering keeping together RHS with identical pruned layer at each step is optimal, as long as the pruned layers identified at each step are independent.

Back to the BUILDGROUPS function, the solution of the coloring problem may not be unique. Even on the simple example of Figure 3.5, there are several ways to define groups, as shown in Figure 3.6 for g_1^1: both strategies satisfy the independence property and minimize the number of groups. The CRITPATHBUILDGROUPS strategy tends to create a large group g_1^2 and a smaller one, g_2^2. In each group the computations on the tree nodes are expected to be well balanced because all branches of the tree rooted at u_0 might be covered by the RHS (assuming thus a reasonably balanced RHS distribution over the tree). The choice of CRITPATHBUILDGROUPS can be driven by tree parallelism considerations, namely, the limitation of the sum of the operation counts on the *critical paths* of all groups. The REGBUILDGROUPS strategy tends to balance the sizes of the groups but may create more unbalance regarding the distribution of work over the tree.

We note that for a given depth, applying BUILDGROUPS on *all* groups may not always be necessary, and that for a given group, enforcing the independence property may create more than two groups. In the next section we minimize the number of groups created using greedy heuristics.

3.2.3 A greedy approach to minimize the number of groups

Compared to Algorithm 3.2, Algorithm 3.3 adds the group selection, limits the number of groups created from a given group to two, and stops depending on a given tolerance on the amount of operations.

First, instead of stepping into each group as in Algorithm 3.2, we select among the current groups the one responsible for most extra computation, that is, the one maximizing $\Delta(g) - \Delta_{min}(g)$. This implies that groups that are candidate for splitting might have been created at different depths and we use a superscript to indicate the depth d at which a group was split, as in the notation g_0^d.

Second, instead of a coloring problem which creates as many groups as colors obtained, we look (procedure BUILDMAXINDEPSET) inside the RHS sets of g_0^d for a maximal group of independent sets at depth $d+1$, denoted g_{imax}^{d+1}. The other sets are left in another group g_c^d,

whose depth remains equal to d. g_c^d may thus consist of dependent sets that may be subdivided later if needed[1]. Rather than an exact algorithm to determine g_{imax}^{d+1}, we use a greedy heuristic.

Finally, we define μ_0 as the tolerance of extra operations authorized. With a typical value $\mu_0 = 1.01$, the algorithm stops when the number of extra operations is within 1% of the minimal number of operations Δ_{min}, returning G as the final set of groups.

Algorithm 3.3 Blocking algorithm

$G \leftarrow \{R_B\}$, $\Delta_{min} \leftarrow \Delta_{min}(R_B)$, $\Delta \leftarrow \Delta(R_B)$
while $\Delta/\Delta_{min} > \mu_0$ **do**
  Select g_0^d such that $\Delta(g_0^d) - \Delta_{min}(g_0^d) = \max_{g \in G} (\Delta(g) - \Delta_{min}(g))$ ▷ Group selection
  $(g_{imax}^{d+1}, g_c^d) \leftarrow$ BUILDMAXINDEPSET(g_0^d, d+1)
  $G \leftarrow G \cup \{g_{imax}^{d+1}, g_c^d\} \setminus \{g_0^d\}$
  $\Delta \leftarrow \Delta - \Delta(g_0^d) + \Delta(g_{imax}^{d+1}) + \Delta(g_c^d)$
end while

3.3 Experimental results

In this section, we report on the impact of the proposed permutation and blocking algorithms on the forward substitution (Equation (3.2)), using a set of 3D regular finite difference problems coming from seismic and electromagnetism modeling [4, 60], for which the solve phase is costly. The characteristics of the corresponding matrices and RHS are presented in Table III. In both applications, the nonzeros of each RHS correspond to a small set of close points, near the top of the 3D grid corresponding to the physical domain, with some overlap between RHS. Except in Section 3.3.3, a geometric nested dissection (ND) algorithm is used to reorder the matrix.

3.3.1 Impact of the flat tree algorithm

We first introduce the terminology used to denote the different strategies developed in this study and that impact the number of operations Δ. `DEN` represents the dense case, where no optimization is used to reduce Δ, and `TP` means tree pruning. When column intervals are exploited at each tree node, we denote by `RAN`, `INI`, `PO` and `FT` the random, initial ($\sigma = id$), postorder ($\sigma_{\texttt{PO}}$) and flat tree ($\sigma_{\texttt{FT}}$) permutations, respectively.

The improvements brought by the different strategies are presented in Table I. Compared to the dense case, `TP` divides Δ by at least a factor 2. When column intervals are exploited at each node, the large gap between `RAN` and `INI` shows that the original column order holds geometrical properties. `FT` behaves better than `INI` and `PO` and gets reasonably close to Δ_{min}. Overall, `FT` provides a 13% gain on average over `PO`. However, the gain on Δ decreases from 25% on the 10Hz problem to 1% on the H116 problem. This can be explained by the fact that B is denser for the seismic applications than for the electromagnetism applications (see Table III).

[1] In case g_c^d consists of independent sets and is selected, $g_c^{d+1} = g_c^d$ will only be subdivided at depth $d + 2$.

Table I: Number of operations ($\times 10^{13}$) during the forward substitution ($LY = B$) according to the strategy used (ND ordering).

Δ	DEN	TP	RAN	INI	PO	FT	Δ_{min}
5Hz	1.73	.74	.74	.44	.36	**.28**	.22
7Hz	5.94	2.54	2.52	1.46	1.21	**.92**	.69
10Hz	20.62	9.01	8.92	4.78	3.85	**2.87**	2.26
H0	.39	.11	.11	.086	.070	**.057**	.050
H3	7.19	3.33	3.31	2.48	1.47	**1.26**	.95
H17	81.34	37.15	36.97	27.52	10.41	**10.21**	10.12
H116	990.02	448.31	445.91	327.89	123.79	**121.76**	120.68
S3	13.36	4.98	4.91	3.73	2.65	**2.17**	1.71
S21	156.20	49.04	48.07	35.42	25.73	**22.53**	19.43
S84	983.48	286.57	282.70	222.59	161.87	**138.56**	118.51
D30	71.60	39.78	39.38	19.49	10.93	**10.21**	7.31

Table II: Theoretical tree parallelism according to the strategy used (ND ordering).

S	DEN	TP	RAN	INI	PO	FT
5Hz	8.60	3.91	3.88	3.11	2.39	**2.54**
7Hz	8.92	3.97	3.94	3.02	2.25	**2.48**
10Hz	9.10	4.04	4.02	2.96	2.30	2.30
H0	5.88	2.11	2.11	1.75	1.51	1.45
H3	5.99	3.22	3.21	2.47	2.02	2.11
H17	6.32	3.34	3.32	2.54	2.00	1.97
H116	7.92	3.63	3.61	2.75	2.05	2.02
S3	6.12	2.84	2.83	2.18	1.73	1.61
S21	6.30	2.56	2.46	1.85	1.49	1.47
S84	8.01	2.41	2.38	1.90	1.53	1.52
D30	8.50	4.73	4.70	2.86	2.05	2.56

Table III: Impact of the number of groups NG on the normalized operation count, until Δ_{NG}/Δ_{min} becomes smaller than the tolerance $\mu_0 = 1.01$ (ND ordering).

Δ_{NG}/Δ_{min}	FT	NG= 2	NG= 3	NG= 4	NG= 5
5Hz	1.283	1.111	1.001	x	x
7Hz	1.321	1.116	1.002	x	x
10Hz	1.269	1.029	1.002	x	x
H0	1.148	1.029	1.010	1.002	x
H3	1.329	1.068	1.027	1.005	x
H17	1.009	x	x	x	x
H116	1.009	x	x	x	x
S3	1.275	1.120	1.045	1.012	1.003
S21	1.160	1.037	1.015	1.003	x
S84	1.169	1.041	1.015	1.002	x
D30	1.397	1.082	1.058	1.024	1.004

Indeed, the sparser B, the closer we are to a single nonzero per RHS in which case both FT and PO are optimal.

Second, we evaluate the impact of exploiting RHS sparsity on tree parallelism. Table II gives the maximal theoretical speed-up S that can be reached using tree parallelism only (node parallelism is also needed, for example on the root). It is defined as $S = \frac{\Delta}{\Delta_{cp}}$ where Δ_{cp} is the number of operations on the critical path of the tree. We observe that, when sparsity is exploited, tree parallelism is significantly smaller than in the dense case. This is because the depth of the pruned tree $T_p(B)$ is similar to that of the original tree (some nonzeros of B generally appear in the leaves), while the tree effectively processed is pruned and thus the overall amount of operations is reduced. For the same reason, S is smaller for test cases where $D(B)$ is small. For the 5Hz, 7Hz, and 10Hz problems which have more nonzeros per column of B, besides decreasing the operation count more than the other strategies, FT exhibits equivalent or even better tree parallelism than PO. For such matrices, where $D(B)$ is large, FT balances

the work on the tree better than PO and reduces the work on the critical path more than the total work. Overall, FT reduces the operation count better than any other strategy and has good parallel properties.

3.3.2 Impact of the blocking algorithm

First, we show that the blocking algorithm decreases the operation count Δ while creating a small number of groups. Second, we discuss parallel properties of the clustering strategies illustrated in Figure 3.6. In Table III, we report the value of $\frac{\Delta_{NG}}{\Delta_{min}}$ as a function of the number of groups created. x means that the blocking algorithm stopped because the condition $\Delta_{NG}/\Delta_{min} \leq \mu_0$ was reached, with μ_0 for Algorithm 3.3 set to 1.01. Computing from Table III the ratio of extra operations reduction $1 - \frac{\Delta_{NG}-\Delta_{min}}{\Delta_1-\Delta_{min}}$ for NG groups created, we observe an average reduction of 74% of the extra operations when $NG = 2$, *i.e.*, when only two groups are created. Table III also shows that Δ_{NG} reaches a value close to Δ_{min} very quickly.

Table IV: Sum of critical paths' operations ($\times 10^{13}$) for two grouping strategies when three groups are created.

$\sum_g \Delta_{cp}(g)$	5Hz	7Hz	10Hz	H0	H3
CRITPATHBUILDGROUP	**.092**	**.30**	**1.00**	**.037**	**.50**
REGBUILDGROUP	.12	.43	1.58	.044	.72

In Table IV, we report the sum of operation counts on the critical paths Δ_{cp} over all groups created using CRITPATHBUILDGROUP and REGBUILDGROUP strategies, when the number of groups created is three, leading to a value Δ close to Δ_{min}, see column "NG=3" of Table III. In this case, the total number of operations Δ during the forward solution phase on all groups is equal whether we use CRITPATHBUILDGROUP or REGBUILDGROUP. Tree parallelism is thus a crucial discriminant between both strategies, and we indeed observe in Table IV that CRITPATHBUILDGROUP effectively limits the length of critical paths over the three groups created, justifying its use.

3.3.3 Experiments with other orderings

As mentioned earlier, several orderings may be used to order the unknowns of the original matrix, thanks to the algebraic nature of our flat tree and blocking algorithms. Although local ordering methods (AMD, AMF as provided by the MUMPS package[2]) are known not to be competitive with respect to algebraic nested dissection-based approaches such as SCOTCH[3] or METIS[4] on large 3D problems, we include them in Table V in order to study how the flat tree and blocking algorithms behave in general situations.

First, an important aspect of using other orderings is that they often produce much more irregular trees, leading to a large number of pruned layers to sequence. The FT permutation

[2] http://mumps.enseeiht.fr/
[3] http://www.labri.fr/perso/pelegrin/scotch/
[4] http://glaros.dtc.umn.edu/gkhome/metis/metis/overview

Table V: Operation count $\Delta(\times 10^{13})$ for permutation strategies PO and FT, and number of groups NG required to reach $\frac{\Delta_{NG}}{\Delta_{min}} \leq 1.01$ for blocking strategies REG and BLK. Different orderings (AMD, AMF, SCOTCH, METIS) are used.

	AMD					AMF					SCOTCH					METIS				
σ	PO	REG	FT	BLK	Δ_{min}	PO	REG	FT	BLK	Δ_{min}	PO	REG	FT	BLK	Δ_{min}	PO	REG	FT	BLK	Δ_{min}
	Δ	NG	Δ	NG		Δ	NG	Δ	NG		Δ	NG	Δ	NG		Δ	NG	Δ	NG	
5Hz	1.44	53	**1.36**	4	1.25	**.75**	51	.87	7	.68	.47	328	**.32**	3	.25	.43	230	**.30**	3	.24
7Hz	5.03	38	**4.44**	4	4.35	**15.3**	18	17.8	12	14.9	1.60	287	**1.14**	3	.86	1.42	230	**1.08**	3	.82
10Hz	**19.3**	154	19.8	3	15.3	**96.9**	253	99.1	11	82.1	5.86	288	**4.21**	3	3.05	4.67	281	**3.44**	3	2.53
H0	.54	533	**.53**	4	.47	**.12**	333	.12	5	.091	.073	499	**.063**	3	.055	.077	615	**.067**	3	.057
H3	135	380	**106**	5	9.07	**101**	63	134	17	9.26	2.19	615	**1.76**	5	1.18	1.95	533	**1.53**	5	1.12
H17	183	266	**226**	7	136	**467**	173	558	50	396	22.8	380	**19.0**	5	12.6	21.49	242	**16.8**	4	12.3
H116	2244	1	2244	1	2244	**39383**	1	39384	1	39383	291	109	**224**	4	153	264	78	**215**	4	157
S3	20.9	725	**17.9**	6	15.1	**20.7**	184	24.8	10	17.8	4.5	771	**3.41**	5	2.72	3.24	771	**2.64**	5	2.09
S21	393	685	**349**	5	311	**1141**	493	1352	77	831	50.9	492	**39.6**	4	31.7	34.3	223	**28.5**	5	24.9
S84	3025	352	**2848**	5	2501	**38664**	725	45346	213	30977	289	286	**228**	4	193	207	171	**174**	4	151
D30	**115**	111	121	8	94.5	**1015**	139	1280	75	825	16.7	156	**12.8**	5	8.77	15.5	144	**13.0**	5	8.61

reduces the operation count significantly with SCOTCH and METIS, for which we observe an average 31% and 26% reduction compared to the PO permutation. Gains are also obtained with AMD for most test cases. However, FT does not perform well with AMF. This can be explained by the fact that AMF produces too irregular trees which do not fit well with our design of the FT strategy.

Second, we evaluate the blocking algorithm (BLK) and compare it with a regular blocking algorithm (REG) based on the PO permutation, that divides the initial set of columns into regular chunks of columns. Table V shows that the number of groups required to reach $\frac{\Delta}{\Delta_{min}} \leq 1.01$ is much smaller for BLK than for REG in all cases. Our blocking algorithm is very efficient with most orderings except AMF, where the number of groups created is high (but still lower than REG).

3.3.4 Sequential performance

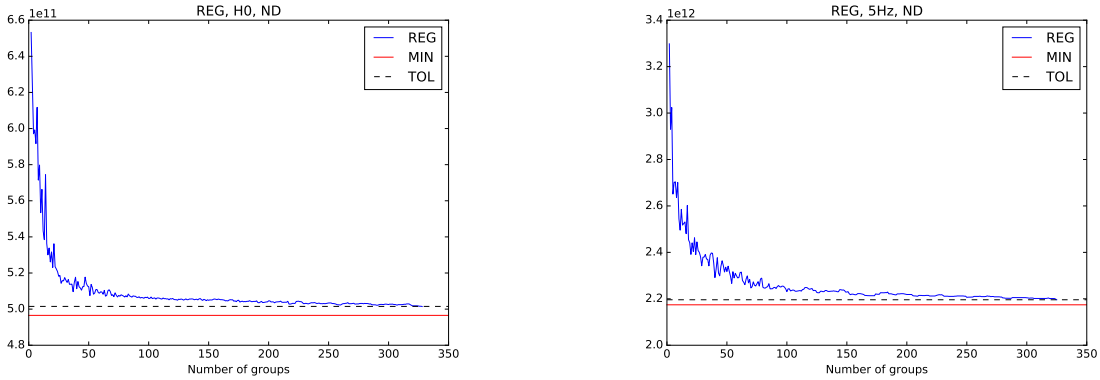
We analyze in this section the time for the forward substitution and for the flat tree and blocking algorithms on a single Intel Xeon core @2.2GHz. A performance analysis in multithreaded or distributed environment for the largest problems is out of the scope of this study. In Table VI, we report the time of the forward substitution of the MUMPS solver[5] and the percentage of time spent in BLAS operations, excluding the time for data manipulation and copies. Timings with the regular blocking REG to reach our target number of operations ($\frac{\Delta_{NG}}{\Delta_{min}} \leq 1.01$), at the cost of a larger number of groups, $NG \gg 1$, are also indicated.

Table VI shows that the time reduction is in agreement with the operations reduction reported in Table I. One can also notice that when exploiting sparsity, the proportion of time spent in BLAS operations increases. This is due to the fact that the relative weight of the top of the tree (with larger fronts for which time is dominated by BLAS operations) is larger when sparsity is exploited. When targeting a reduction of the number of operations with a regular blocking (REG), the large number of blocks (last column of Table VI) makes the granularity of the BLAS operations too small to reach the performance of the BLK strategy. On the other

[5] http://mumps-solver.org

Table VI: Time (seconds) of the forward substitution according to the strategy used and number of groups NG created with BLK and REG (ND ordering). The percentage of actual computation time (BLAS) is indicated in parentheses.

	DEN	TP	INI	PO	REG	FT	BLK	NG_{BLK}	NG_{REG}
5Hz	3132 (34)	561 (80)	305 (88)	246 (89)	429 (95)	190 (89)	148 (91)	3	328
7Hz	9101 (40)	1727 (89)	951 (93)	784 (94)	1137 (97)	594 (94)	460 (95)	3	255
H0	862 (58)	161 (82)	125 (85)	97 (89)	88 (96)	79 (89)	67 (92)	4	328
H3	12419 (72)	4478 (92)	3274 (94)	1901 (96)	1709 (98)	1624 (96)	1234 (97)	4	306
S3	26328 (64)	6839 (89)	5005 (92)	3429 (95)	3450 (99)	2804 (96)	2195 (97)	5	536



(a) Matrix H3 from EMGS.

(b) Matrix 5Hz from SEISCOPE.

Figure 3.7: Number of operations for the forward substitution as a function of the number of groups created with the REG strategy. MIN and TOL are respectively Δ_{min} and $1.01 \times \Delta_{min}$. ND ordering is used.

hand, Table VII shows that larger regular blocks improve the Gflop rate but are not competitive with respect to BLK because of the increase in the number of operations. We also observe that the best block size for REG is problem-dependent.

In Figure 3.7, we represented for the two matrices H3 and 5Hz the number of operations when increasing the number of regular groups created. This figure confirms the difficulty of the REG strategy to decrease efficiently the number of operations down to a low threshold, here $1.01 \times \Delta_{min}$. Figures 3.7a and 3.7b both show a decrease for the first 50 groups but then the REG strategy experiences a plateau before reaching the requested threshold. We also mention the irregularity of the decreasing and expect the same behavior for other matrices. Overall, it is difficult to determine the number of groups that should be chosen for the ReG strategy.

Table VIII relates the execution time of the flat tree and blocking algorithms. The execution times are reasonable compared to the corresponding execution time of the forward substitution. Moreover, we observe that the time of the flat tree algorithm only slowly increases with the problem size. The time of the blocking algorithm, due to its limited number of iterations, is not critical.

Table VII: Operation count ($\times 10^{13}$) and time (seconds) of REG for different block sizes.

REG	64		128		256		512		1024	
	Δ	T_s	Δ	T_s	Δ	T_s	Δ	T_s	Δ	T_s
5Hz	.24	188	.25	179	.27	188	.27	188	.31	209
7Hz	.74	558	.78	538	.86	575	.93	611	.98	643
H0	.051	77	.051	74	.052	72	.054	74	.058	79
H3	.98	1447	1.02	1405	1.04	1369	1.09	1407	1.16	1499
S3	1.75	2533	1.78	2448	1.80	2358	1.86	2379	1.91	2424

Table VIII: Time (seconds) of the flat tree and blocking algorithms for different orderings.

	ND		AMD		AMF		SCOTCH		METIS	
	FT	BLK	FT	BLK	FT	BLK	FT	BLK	FT	BLK
5Hz	.77	.11	.64	.19	1.35	.34	.79	.20	1.02	.13
7Hz	1.30	.30	1.56	.62	5.68	3.9	1.85	.56	2.48	.34
H0	.09	.04	1.72	.04	.13	.04	.09	.02	.12	.04
H3	.52	.26	1.56	.29	1.15	3.0	.55	.24	1.15	.54
S3	.88	.33	2.20	.41	2.51	1.3	.84	.36	1.75	.49

3.4 Guided nested dissection

Given an ordering and a tree, one may think of moving the unknowns corresponding to RHS nonzeros to supernodes higher in the tree with on the one hand, a smaller pruned tree, but on the other hand, an increase in the factor size due to larger supernodes. Better, one may guide the ordering to include as many nonzeros of B as possible within separators during the top-down nested dissection and prune larger subtrees. This will however involve a significant extra cost for applications where each RHS contains several contiguous nodes in the grid, *e.g.*, form a small parallelepiped. For such applications, the geometry of the RHS nonzeros could however be exploited. A first idea avoids problematic RHS by choosing separators that *do not* intersect RHS nonzeros. Although this idea could be tested by adding edges between RHS nonzeros before applying SCOTCH or METIS, this does not appear to be so useful in our applications, where we observed significant overlap between successive RHS. Another idea, when all RHS are localized in a specific area of the domain, is to shift the separators from the nested dissection to insulate the RHS in a small part of the domain. Such a modification of the ordering yields an unbalanced tree in which the RHS nonzeros appear at the smaller side of the tree, improving the efficiency of tree pruning and resulting in a reduction of Δ_{min}, and thus Δ. This so called *guided* nested dissection was implemented and tested on the set of test cases shown in Table IX, where we observe that the number of operations Δ_{min} is decreased, as expected. Since the factor size has also increased significantly, a trade-off may be needed to avoid increasing too much the cost of the factorization.

Table IX: Number of operations Δ_{min} and factor size for original (ND) and guided (GND) nested dissection orderings.

Matrices	5Hz		7Hz		10Hz		H0		H3	
Strategy	ND	GND	ND	GND	ND	GND	ND	GND	ND	GND
$\Delta_{min}(\times 10^{13})$	.22	**.19**	.69	**.62**	2.26	**1.99**	.050	**.025**	.95	**.81**
factor size ($\times 10^9$)	**3.72**	5.18	**12.8**	19.7	**44.8**	73.4	**.24**	.37	**4.50**	5.57

3.5 Applications and related problems

We describe applications where our contributions can be applied. When only part of the solution is needed, one can show that the approaches described in Section 1.3.2 can be applied to the backward substitution ($UX = Y$), which involves similar mechanisms as the forward substitution [61, 69]. The backward substitution traverses the tree nodes from top to bottom so that the interval mechanism is reversed, *i.e.*, the interval from a parent includes the intervals from its children and the properties of local optimality are preserved. If the structure of the partial solution requested differs from the RHS structure, another call to the flat tree algorithm must then be performed to optimize the number of operations. Exploiting sparsity also in the backward step can for instance be useful in some augmented approaches [70] to deal with small matrix updates without complete refactoring, and in some 3D EM geophysics applications [60]. Another application of this work is the computation of Schur complements, where instead of truncating a factorization of the whole system $\left(\begin{smallmatrix} A & C \\ B & D \end{smallmatrix}\right)$, one exploits the factorization of A to use triangular solves with sparse RHS. Taking the symmetric case where $C = B^T$, the Schur complement S can be written $S = D - BA^{-1}B^T = D - B(LL^T)^{-1}B^T = D - (L^{-1}B^T)^T(L^{-1}B^T)$, as in the PDSLin solver [67]. Since B is sparse, $B' = L^{-1}B^T$ can be computed thanks to the algorithms developed in this chapter before computing the sparse product $B'^T B'$.

Finally, we comment on related problems and algorithms. We indicated that the blocking algorithm is closely related to graph algorithms like coloring and maximum independent set. Concerning the minimization problem (1.12) which we addressed with the flat tree algorithm, it can also be regarded globally: using the structure of $L^{-1}B$, the objective is to find a permutation of the columns that minimizes the sum of the intervals weighted with δ_u. This interval minimization problem is similar to a sparse matrix profile reduction problem [20, 54] (and we thus suspect it to be NP-complete). As mentioned in the introduction, hypergraph models have been used in the context of blocking algorithms, with different constraints and objectives compared to ours [10, 67]. Modeling $L^{-1}B$ as an hypergraph might lead to other heuristics than the flat tree algorithm using some variants of hypergraph partitioning, although dense parts in $L^{-1}B$ might need special treatment. One advantage of our permutation and blocking algorithms is that, instead of tackling the problem globally, they decompose the problem into easier subproblems with low complexity by making use of the separator tree T, thereby exploiting the fact that $L^{-1}B$ has a very special structure closely related to the tree. In the context of general unsymmetric matrices, the structure of the solution of the forward step is given by the set of reachable vertices in the elimination dag of L^T [36]. To make the elimination dag a tree to be used in our context, one could consider adding a limited number of entries in L. Similarly to the case of matrices with a symmetric or quasi-symmetric pattern for which the elimination tree of the matrix $A + A^T$ is used, one could add entries in L having a symmetric counterpart in U. Another possibility is to use the work presented in [32] that extends the notion of elimination tree to unsymmetric matrices by considering paths in the factor matrices to characterize the elimination tree. How useful this generalization of the elimination tree can be in our context would deserve to be further studied.

3.6 Conclusion

We introduced permutation and blocking algorithms to improve the tree pruning [61] and the node interval [11] algorithms. A first contribution is the "flat tree" algorithm which permutes RHS to reduce the cost of the forward substitution. A second contribution is a blocking algorithm that further decreases this cost by adequately choosing groups of RHS that can be processed together. Although both algorithms are based on geometrical observations, they are designed with an algebraic approach, giving a general scope to this work. Notions of node optimality and RHS independence were introduced and formalized, together with theoretical properties to provide insight and to support the proposed algorithms. Experimental results on real test cases showed the effectiveness of both the flat tree and the blocking algorithms. Compared to a postorder-based permutation, the flat tree permutation showed an average (resp. maximum) gain of 13% (resp. 25%) on the total operation count with a nested dissection ordering, and interesting parallel properties. Moreover, results with the blocking algorithm validated our approach since only a handful of groups is created compared to several hundreds when using a regular blocking technique. Furthermore, sequential performance results confirmed the good potential of the proposed approaches. Finally, we have discussed possible variants of the nested dissection ordering to exploit RHS sparsity, and discussed possible applications of these contributions regarding for example the backward substitution and the computation of Schur complements.

Chapter 4

On the parallel efficiency of the solve phase with multiple sparse right-hand sides

In this chapter we consider an approach to solve marine controlled-source electromagnetic (CSEM) for which the solution of sparse linear equations with a large number of sparse right-hand sides (few thousands RHS) is required. In this context, the solution phase becomes the most critical step of the complete simulation. This observation has motivated a very fruitful collaboration with a research group at EMGS ASA[1].

In this application, each RHS corresponds to a CSEM source or receiver and one of the objectives of this chapter is to carefully explain the relations between the application and the sparsity structure of both the right-hand sides and requested entries of the solution at each iteration of a Gauss-Newton method. To explain how sparsity can be exploited to reduce operations and to exploit parallelism, we relate the sparsity and the ordering of the columns of the RHS to geometric properties of the sources/receivers of the CSEM application. Doing so it is possible to give a geometric intuition of the proposed algorithms without using all the graph formalism introduced in the previous chapter. We also describe performance improvement of the solve phase for applications with many RHS.

Note that this work has been done in parallel of Chapter 3 so that results do not take into account the flat tree and blocking algorithms introduced in Chapter 3. This will be the object of Chapter 5. This chapter is intended to be self-contained and thus may briefly redefine notions that were previously introduced in Section 1.3.2.

4.1 Introduction

It was demonstrated in 2002 that marine controlled-source electromagnetic (CSEM) method could be used to detect offshore hydrocarbon reservoirs [33]. Over years the CSEM method has become an established tool for oil and gas exploration [24], and the technology development

[1] www.emgs.com

keeps going at a high pace [40]. Successful interpretation of the growing volume of geophysical CSEM data, including also land EM data [62], requires efficient large-scale 3D electromagnetic (EM) modeling algorithms.

Among various approaches to handle 3D EM problems, the most popular is to solve a sparse linear system of frequency-domain Maxwell equations built using finite-difference or finite-element methods [18, 23]. Recent applications of the Gauss-Newton inversion algorithm to large-scale marine CSEM problems indicates that it is very efficient and will likely become the standard inversion approach in the nearest future [51]. The Gauss-Newton method requires that the linear system is solved for all transmitter positions in a given survey often resulting in several thousands of RHSs. The system of linear equations may then take the form $\boldsymbol{MX} = \boldsymbol{S}$, where $\boldsymbol{M}$ is a sparse symmetric matrix of size $n \times n$, while RHS matrix $\boldsymbol{S}$ and solution $\boldsymbol{X}$ are of size $n \times m$. Such systems can be solved with iterative methods which in general are relatively cheap in terms of memory and computational requirements, but may have convergence issues in some cases. In this case the cost is proportional to the number of right-hand sides, i.e. the number of EM sources. Direct methods are numerically robust and well suited to multi-source simulations since once the matrix factorization is performed, only the solve phase needs be applied on all the right-hand side vectors. The complexity of the factorization phase can be a bottleneck for large 3D problems since the number of floating-point operations scales as $O(n^2)$ and the number of nonzero entries in the factors is $O(n^{4/3})$ which is also proportional to the complexity of the solve phase. However, in the context of CSEM applications, it has been shown [60] that using Block Low-Rank (so-called *BLR*) format and related approximations significantly improves the performance of the direct approach, making it competitive with respect to iterative approaches. Indeed, the complexity of the factorization can be reduced to $O(n^{4/3})$ for a simple block low-rank (BLR) format [5] or $O(n)$ for fully structured hierarchical formats [65]. Thanks to the improvements of the factorization phase due to low rank compression (further improved in [5, 6] with respect to [60]), and since CSEM modeling involves thousands of right-hand sides, the time needed to perform the complete CSEM simulation becomes largely dominated by the solve phase of the direct solver. Furthermore, the complexity of the solve step $O(n^{4/3})$ (multiplied by the number of columns in $\boldsymbol{S}$) becomes comparable to that of a low-rank factorization, so that the cost of the solve phase will become more critical as the sizes of the problems grow.

To improve the performance of the solve phase in the context of many right-hand side vectors, we first explain how to exploit both the structure of CSEM sources and the fact that only a partial solution might be needed. Indeed, the positions of the sources in the 3D domain define the sparsity structure of the right-hand sides and the locations where the solution is required. It was showed in [35] that the nonzero structure of $\boldsymbol{Y}$, after the forward substitution, could be predicted beforehand so that one could only target the nonzero variables and thus limit the number of operations, see Chapter 1. This was also referred to as *tree pruning* in [61]. In the context of computing selected entries of the inverse of a matrix [11], the notion of intervals combined to an appropriate column permutation of the right-hand sides was introduced. We will explain why such techniques can be applied or adapted in the context of the CSEM application.

In the chapter we also introduce several new algorithms and techniques to improve the performance of the solution phase for CSEM applications on modern parallel architectures. In a distributed memory parallel environment, how computational tasks are mapped onto the

computer nodes is critical for performance. Mapping controls the equilibration of the work between processors and is driven by factorization phase metrics. We show that using workloads metrics from the solution phase can improve the overall performance of the CSEM simulation. Finally to benefit from good arithmetic intensity and parallelism, large blocks of right-hand sides must be processed simultaneously. For this approach to be efficient, we show that locality of computations should be improved during the solve phase, especially when several threads are used within each distributed memory process (MPI process). This corresponds to an hybrid distributed-multithreaded setting, well adapted to the clusters of multicore processors that we target in this type of applications.

This chapter is organized as follows. In Section 4.2, we describe our frequency-domain finite-difference EM modeling approach in the context of nested dissection, and emphasize on the structure of the right-hand sides. We also describe the test problems and show how much the cost of the solve phase of a direct solver can be critical for CSEM applications, compared to the other phases of a direct solver. We consider direct methods based on a multifrontal approach [28, 30] even if the proposed algorithms are more general and could also be applied to other sparse direct methods. A brief background on direct solvers, with a focus on the solve phase is then provided. In Section 4.3, we explain how the sparse structure of the right-hand sides (RHS) may influence the solve phase and can be used to reduce the amount of computations. In Section 4.4, we discuss parallel aspects of the solution phase. First, we propose strategies to balance the workload for the solution phase, which differ from the ones typically used for the factorization. Then, we show that RHS sparsity and parallelism are contradictory objectives and present ways to group RHS columns together to recover some parallelism. While RHS sparsity can only be exploited during the forward substitution, we show in Section 4.5 that the same ideas can be transposed to the backward substitution, leading to further computational gains. This is due to the particularity of the CSEM application where only part of the solution may be needed. Section 4.6 then studies and illustrates the effects of each of the proposed algorithms on the performance of the solve phase. The global results are summarized in Section 4.6.3, showing that thanks to this work, a direct approach becomes very competitive with the iterative approach previously used. Concluding remarks are provided in Section 4.7.

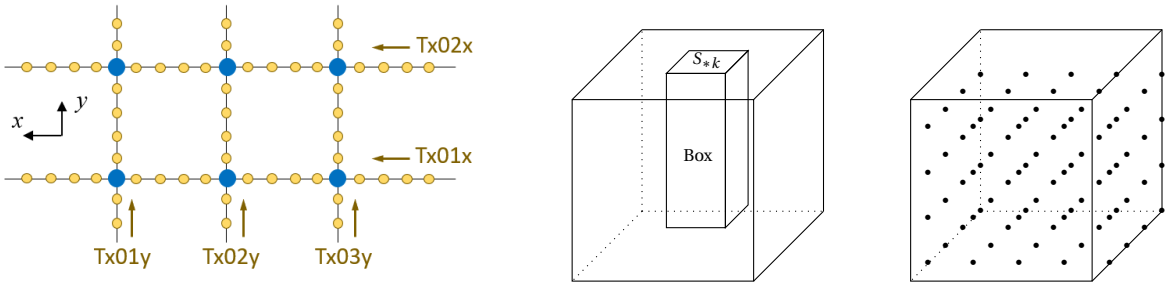
4.2 Background and motivations

4.2.1 Finite difference electromagnetic modeling

The frequency-domain Maxwell equations in the conductive earth in a presence of a current source $\boldsymbol{J}$ can be approximated as follows:

$$\nabla \times \nabla \times \boldsymbol{E} - i\omega\mu\bar{\boldsymbol{\sigma}}\boldsymbol{E} = i\omega\mu\boldsymbol{J}, \tag{4.1}$$

where $\boldsymbol{E}$ is the electric field, $\bar{\boldsymbol{\sigma}}$ is the conductivity tensor, μ is the magnetic permeability, and ω is the frequency. Using finite differences on a grid of size $N = N_x \times N_y \times N_z$ corresponding to the discretization of the physical domain, the electric field has three components E_x, E_y, E_z at each grid point and can be approximated by solving linear systems of the form $\boldsymbol{M}x = s$, where



(a) Sources layout. (b) Boxing. (c) Sampling.

Figure 4.1: (a) Schematic example of a CSEM survey with locations of receivers (blue circles) and transmitter positions (brown circles). (b) Entries of the final solution are those inside the "box" centered around the corresponding source S_{*k}. (c) Entries of the final solution correspond to a subset of nodal points uniformly distributed in the domain.

$\boldsymbol{M}$ is a sparse matrix of order $n = 3N$ and s is a source vector resulting from the right hand-side in Equation (4.1). $\boldsymbol{M}$ can easily be made symmetric. Let us now discuss the properties of RHSs and the solution that follow from the geometry of marine CSEM surveys as well as inversion approaches used to analyze CSEM data.

The CSEM receivers are typically placed at the seafloor in a regular grid with 1–3 km spacing, while the transmitter is towed above the receiver lines. To achieve illumination of subsurface with different transmitter orientations, two orthogonal directions of towlines are often chosen. A schematic picture of such a CSEM survey outline is presented in Figure 4.1a, where receiver locations are indicated with blue circles. Yellow circles along the towlines indicate transmitter locations: it is usually assumed that distinct transmitter positions are spaced by 100 m. Since the transmitter is moving, while seabed receivers are fixed, the number of transmitters n_t is much larger (by 1–2 orders of magnitude) than the number of receivers n_r.

The number of right-hand sides is determined by the inversion algorithm used to analyze CSEM data. The gradient-based (BFGS) scheme [71] is relatively cheap: at each iteration one needs to solve a linear system of equations for source terms placed only at the receiver positions (due to reciprocity). In this chapter we will however focus on the more powerful Gauss-Newton scheme that is expected to soon become the prevailing inversion method [51]. In the Gauss-Newton method, the sources should also be placed at each transmitter position, *i.e.*, the total number of RHSs is becoming much larger since $n_t \gg n_r$. Note also that the transmitter is usually towed within 30–100 m above the seafloor therefore all RHSs (due to both transmitters and receivers) belong to a narrow depth interval near the seafloor – the property we shall utilize later in the chapter.

The right hand sides are usually very sparse since they describe a source term that is localized in space. A point transmitter is usually represented by placing source terms at $2\times2\times2 = 8$ nearest nodes in 3D problems, *i.e.*, a RHS will have only 8 nonzero elements. In marine CSEM a horizontal electric-dipole transmitter is often an extended antenna of ~ 300 m length, rather than a point. In that case, the number of nonzero elements will be slightly larger (e.g. 16 or 24), but this complication will have only minor effect on our results, thus for the sake of simplicity we shall stick to considering point sources with 8 nonzero elements in RHSs.

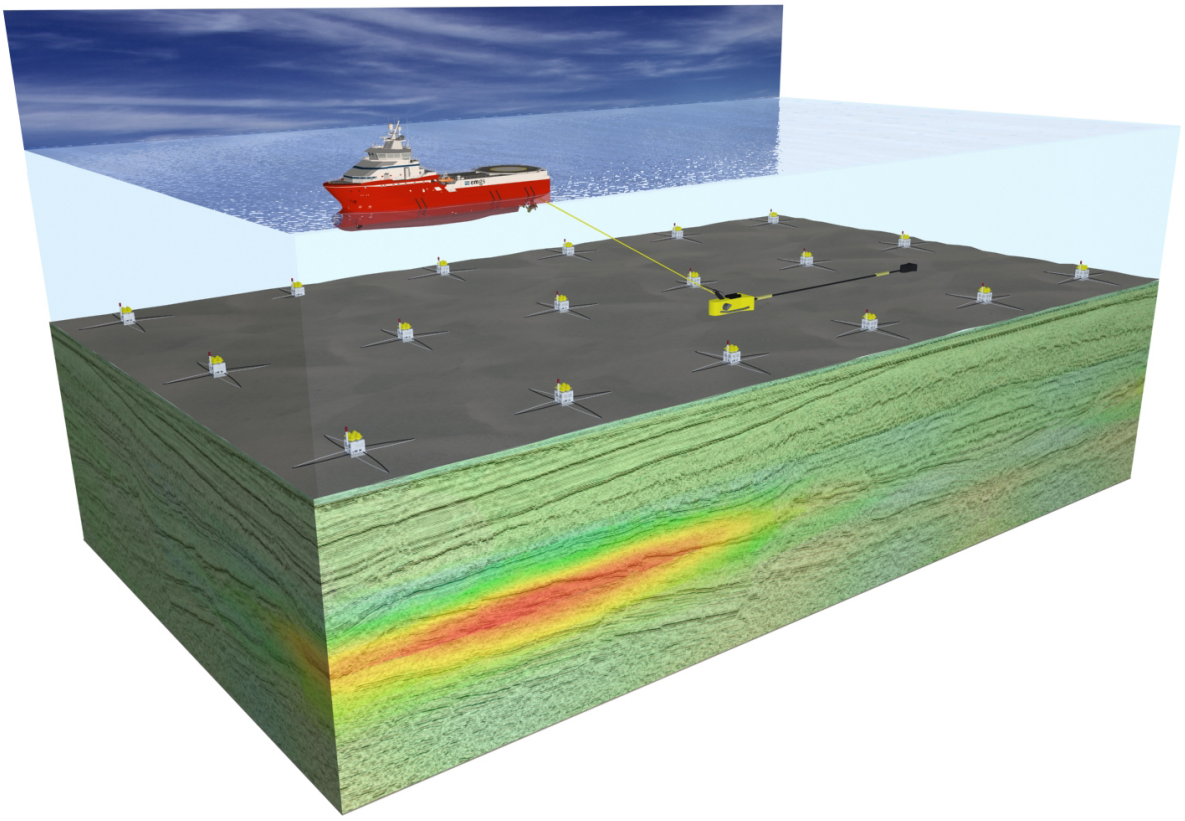


Figure 4.2: Illustration of the acquisition of data in a CSEM study.

The initial ordering of RHSs defining order of the columns in $\boldsymbol{X}$ usually reflects the transmitter trajectory. In our context, the ordering of transmitter positions obeys the following simple rule, see Figure 4.1a. We start with towline Tx01x, and there go over all transmitter positions in the $\vec{x}$ direction, see Figure 4.1a. Then we switch to towline Tx02x and follow the same ordering, and so on until we reach the last $\vec{x}$-directed towline. Then we switch to $\vec{y}$-directed towlines, starting with Tx01y, and for each of them go over all transmitter positions in the direction of $\vec{y}$ axis. As we shall see below, this continuous ordering of RHSs is not optimal for the solver performance, and considerable gains can be achieved by appropriate reorderings. Strictly speaking, in the Gauss-Newton scheme, one also has to handle right-hand sides related to receiver positions. However their number is much smaller since $n_t \gg n_r$, and therefore we have not included them in the analysis below for the sake of simplicity.

Only a subset of entries in the solution $\boldsymbol{X}$ needs to be computed when inverting marine CSEM data. For each column k and source vector S_{*k}, only the entries in a box with a square section and centered around S_{*k} are needed, see Figure 4.1b. The box excludes the top of the domain: the air since its resistivity is known, and also the water layer since water conductivity is usually measured during the CSEM survey. The CSEM data decay with increasing offset between transmitter and receiver and eventually drop below the noise level. We shall assume that the maximum offset for CSEM data is 12.5 km and therefore the lateral extent of the box will be 25 km $\times$ 25 km. All the regions beyond this box, in particular, the perfectly matched layers at the edges, are excluded from the solve phase. Depending on the problem, the box may represent around one half of the whole computational domain.

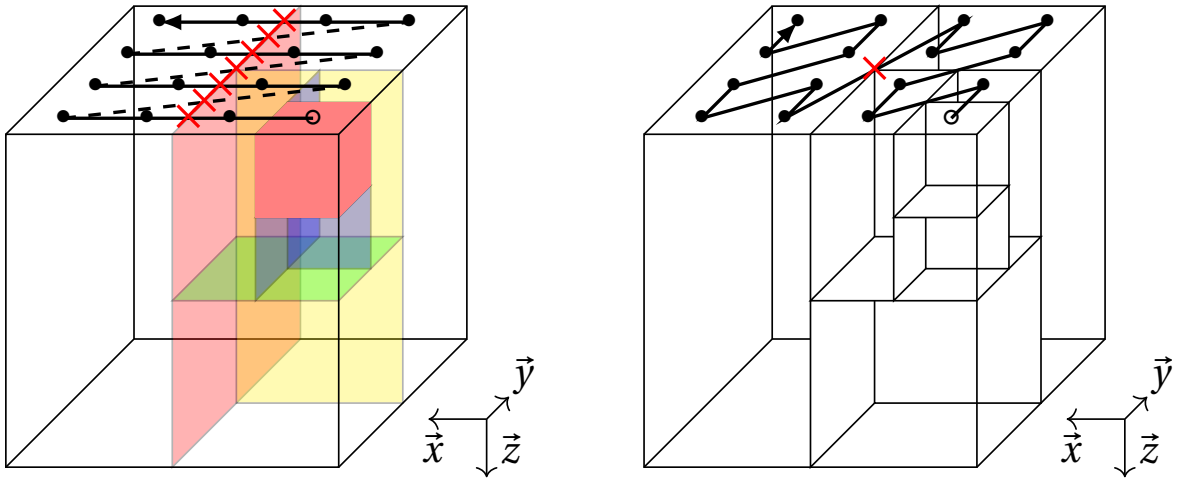
Reducing the number of inversion parameters can make the Gauss-Newton inversion faster. Since the CSEM method resolution decreases with depth, it is common to use fewer inversion parameters in deeper formation layers. As a result, the solution $\boldsymbol{X}$ in some regions may be required for a coarser sampling than the grid used to build the system matrix. In Section 4.6.1, we assume that the solution could be required on a uniformly distributed subset of nodal points, see Fig. 4.1c, where only every 20th or every 100th point is included into the subset.

In the next paragraphs we give some preliminary hints on how sparsity described above can be used to reduce the solver complexity.

4.2.2 Impact of the source structure

In this section we focus on the forward substitution,and on the sparsity in the source vectors $\boldsymbol{S}$. The discussion on exploiting sparsity in the backward substitution relies on similar ideas but is postponed to Section 4.5.

Application of direct solvers to linear EM systems built on finite-difference methods is often illustrated by a hierarchical domain decomposition based on nested dissection [34], see Section 1.2.1 for more details. A separator can be defined as a set of nodal points the removal of which divides the domain (or subdomains) into two balanced and *disjoint* subdomains. In a regular 3D grid such as the one represented in Figure 4.3, the separator shapes are planes with normals in the $\vec{x}, \vec{y}$ or $\vec{z}$ directions. In Figure 4.3a, we represented with (red, yellow, green, ...) colors the successive *top separators* that led to the subdomain corresponding to the red subcube in the top right corner of the domain.



(a) Initial source order.

(b) Optimized source order.

Figure 4.3: A 3D domain with sources (• or ○) and a connecting line representing the transmitter trajectory. A hierarchical domain decomposition with 2D plane-shaped separators is applied. (a) shows the separators leading to the red subcube containing the first source (○). The red crosses show indicates where the transmitter trajectory crosses the main separator. (b) shows a source ordering that minimizes the crossings of top separators.

Since the source term is geometrically localized, all nonzero elements of a source vector usually belong to the same subdomain. The nested dissection creates independence between disjoint subdomains, hence the source contribution during the forward substitution will not affect any other subdomains. In other words, for the computation of $\boldsymbol{Y}$, we have to consider only the given subdomain and *top separators*. In Figure 4.3a, the contribution of the first source is represented by its subdomain (red cube), top separators (colored planes), while all other parts of the domain will stay out of its area of influence. Section 4.3 is dedicated to the exploitation of this feature.

Furthermore, we will show that the initial ordering of sources is not optimal with respect to the operation counts, especially if one aims at processing simultaneously many sources. A simplified example of initial ordering is depicted in Figure 4.3a showing the source locations (symbols) and their connecting line, “transmitter trajectory”. The important message here is that the transmitter trajectory crosses the top separator multiple times. We will show that the optimal ordering will be such that the transmitter trajectory has the smallest possible number of crossings of top separators. The transmitter trajectory displayed in Figure 4.3b will be shown in Section 4.3 to possess most of the properties of the theoretically optimal solution.

4.2.3 Characteristics of the models and computing environment

Our study is based on realistic anisotropic earth resistivity models characteristic for marine CSEM applications. The models are discretized using finite-difference Yee grids with a uniform core and growing cell sizes at the model edges and an air layer on top. Properties of system matrices and right-hand sides resulting from these discretizations are summarized in Table I.

Table I: Characteristics of the systems of equations $\boldsymbol{MX} = \boldsymbol{S}$. $n = 3N_x \times N_y \times N_z$ is the order of $\boldsymbol{M}$, m is the number of columns of the right-hand side matrix $\boldsymbol{S}$, $D(\boldsymbol{M})$ and $D(\boldsymbol{S})$ are the average numbers of nonzeros per column for $\boldsymbol{M}$ and $\boldsymbol{S}$, respectively. The resolution times for the different phases of a sparse direct solver using 90 MPI processes and 10 threads per MPI process are also reported: T_a for analysis, T_f for factorization, T_s for solve and $T_{\text{total}} = T_a + T_f + T_s$ for the entire resolution.

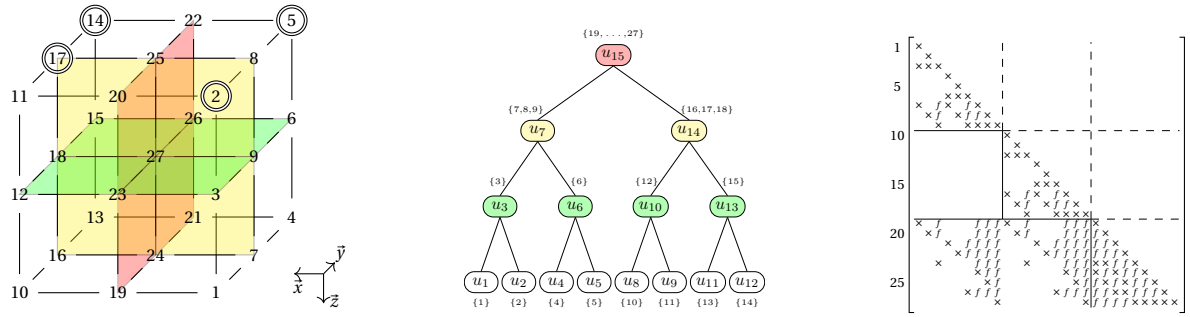
Model	System	Grid shape	Matrix $\boldsymbol{M}(n \times n)$		RHS $\boldsymbol{S}(m \times n)$		Timings in seconds (percentage of total time)			
		$N_x \times N_y \times N_z$	n	$D(\boldsymbol{M})$	m	$D(\boldsymbol{S})$	T_a	T_f	T_s	T_{total}
Shallow water	H3	$114 \times 114 \times 74$	2,885,112	12.9	8000	7.5	10 (1%)	34 (4%)	806 (**95%**)	850
	H17	$214 \times 214 \times 127$	17,448,276	12.9	8000	6	56 (1%)	378 (8%)	4133 (**91%**)	4567
SEAM	S21	$181 \times 160 \times 237$	20,590,560	12.9	12340	9.5	68 (1%)	476 (6%)	7819 (**93%**)	8363
DayBreak	DB30	$230 \times 422 \times 102$	29,700,360	12.9	3914	7.6	106 (2%)	765 (15%)	4246 (**83%**)	5117

The matrices H3, H17 and S21 are described in detail in [60]. The two H-matrices are based on a half-space 1 Ωm model with a 100 m water layer and a pizza-box resistor of 100 Ωm. The H3 matrix is based on a coarser grid, with cell sizes (in the central part) double of those used for the H17 matrix. The S21 matrix is obtained from the SEAM (SEG Advanced Modeling Corporation) Phase 1 resistivity model representative of the Gulf of Mexico: it has a rough bathymetry, hydrocarbon reservoirs and salt bodies. The DB30 matrix is built from a resistivity model corresponding to a CSEM survey "Daybreak" acquired in Alaminos Canyon, Gulf of Mexico [41].

The RHSs are generated by listing all transmitter positions using the ordering indicated in Figure 4.3a. For example, for the SEAM S21 matrix, the survey layout suggested 36 towlines, 40 km long, in one direction, and 29 towlines, 35 km long in the orthogonal direction. The distance between towlines was 1 km. We downsampled the transmitter positions to 200 m spacing, which resulted in in $36 \times 201 + 29 \times 176 = 12340$ RHSs. RHSs for the Daybreak matrix were given by the real survey that included 12 towlines of 60 km length and 2 km apart, and 2 orthogonal towlines of 30 km length, 4 km apart. For each system, the number of right-hand sides m reaches several thousands and their density $D(\boldsymbol{S})$ is below 10 nonzeros per column.

We also report in Table I the analysis, factorization and solve times of the sparse direct solver MUMPS using BLR compression [6] on the CALMIP supercomputer EOS (https://www.calmip.univ-toulouse.fr/), which is a BULLx DLC system composed of 612 computing nodes, each composed of two Intel Ivybridge processors with 10 cores (total 12 240 cores) running at 2.8 GHz, with 64 GBytes of memory per node. As mentioned earlier, the introduction of low-rank approximations has significantly reduced the factorization time [60], and the initial solve time T_s (not using the work presented in this chapter) has become predominant. Note that the solve phase was performed by blocks of size $BLK = 1024$ for H3 and $BLK = 512$ for H17, S21 and DB30. This was mandatory as the memory required to process all right-hand sides in one shot otherwise exceeded the available memory.

In the following subsection, we give some background on the solve phase of sparse direct solvers, before explaining in Section 4.3 how to take advantage of the right-hand side sparsity resulting from the geometrical structure of CSEM applications.



(a) $3 \times 3 \times 3$ regular mesh. (b) Assembly/separators tree. (c) Structure of $\boldsymbol{L}$.

Figure 4.4: (a) A 3D regular mesh based on a 7-point stencil; each node is numbered according to the nested dissection algorithm following a postorder. Each double circles are elementary sources modeling two stacked sources from the CSEM application. (c) Resulting separators or assembly tree; also showing the sets of variables to be eliminated at each node. (c) Corresponding matrix with initial nonzeros ($\times$) in A and fill-in (f) in L.

4.2.4 Solve phase algorithms

We first describe the algorithms used to solve the linear systems $\boldsymbol{LDL^TX = S}$, where $\boldsymbol{M = LDL^T}$, from an algebraic point of view. We also explain how they can be interpreted and correlated to the structural and geometrical properties of CSEM applications.

$\boldsymbol{L}$ is a unit lower triangular sparse matrix of order n whereas $\boldsymbol{S}$ is an $n \times m$ matrix of right-hand sides. As mentioned earlier, the first part of the solve algorithm consists in performing the forward substitution, which can be written as $\boldsymbol{LY = S}$.

We assume that $Y_{*k} \leftarrow S_{*k}$ for each column k so that all our algorithms can be expressed only in terms of modifications of Y_{*k}. The first version of our forward algorithm is a scalar two-loop algorithm limited to nonzero entries in $\boldsymbol{L}$.

$$
\begin{array}{l}
Y_{*k} \leftarrow \boldsymbol{L^{-1}} Y_{*k} \quad \text{(Scalar two-loop algorithm)} \\
\qquad \textbf{for } j = 1, \ldots, n-1 \\
\qquad\qquad \textbf{for } i \geq j+1 \text{ such that } l_{ij} \neq 0 \\
\qquad\qquad\qquad y_{ik} \leftarrow y_{ik} - l_{ij} y_{jk}
\end{array}
\tag{4.2}
$$

The algorithm described in (4.2) exploits the fact that the diagonal of $\boldsymbol{L}$ is the identity, and that $\boldsymbol{L}$ is sparse, *i.e.*, many of the l_{ij} entries are zero. We explain why and how sparsity can be exploited in an efficient way based on the example of Figure 4.4 and will reformulate the algorithm to illustrate it.

Figure 4.4a provides a simplified version of the CSEM application, where we reduced the number of degrees of freedom from 3 to 1 on each nodal point and used a 7-point stencil to represent the mesh. The corresponding matrix is represented in Figure 4.4c. Moreover, for the right-hand side matrix $\boldsymbol{S}$, we consider only eight sources with a single nonzero per source. The sources are placed at nodal points 2, 5, 14 and 17 which belong to the same $\vec{z}$-plane to illustrate

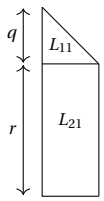


Figure 4.5: Structure of the factor associated to a node from the separator tree.

the fact that the transmitter trajectory along the seafloor is usually quite horizontal. The initial ordering of the sources is assumed to be 2-17-5-14 and then 2-5-17-14, which is similar to the ordering shown in Figure 4.1a and convenient for illustrative purposes. The matrix $\boldsymbol{S}$ has the structure depicted in Figure 4.6a. We will reuse this matrix of sources in Section 4.3.

We focus now on the independence created through the nested dissection process. Initially, the process builds a separator that divides the domain into two disjoint and *independent* subdomains, see Figure 4.4a. It numbers the variables of each subdomain contiguously and the variables of the separator last. This can also be expressed as a root node u_{15} (separator) and two subtrees (subdomains) in the separator tree of Figure 1.5c. The two subtrees characterize the aforementioned independence between the variables of both subdomains which is reflected by the empty square in the structure of $\boldsymbol{L}$ corresponding to rows $[10, 18]$ and columns $[1, 9]$ in Figure 4.4c. From the algorithm corresponding to Equation (4.2), the computation of the components of $\boldsymbol{Y}$ inside each subdomain will then be independent from each other. For $i \in [10, 18]$ and $j \in [1, 9]$ we have $l_{ij} = 0$ so that, for any k, and for any $i_1 \in [1, 9]$ and $i_2 \in [10, 18]$ component $y_{i_1 k}$ does not depend on component $y_{i_2 k}$. The separator tree also characterizes the parallelism of the solve phase. The nested dissection process is reproduced recursively on both subdomains, preserving the mentioned properties.

In the context of the multifrontal method, each node of the separator tree may be represented by a dense matrix called front which is used to compute a part of the $\boldsymbol{L}$ factor, as illustrated in Figure 4.5. Each front is associated with two sets of variables: the q variables of the separator (also called fully-summed variables), which are used to compute entries of the $\boldsymbol{Y}$ or $\boldsymbol{X}$ solutions; and the r off-diagonal variables (or non fully-summed variables), which are used to compute contributions. Data computed at each node will be used by the parent (resp. children) fronts in case of forward (resp. backward) substitution. More precisely, the forward substitution is a bottom-up process which performs, for each front, the two block operations $\boldsymbol{Y}_1 \leftarrow \boldsymbol{L}_{11}^{-1}\boldsymbol{Y}_1$ and $\boldsymbol{Y}_2 \leftarrow \boldsymbol{Y}_2 - \boldsymbol{L}_{21}\boldsymbol{Y}_1$, whereas the backward substitution is a top-down process which performs, for each front, the block operations $\boldsymbol{X}_1 \leftarrow \boldsymbol{X}_1 - \boldsymbol{L}_{21}^T\boldsymbol{X}_2$ and $\boldsymbol{X}_1 \leftarrow \boldsymbol{L}_{11}^{-T}\boldsymbol{X}_1$. Here, $\boldsymbol{Y}_1, \boldsymbol{Y}_2, \boldsymbol{X}_1, \boldsymbol{X}_2$ are partial matrices of $\boldsymbol{Y}$ and $\boldsymbol{X}$ containing only the variables corresponding to the $q + r$ rows of the front. As follows from the properties of the separator tree, if two fronts belong to different subtrees, the computations at those fronts can be done in parallel.

We will use the notation $u(j)$ to denote the node of the separator tree containing variable j. We have, for example, $u(14) = u_{12}$, or $u(25) = u_{15}$. Thanks to the compact representation of the structure of the factors at each node (see Figure 4.5), operations reported in Equation (4.2) can be performed on dense matrices and the condition "$l_{ij} \neq 0$" is replaced by "i in the structure of the factors at node $u(j)$", as will be indicated in Equation (4.3). Furthermore, in the context of sparse RHSs y_{*k} might remain equal to zero so that Equation (4.2)

should perform the update of y_{ik} only for nonzero entries y_{jk}, Equation (4.2) becomes:

$$\begin{array}{l} \boldsymbol{Y} \leftarrow \boldsymbol{L^{-1}Y} \quad \text{(Nodal algorithm)} \\ \quad \textbf{for } j = 1, \ldots, n-1 \\ \quad\quad \textbf{for } i \text{ in the structure of the factors at node } u(j), i > j \\ \quad\quad\quad \textbf{if } (y_{jk} \neq 0) \;\; y_{ik} \leftarrow y_{ik} - l_{ij} y_{jk} \end{array} \tag{4.3}$$

Note that in our example, the numbering of the node identifiers $u_1, u_2, \ldots, u_{15}$ obeys the following *postordering* rule: all nodes in any subtree are numbered consecutively and precede the number for the root of the subtree. Moreover, any subtree of T rooted at node u (which we denote as $T[u]$), corresponds to a subdomain created through the nested dissection. For example, $T[u_7]$ corresponds to the subdomain on the right of the first separator (u_{15}) and is composed of the variables {1,2,3,4,5,6,4,7,8,9}. Note also that the resolution of the diagonal system $\boldsymbol{DZ} = \boldsymbol{Y}$ can be performed in-between the forward and the backward substitutions or can be combined with one of these phases by computing each component as $z_{ik} = y_{ik}/d_{ii}$.

4.3 Exploiting RHS sparsity to reduce the amount of computations

In the previous section, we have shown that thanks to the knowledge of the frontal matrix structure at each node of the separator tree, testing nonzero entries in the rows i of column L_{*j} was not needed and the two-loop algorithm (Equation 4.2) could be simplified. Furthermore, since $\boldsymbol{S}$ is sparse, some elements y_{jk} in Equation (4.3) may remain equal to zero. Similarly one would like to avoid systematic testing for $y_{jk} \neq 0$ at each update of the nodal algorithm (Equation 4.3). For efficiency, we also want to perform operations on a block of columns and thus to *a priori* identify blocks of columns sharing the same structure and allowing simultaneous operations.

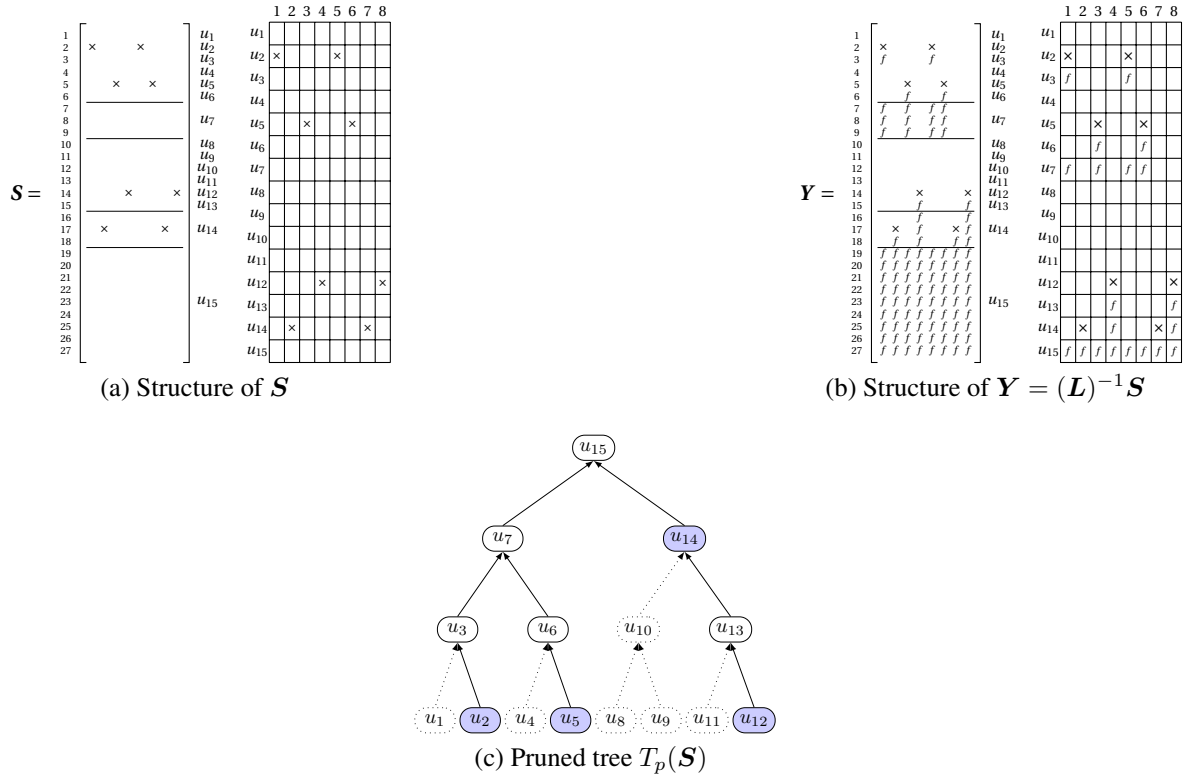
We describe the graph structure that needs to be introduced and exploited to avoid systematic testing and relate this structure to the geometric properties of the CSEM application.

We focus in this section on the forward substitution ($\boldsymbol{LY} = \boldsymbol{S}$), but the same ideas can be applied to the backward substitution ($\boldsymbol{L}^T\boldsymbol{X} = \boldsymbol{Z}$) when a partial solution is needed, as will be discussed in Section 4.5.

Making efficient use of the sparsity in the RHS matrix is a three-step process:

- firstly, exploit sparsity within the columns of the sources (*i.e.*, detecting empty rows), referred to as *vertical* sparsity
- secondly, exploit sparsity within the rows (*i.e.*, detecting nonzero blocks within non-empty rows) of $\boldsymbol{Y}$, referred to as *horizontal* sparsity
- finally find a suitable column *ordering* to improve the performance of horizontal sparsity.

We describe in the following each step and also relate it to geometric/applicative interpretations. We refer the reader to Section 1.3.2 to get of more "technical" point of view of the exploitation of sparsity.



(a) Structure of $\boldsymbol{S}$

(b) Structure of $\boldsymbol{Y} = (\boldsymbol{L})^{-1}\boldsymbol{S}$

(c) Pruned tree $T_p(\boldsymbol{S})$

Figure 4.6: (a) Structure of right-hand side matrix $\boldsymbol{S}$ (associated to sources described in Figure 4.4) and its node representation, nonzero are represented with $\times$. (b) Structure of $\boldsymbol{Y}$ after the forward substitution, fill in are represented with f (left and right). (c) Corresponding pruned tree, pruned nodes are dotted, active nodes are plain.

Vertical sparsity

Exploiting vertical sparsity in the forward substitution makes use of the properties proved in [35] and was also formulated in terms of paths using the tree structure in [36]. We explain in this section how it relates to our application and to the position of the sources in the CSEM application.

In Figure 4.3a of Section 4.2.1, we illustrated the fact that the contribution of each source is limited to its local subdomain (the red subcube) and to the top separators. For a given source, or equivalently a column S_{*k} of the RHS matrix $\boldsymbol{S}$, the aforementioned contribution corresponds to nonzero components of Y_{*k}. This gives the intuition of the importance of the separator tree. In the following, we explain how the separator tree can be used to efficiently characterize nonzero entries in $\boldsymbol{Y}$ so that the loop on index j can be set up *a priori* without need for any checks to restrict the subset of indices.

Figure 4.6a represents a matrix $\boldsymbol{S}$ composed of 8 right-hand sides associated with 8 sources (Figure 4.4) placed at nodes 2, 17, 5, 14, 2, 5, 17, 14, in this precise order. In our simplified model and for the sake of clarity, we considered a single nonzero element per source. To simplify the figure, we also provide a compact representation of matrix $\boldsymbol{S}$ where each row

corresponds to the set of variables from a node of the tree. Finally, each node whose set of variables includes at least one nonzero from matrix $\boldsymbol{S}$, *i.e.*, each node $u(i)$ for which there exists a column index k such that $s_{ik} \neq 0$, will be called an *active node*[2]. Active nodes have been filled in the separator tree represented in Figure 4.6c corresponding to our simplified model.

As the solve algorithm proceeds, new nonzero entries (so called fill-in) with respect to the original entries of $\boldsymbol{S}$ appear in $\boldsymbol{Y}$. Given the initial nonzero structure of $\boldsymbol{S}$, [35] and [36] showed that it is possible to predict the nonzero structure of $\boldsymbol{Y}$. In our context, [36, Theorem 2.1] can be translated into:

Theorem 4.1. *When solving $\boldsymbol{L}Y_{*k} = S_{*k}$, the structure of the vector Y_{*k} is given by the union of the variables in nodes on paths in the tree T from the set of* active nodes *of S_{*k} up to the root.*

As a consequence, a component y_{ik} will be different from zero if and only if $s_{ik} \neq 0$ or there exists an $s_{jk} \neq 0$ such that either $u(j) = u(i)$ or $u(j)$ is a descendant of $u(i)$ in T. Equation (4.3) is only computed for variables j belonging to such nodes. This was referred to as *tree pruning* in Section 1.3.2. As an example, take S_{*1} from Figure 4.6a with $s_{2,1} \neq 0$ and $u(2) = u_2$. Then every nonzero component of Y_{*1} belongs to nodes that are on the path from u_2 to u_{15}. This algebraic perspective translates into the geometrical interpretation illustrated in Figure 4.3a.

Furthermore, to enhance the performance of the solve phase, computation should be done on multiple columns at the same time. In doing so, one can benefit from the use of BLAS 3 operations [26] that can often reach the peak performance of a processor. Theorem 4.1 is then performed for the union of the set of *active nodes* of each column, see Section 4.2.3. The tree resulting from the pruning process is called the pruned tree and, if we consider the whole matrix $\boldsymbol{S}$ as one block, it is noted $T_p(\boldsymbol{S})$ and shown in Figure 4.6c. Therefore, Equation (4.3) becomes:

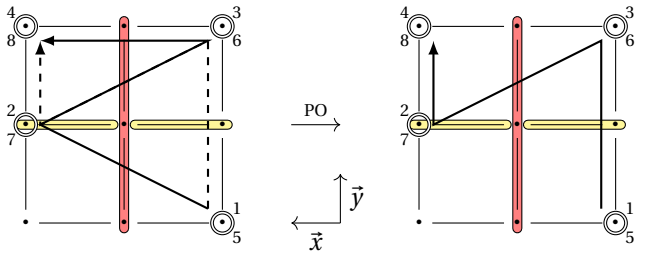
$$\begin{array}{l} \boldsymbol{Y} \leftarrow \boldsymbol{L}^{-1}\boldsymbol{Y} \quad \text{(Pruned tree nodal algorithm)} \\ \qquad \textbf{for } Y_{j*} \neq 0, 1 \leq j \leq n-1 \quad \text{(i.e., variable } j \text{ belongs to the pruned tree)} \\ \qquad\quad \textbf{for } i \text{ in the structure of the factors at node } u(j), i > j \\ \qquad\qquad Y_{i*} \leftarrow Y_{i*} - l_{ij}Y_{j*} \end{array} \tag{4.4}$$

where Y_{j*} is the j-th row of $\boldsymbol{Y}$ and $Y_{j*} \neq 0$ means that *at least one* of its component is different from 0. In Figure 4.6c, each pruned node in $T_p(\boldsymbol{S})$ corresponds to an empty row in $\boldsymbol{Y}$, this is why sparsity is exploited *vertically*.

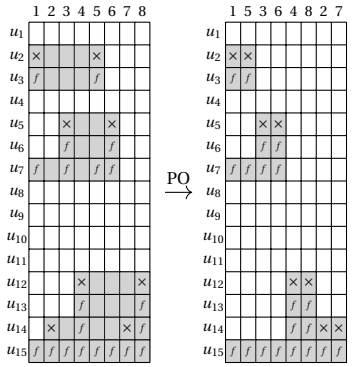
Horizontal sparsity and column ordering

Equation (4.4) assumes that all columns are processed at each node of the pruned tree. However, sources do not all have the same structure and thus it is possible to further exploit sparsity by reducing the number of columns on which Equation (4.4) is applied. This will be referred to

[2] This would have been defined as $V_S = \cup_{1 \leq k \leq m} V_{S_{*k}}$ in Section 1.3.2



(a) Initial column ordering (1-2-3-4-5-6-7-8, left, corresponding to nodal points 2-7-5-14-2-5-17-14 in Figure 4.4a) and Postorder (1-5-3-7-2-6-4-8, right).



(b) Initial (left) and Postorder (right) orderings.

Figure 4.7: (a) A 2D section from Figure 4.4a located on the plane containing the source positions. The numbering of nodal points has been omitted for clarity. (b) Illustration of horizontal sparsity with node intervals and influence of the ordering of columns on sparsity.

as *horizontal sparsity*. To do so, we explain how and why the notion of node intervals combined with column ordering introduced for computing selective entries of the inverse of a matrix [11] can be effective in our context to reduce the number of operations. We illustrate these aspects in Figure 4.7 on the same simplified example with 8 sources as in the previous section.

The subset of columns of $\boldsymbol{S}$ that possesses at least one nonzero element at a given node $u \in T$ is called the set of *active columns* at node u, see Section 1.3.2 for more details on the definition. For example for node u_5, corresponding to row u_5 in Figure 4.7b (left), there are only two active columns: 3 and 6. Ideally, one would like to operate only on these two columns which would require complex data reorganizations or the computation of the columns one after the other. This would not be efficient since processing simultaneously a block of columns is faster that processing them one by one. What can be done at no extra reorganization cost is to consider a *subinterval of columns* including the first and the last indices of the active columns at that node. The intervals are thus defined for each node of the separator tree. In Equation (4.4) and for the computation of component Y_{i*}, $*$ is replaced by the interval defined for node $u(i)$. For example, the active columns for node u_5 are 3 and 6, thus the interval at node u_5 includes only four columns: 3, 4, 5 and 6, rather than all 8 columns. With intervals, we reduce computation on columns and thus exploit *horizontal* sparsity.

Clearly, the size of the intervals is influenced by the ordering of the columns. The idea is to order successively columns with close initial nonzero structure or, equivalently, to limit the crossing of top separators as was mentioned in relation to Figure 4.3b. The permutation used in this study is called a *postorder* and is built as follows: let the tree be numbered following a postordering, as in Section 4.2.4. For a column k of $\boldsymbol{S}$, we define $u_{rep}(k)$ as the node among the active nodes for column S_{*k} ($\{u(i), s_{ik} \neq 0\}$) that appears first in the postordering of the tree. We have for example $u_{rep}(1) = u_2$ and $u_{rep}(2) = u_{14}$ in Figure 4.6a (and 4.7b, left), and call $u_{rep}(k)$ the *representative* node of column k.

Now $\boldsymbol{S}$ is said to be *postordered* if and only if: $\forall k_1, k_2, 1 \leq k_1 < k_2 \leq m$, $u_{rep}(k_1)$ appears before (or is identical to) $u_{rep}(k_2)$ in the postordering. In other words, the order of the columns S_{*k} and the postordering of their representative nodes $u_{rep}(k)$ are compatible.

In Section 4.2.1, we said that the postorder trajectory should minimize the number of crossing of top separators. We see in Figure 4.7b that the postorder trajectory can also be interpreted in terms of nonzero structure of $\boldsymbol{S}$ and $\boldsymbol{Y}$. Namely, it corresponds to order the columns of $\boldsymbol{S}$ such that two successive columns have a close nonzero structure (in $\boldsymbol{Y}$). Indeed, the initial transmitter trajectory, see Figure 4.7a, first implies a "superposition" of non-successive sources in $\boldsymbol{S}$. Thus, columns with close positions were not initially close in the column ordering. This resulted in large interval sizes, see rows u_7 and u_3 in Figure 4.7b (left). The postorder heuristic addresses this problem, see Figure 4.7b (right) and is optimal in this case since the gray areas do not include zero entries anymore. Note that for the purpose of our illustration we have considered sources with only one nonzero entry and that in this case the postorder heuristic has been shown to be optimal [11]. On our CSEM application, each source has more than one entry per column, thus possibly more than one active node, hence the definition of u_{rep}.

Tree pruning, node intervals and a suitable column ordering exploit the sparsity of the application to reduce the amount of computations in the solve phase. However, this is done at the expense of reduced parallelism. The next section shows how to still exploit parallelism efficiently in the solve phase, even when dealing with sparse right-hand sides.

4.4 Improving the parallel aspects of the solve algorithms

In this section, we first explain the differences between the factorization and the solve phase in terms of parallel algorithms. We then show how the blocks of sparse RHS can be defined and how the solve phase can be adapted to improve the available parallelism.

4.4.1 Differences between the factorization and the solve algorithms

The factorization and the solve algorithms have different properties in terms of parallelism and load balancing. Although in pratice we apply a BLR factorization, we consider in this section Full-Rank metrics because they are the basis for the mapping and scheduling algorithms we use [12]. We recall that, on the one hand, *tree parallelism* is represented by the separator tree (two nodes from different subtrees can be processed independently, as explained in Section 4.2.4). On the other hand, large nodes of the separator tree offer an additional potential for parallelism. This will be referred to as *node parallelism*. Moreover, on a dense matrix of order n, the complexity in terms of number of operations of the factorization and solve phases, respectively $O(n^3)$ and $O(n^2)$, is quite different. With nested dissection, the size of the separators and thus the size of the frontal matrices increases as we get closer to the top of the tree. Computation is thus concentrated near the top of the tree and this is more true for the factorization than for the solve. This effect is illustrated in Figure 4.8, which compares the distribution

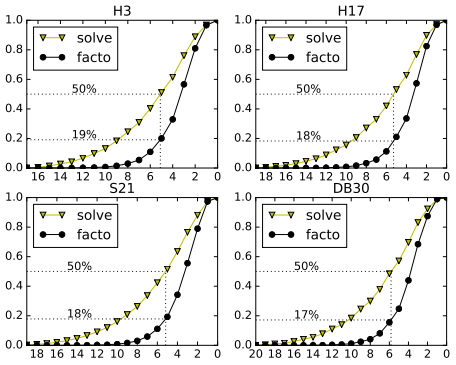
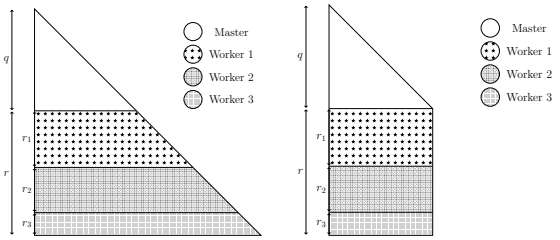


Figure 4.8: Normalized operation count of the solve and factorization phases as a function of the depth in the separator tree, with $depth(root) = 0$. The nested dissection ordering has been used.



(a) Front during factorization (triangular frontal matrix).

(b) Front during solve (factors).

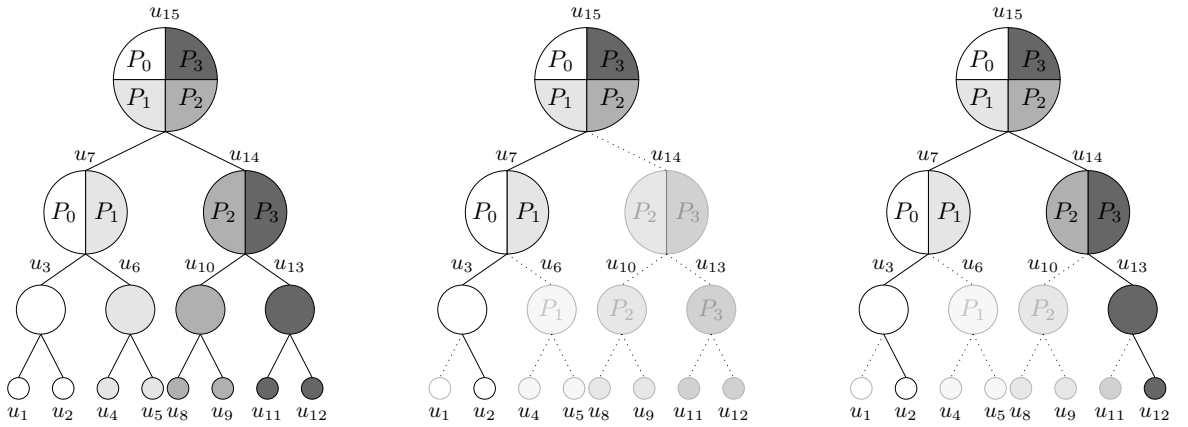
Figure 4.9: Mapping of the rows of a front to balance the workload of the factorization between processors.

of computations in the separator tree for both phases. At a depth where 50% of the computation is completed for the solve phase, only 20% is completed for the factorization. This indicates that the solve phase will benefit more from tree parallelism than the factorization phase.

Tree pruning limits the number of branches of the separator tree that can be computed independently, so that tree parallelism and tree pruning introduced to exploit RHS sparsity are thus conflicting objectives. A classical approach to balance the workload between the processors during the factorization is to use a *proportional mapping* [53]. Starting from the root node to which all processors are allocated and going down the tree, at each level of the tree the list of processors of the current node is partitioned between its sons according to the load of each of the subtrees rooted at each son. This is referred to as strict proportional mapping and is illustrated in Figure 4.10a. It can be adapted or relaxed in order to allow for dynamic mapping and scheduling decisions, or to reduce memory usage [12]. If the whole set $\boldsymbol{S}$ is considered, and if the set of sources is separated by the top level separators, then the width of the pruned tree $T_p(\boldsymbol{S})$ may be large enough to cover most of the tree and almost fully benefit from tree parallelism. However, because of the memory constraints mentioned in Section 4.2.3, the solve phase is generally processed by blocks of limited size (BLK), potentially reducing the width and parallelism of the pruned tree. In this context, it is important to decide how these blocks can be created to minimize the loss of tree parallelism introduced by the use of RHS sparsity.

Furthermore, at each node of the separator tree, a symmetric frontal matrix is partially factored. For frontal matrices associated with large separators near the top of the tree, the proportional mapping assigns several processors and the workload of the factorization is divided between a master and several workers. This is illustrated in Figure 4.9 where q is the size of the separator and r is the number of rows to be updated. At each node the first r variables are factorized so that the number of operations is

$$W^f(q,r) = W^f_m(q) + W^f_w(q,r) = \frac{1}{6}(2q^3 + 3q^2 - 5q) + qr(q + r + 1), \tag{4.5}$$



(a) T with 4 processors. (b) $T_p(\boldsymbol{S}_1)$ with $\boldsymbol{S}_1 = (S_{*1}, S_{*5})$. (c) $T_p(\boldsymbol{S}_2)$ with $\boldsymbol{S}_2 = (S_{*1}, S_{*4})$.

Figure 4.10: Proportional mapping and comparison of tree coverage between three blocks of right-hand sides based on the example from Figure 4.4. (a) Dense RHS (all the tree is covered); (b) set of two close sources; (c) set of two distant sources.

where $W_m^f(q) = \frac{1}{6}(2q^3 + 3q^2 - 5q)$ corresponds to the operation count on the master processor and $W_w^f(q, r) = qr(q + r + 1)$ to the operation count on the workers. As shown in Figure 4.9a, more rows must then be associated to the processors that appear first in the front. Note that we also want to adjust relative size of q and r to balance the workload between the master and each worker by splitting nodes of the separator tree [9, 14].

In CSEM applications, where the solve phase becomes predominant, we need to drive our algorithms with metrics related to the solution phase, as described in the following subsection.

4.4.2 Improving algorithms for the solve phase

Because of memory constraints, we have seen that the columns of $\boldsymbol{S}$ need be processed by blocks. In the scheme presented in Section 4.2.3, the columns of matrix $\boldsymbol{S}$ are processed using the initial ordering and by blocks of size BLK. In that case, only a subpart of the domain is covered by the partial transmitter trajectory within each block. Large subdomains, or subtrees, will be pruned from the separator tree, limiting the number of operations but also leading to a significant loss of tree parallelism. Figures 4.10b and 4.10c illustrate this property with two sets $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$, containing close and distant sources, respectively. To improve the tree coverage, one can select non-contiguous columns from matrix $\boldsymbol{S}$. They will better cover the physical domain because the transmitter follows a regular trajectory. Furthermore, since we also want the efficiency of BLAS-3 kernels, we propose to select each block of BLK columns such that it is composed of a set of sub-blocks of constant size equally distributed onto the transmitter trajectory. To do so for a given sub-block size whose size is related to the BLAS-3 performance kernel, one can compute a constant gap to provide a good trajectory coverage and thus a good separator tree coverage. We mention that within each block, we still apply a postordering permutation to maximize the effect of horizontal sparsity.

Moreover, node parallelism has an important role in the performance of the solve phase. As shown before, Figure 4.9b illustrates the distribution of data among processors when the work is balanced for the factorization. Considering the solve, the number of operations performed during the forward substitution (or the backward substitution) at each node is:

$$W^{fwd}(q,r) = W_m^{fwd}(q) + W_w^{fwd}(q,r) = q(q-1) + 2rq, \tag{4.6}$$

where $W_m^{fwd}(q) = q(q-1)$ and $W_w^{fwd} = 2rq$. As a consequence, to balance W_w^{fwd} among workers, data need to be reorganized so that all workers possess the same number of rows. For that, we also switched off the dynamic schedulers from the factorization that lead to irregular partitions with a dynamic choice of the workers at each node and, as a consequence, used a strict static proportional mapping of the processors in the tree. This strategy will be referred to as S-ROWDISTRIB().

Furthermore, to balance the work between the master and each worker, we aim at splitting nodes in the separator tree so that $W_m^{fwd}(q) \approx W_w^{fwd}(q, r_i)$, where r_i, the number of rows of each worker is equal to r divided by the number of workers. This strategy is referred to as S-SPLIT().

In summary, the optimizations above aim at favoring tree and node parallelism during the solve phase when dealing with either sparse or dense RHSs. Concerning the optimizations specific to sparse RHSs, we focused on the forward substitution but they also apply to the backward substitution, as discussed in Section 4.5. In Section 4.6, we also experiment with another optimization of the solve phase regarding locality of data access and multithreading, which was motivated by the need to process large blocks of columns to improve tree coverage and tree parallelism.

4.5 Exploiting sparsity during the backward substitution

During the backward phase ($\boldsymbol{L}^T\boldsymbol{X} = \boldsymbol{Z}$), the nonzero structure of $\boldsymbol{Z}$ results from the operations performed during the forward substitution since $\boldsymbol{DZ} = \boldsymbol{Y}$ with $\boldsymbol{D}$ diagonal. When the matrix $\boldsymbol{M}$ is irreducible, which is the case in the CSEM application, the variables of $\boldsymbol{Y}$ belonging to the root node of the separator tree will be nonzero, independently of the position of the sources. The backward substitution processes the $\boldsymbol{L}$ matrix in a backward way which translates into a top-down traversal of the separator tree. As a result, all the nodes in the tree are reached and need to be processed during the backward phase. This translates back into the fact that $\boldsymbol{Z}$ is dense and that sparsity in the sources $\boldsymbol{S}$ does not result in any reduction of the number of backward-phase operations.

However, the sparsity of the solution can result from the properties of the physical problem, typically when only part of the solution has value and needs to be computed. As explained in Section 4.2.1 and illustrated in Figure 4.1, boxing and/or regular sampling can be used to select a subset of valuable entries. Why and how sparsity can be exploited during the backward substitution is explained below.

Given a valuable entry x_{ik} in the solution, the computations that contribute to updating x_{ik} can be characterized, similarly to the forward phase, by Theorem 4.1. Only nodes in the path

from the root node to node $u(i)$ need be considered to compute x_{ik}. In other words and from a geometric perspective, if one assumes that i belongs to the filled subdomain of Figure 4.3a then the variables involved in the computation of x_{ik} will correspond to the colored separators and part of the filled-subdomain. As a consequence, the process of tree pruning introduced in Section 4.3 can be applied to the backward substitution [57, Lemma 2.2]. The exploitation of horizontal sparsity also remains unchanged and the computation of a suitable column ordering inside each block follows the same rule, namely, "columns with similar structure of valuable entries should be kept close in the column ordering".

First, sources close to each other have high overlapping boxes of valuable entries and also have close to each other representative nodes in the separator tree. Second, in the case of regular sampling of the entries in the solution, we have no locality property to preserve since all the space is regularly covered by the solution. Thus, the representative nodes of the sources can also be used for the boxes and therefore, the column ordering chosen during the forward phase can be used during the backward phase and the choice of the blocks from Section 4.4.2 can be identical.

The valuable entries in each column of $\boldsymbol{X}$ are thus defined as a sampled set of variables in a box around the corresponding source location. It should be noted that this numerical sparsification of $\boldsymbol{X}$ is quite moderate compared to extreme sparsification of the sources $\boldsymbol{S}$. Thus, $\boldsymbol{X}$ is a much denser matrix with less geometrically localized nonzero variables than $\boldsymbol{S}$. As illustrated in the next section, one should thus expect less impact of exploiting sparsity during the backward step than during the forward step.

4.6 Performance analysis in a parallel context

We analyze the impact exploiting RHS sparsity and of using parallel solve-aware strategies on the performance of the solve phase in a parallel environment. We also present global resolution times showing that the relative weight of the solution phase has significantly decreased compared to the initial results from Table I.

A perfect nested dissection ordering has been chosen for all the following results, which were obtained using the `MUMPS` solver [9, 12]. We list T_f, T_s, T_{fwd}, T_{root}, T_{bwd} – the times to perform the factorization, solve, forward substitution, solve on the root node (through ScaLAPACK [21]) and backward substitution, respectively.

4.6.1 Exploiting sparsity

We first study vertical and horizontal sparsity, and show the impact of the choice of the columns and of their order on parallelism. We consider the forward substitution in Section 4.6.1 and the backward substitution in Section 4.6.1.

The forward substitution

We first report in Table II the performance in terms of number of operations and time for solution of the proposed algorithms (vertical, horizontal sparsity and postorder reordering of

Table II: Number of operations (OPS$\times 10^{10}$) for the forward elimination for 1024 contiguous RHSs of system H3.

OPS ($\times 10^{10}$)		Dense	Vertical sparsity	Horiz. sparsity	Horiz. sparsity and Postorder
Contiguous	first 1024 ($\boldsymbol{S}_1$)	951	270	225	149
	last 1024 ($\boldsymbol{S'}_1$)	951	232	190	151

Table III: Times (seconds) for the forward elimination for 1024 contiguous RHSs of system H3, with 32 MPI and 1 thread per MPI.

T_{fwd} (s)		Dense	Vertical sparsity	Horiz. sparsity	Horiz. sparsity and Postorder
Contiguous	first 1024 ($\boldsymbol{S}_1$)	170	112	85	60
	last 1024 ($\boldsymbol{S'}_1$)	170	114	87	69

Table IV: Number of operations (OPS$\times 10^{10}$) for the forward elimination for 1024 non-contiguous RHSs of system H3.

OPS ($\times 10^{10}$)		Dense	Vertical sparsity	Horiz. sparsity	Horiz. sparsity and Postorder
Non-Contiguous	16 cols, gap 109 ($\boldsymbol{S}_2$)	951	428	306	125
	32 cols, gap 218 ($\boldsymbol{S'}_2$)	951	427	302	132

Table V: Times (seconds) for the forward elimination for 1024 non-contiguous RHSs of system H3, with 32 MPI and 1 thread per MPI.

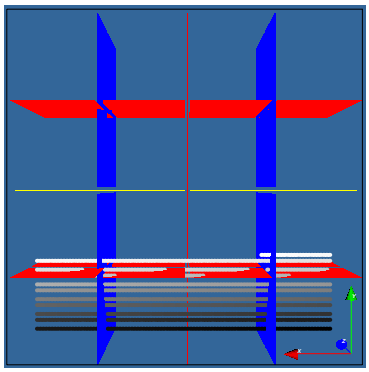
T_{fwd} (s)		Dense	Vertical sparsity	Horiz. sparsity	Horiz. sparsity and Postorder
Non-Contiguous	16 cols, gap 109 ($\boldsymbol{S}_2$)	170	111	98	30
	32 cols, gap 218 ($\boldsymbol{S'}_2$)	169	107	96	31

RHS columns) on the system H3 on 1024 contiguous columns of RHS. As expected from the theory (compare columns 3 and 4 of Table II) using vertical sparsity significantly reduces the number of operations with respect to processing dense RHS. Adding horizontal sparsity and postordering of the columns further reduces the number of operations. However, as shown in Table III, this operation reduction is not fully converted into time reduction and most notably for the operations reduction due to vertical sparsity.

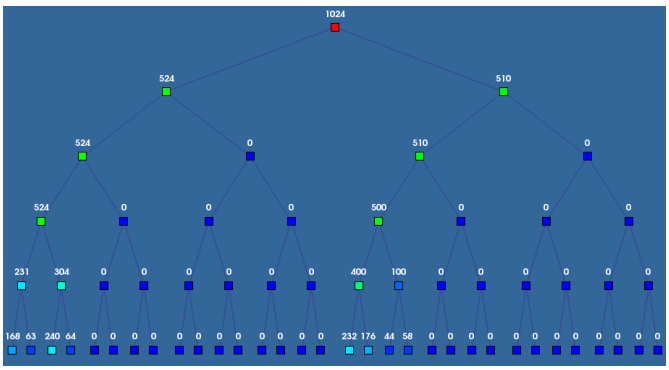
Let us illustrate with Figure 4.11 the conflicting objectives of vertical sparsity and performance and explain how to address this issue. With the initial order of the columns in $\boldsymbol{S}$, the 1024 first ones (set $\boldsymbol{S}_1$) are located at the low y part of the horizontal plane containing the sources, see Figure 4.11a, and appear in the order described in Figure 4.3a. From an algebraic point of view, the effect of tree pruning (see Figure 4.11b) is that $T_p(\boldsymbol{S}_1)$ is quite narrow (many branches have no active columns). On the contrary, choosing 1024 non-contiguous columns spreads the RHS in the domain, as illustrated in Figure 4.11c with the set $\boldsymbol{S}_2$ consisting of sets of 16 columns in $\boldsymbol{S}$ separated by 109 columns. The first consequence of such a distribution is a wider pruned tree. Indeed, comparing $T_p(\boldsymbol{S}_1)$ and $T_p(\boldsymbol{S}_2)$ from Figures 4.11b and 4.11d, we see that the top subdomains are not filled with sources from $\boldsymbol{S}_1$ so that the pruned tree $T_p(\boldsymbol{S}_1)$ contains less nodes than $T_p(\boldsymbol{S}_2)$.

As a consequence, when only vertical sparsity is used, one can expect a larger number of operations with $\boldsymbol{S}_2$ than with $\boldsymbol{S}_1$ (compare columns "Vertical sparsity" of Tables II and IV). However it is also interesting to observe that the exploitation of horizontal sparsity *combined* with a postordering of the columns of RHS recovers this increase in the number of operations (compare last columns of Tables II and IV). We discuss/explain it in the following.

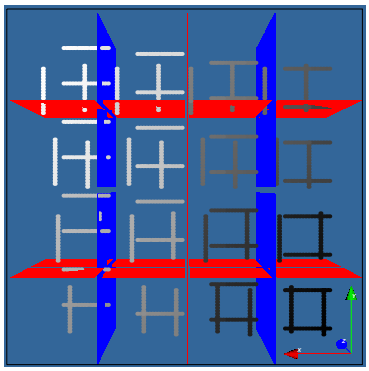
In Section 4.3, we explained that sources closely located in the geometrical domain needed to be close in the column ordering to reduce the operation count. The color gradient from Figure 4.11c illustrates the effect of postordering the columns: sources that belong to the same subdomain become close with respect to the column ordering. This property explains why the efficiency of horizontal sparsity is increased even more when a postordering of the columns is applied. Indeed, for set $\boldsymbol{S}_1$, we have a 17% reduction in the number of operations with horizon-



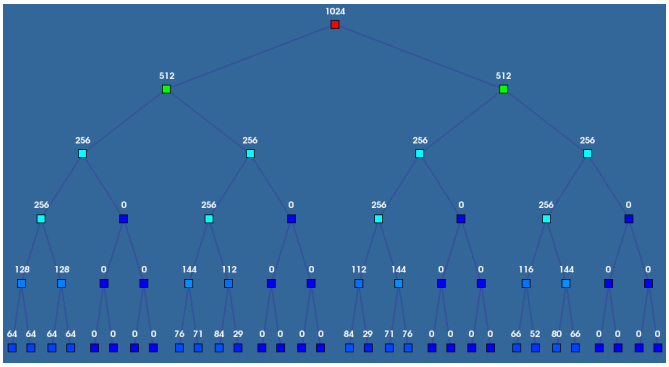
(a) Separators and set of 1024 contiguous sources ($\boldsymbol{S}_1$).



(b) $T_p(\boldsymbol{S}_1)$ with active columns.



(c) Separators and set of 1024 non contiguous sources ($\boldsymbol{S}_2$).



(d) $T_p(\boldsymbol{S}_2)$ with active columns.

Figure 4.11: Geometrical and algebraic RHS distribution for two subsets of 1024 columns for system $H3$. (a) and (c) represent top views of the geometrical domain for, respectively, 1024 contiguous RHS in natural order and 1024 non-contiguous RHS sets of 16 columns with a gap of 109 columns permuted using a postorder. The color gradient indicates the index of the column (source) in the (possibly reordered) set of RHS columns. (b) and (d) are respectively the corresponding top 6 layers of the separator tree with, for each node, the number of active columns, as defined in Section 4.3.

Table VI: Number of operations (OPS$\times 10^{10}$) for the backward substitution for 1024 non-contiguous RHS of system H3. Except for column "Dense", only a subset of the solution is computed with coarse solution vector sampling applied.

OPS ($\times 10^{10}$)	Samp.	Dense	Vertical sparsity	Horiz. sparsity	Horiz. sparsity and Postorder
16 cols, gap 109	20	951	876	702	636
(S_2)	100	951	876	701	635
32 cols, gap 218	20	951	876	703	626
(S'_2)	100	951	876	702	625

Table VII: Estimated times (seconds) for the backward substitution for 1024 non-contiguous RHS of system H3, with 32 MPI and 1 thread per MPI. Except for column "Dense", only a subset of the solution is computed with coarse solution vector sampling applied.

T_{bwd} (s)	Samp.	Dense	Vertical sparsity	Horiz. sparsity	Horiz. sparsity and Postorder
16 cols, gap 109	20	169	160	141	127
(S_2)	100	170	160	140	131
32 cols, gap 218	20	169	160	137	127
(S'_2)	100	170	160	141	127

tal sparsity, reaching 45% when postordering is applied. With non-contiguous columns, the operation reduction due to horizontal sparsity and postordering reaches 71% (see Table IV).Thus, even in the case of non-contiguous columns, the number of operations is comparable (even slightly smaller) than with contiguous columns (compare last columns of Tables II and IV). Non-contiguous columns also expose the forward step to more parallelism and thus the time for the forward step with contiguous columns (already divided by a factor of three with respect to dense RHS processing, compare last and third columns of Table III) is further divided by a factor of two (see last column in Table V).

The backward substitution

We analyze in Tables VI and VII the impact of computing only a subset of the solution on the operation count and on the execution time, respectively. In these tables, the postorder used is identical to the one from the forward substitution, avoiding any RHS permutation between the forward and the backward phases. We only show results with non-contiguous sets of columns which enable, as in the forward phase, to better exploit parallelism. To measure the time and the number of operations, we exploit the fact that performing the backward substitution ($\boldsymbol{L}^T\boldsymbol{X} = \boldsymbol{Z}$) while computing only a subset of the entries of the solution $\boldsymbol{X}$ is equivalent in terms of operations, computation kernels used and parallelism, to performing the forward substitution $\boldsymbol{LY} = \boldsymbol{X}$ exploiting the sparsity of the right-hand side $\boldsymbol{X}$. All options to exploit sparsity developed for the forward phase could then be used to analyze the potential of exploiting sparsity during the backward step.

The density of the solution is such that one should expect much more moderate gains due to sparsity compared to the forward phase. Indeed, the model of Figure 4.1b and the RHS distribution of Figure 4.11c show that most of the domain is concerned by the solve phase and thus most of the nonzero structure will be concerned. Hence, the ratio of operations between the dense and the vertical strategies is close to one. We also observe that the performance using coarse solution vector sampling is not affected when the number of degrees of freedom on which the solution is computed decreases from 1 over 20 to 1 over 100 (from sampling 20 to 100). Although coarse solution vector sampling can be useful to reduce the volume of data corresponding to the solution, it indeed only affects vertical and horizontal sparsity in the

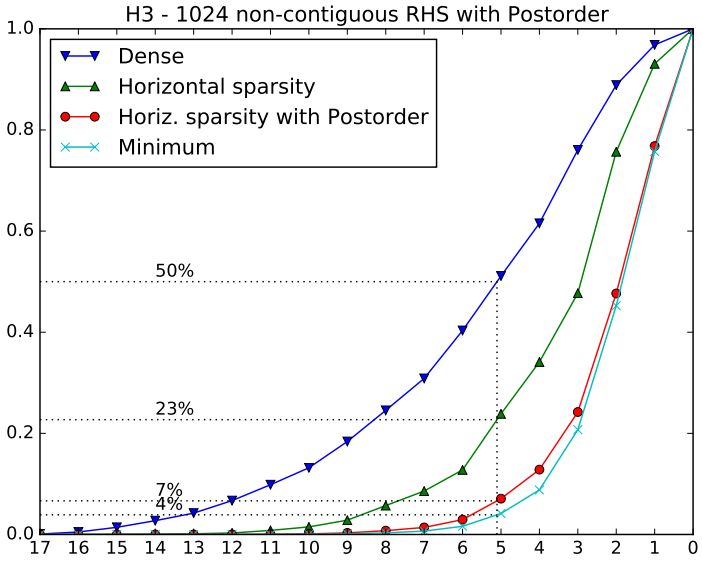


Figure 4.12: Normalized accumulation of operations by levels in the separator tree for the forward substitution with 1024 non-contiguous RHS permuted in postorder ($\boldsymbol{S}_2$) for H3. `Minimum` corresponds to the minimal number of operations, if all operations on zeros were avoided.

lowest levels of the tree, which constitute only a minor part of the computation. Overall, the exploitation of sparsity in the computed entries of $\boldsymbol{X}$ brings an approximate 1.35x gain on the time for the backward substitution. These gains are significant and will be included in the global results of Section 4.6.3.

4.6.2 Improvement through load-balancing and multithreading

Load-balancing is performed at several levels. In this section we first evaluate the impact of sparsity on tree parallelism, and then show the importance of pushing forward node parallelism through balanced workload between workers and between the master and the workers. We mention that sparsity is not exploited for the backward substitution in this section and we report the actual times obtained for the dense backward substitution.

In Figure 4.12 we analyze the relation between tree parallelism and exploitation of right-hand side sparsity during the forward substitution. Note that for the dense RHS case we have shown in Figure 4.8 that the solve phase offers a greater potential for exploiting tree parallelism than the factorization phase. We see in Figure 4.12 that at depth five, when 50% of the computation is performed with dense RHS, only 23% of the computation is performed using horizontal sparsity and only 7% when all optimizations are used. This translates into an important loss of tree parallelism that confirms the increasing relative weight of node parallelism. Table VIII gathers results for the different strategies introduced in Section 4.4.2.

Table VIII: Effect of different mapping strategies on the time for the factorization and solve phases for H3, on EOS, with 1024 RHS (set $\boldsymbol{S}_2$), using 32 MPI processes and 1 thread per MPI process.

Mapping strategy	Factorization time	Forward sparsity	Solve T_s	Solve $T_{fwd}+T_{root}+T_{bwd}$
Standard MUMPS solver	203	No	346	170+9+167
		Yes	206	30+9+167
with S-ROWDISTRIB()	203	No	306	153+9+144
		Yes	174	24+9+144
with S-ROWDISTRIB() and S-SPLIT()	197	No	295	147+9+139
		Yes	167	19+9+139

Table IX: Effect of locality and multithreading optimizations (THREADOPT) on T_s (seconds) to process the RHS columns $\boldsymbol{S}_2$ of system H3. T_s is reported as a a function of the block size BLK (standard or large) and of the number of threads (1 or 10) per MPI process, for 32 MPI processes. Sparsity is exploited during the forward elimination phase only.

# Threads	THREADOPT()	T_f	T_s BLK =16	T_s BLK =1024
1	Off	197	272	167
	On	197	271	136
10	Off	48	106	79
	On	48	87	28

We first observe that, thanks to the efficient use of sparsity during the forward step, the solve phase is dominated by the backward step. However, balancing the workload between workers through equal distribution of rows (S-ROWDISTRIB() strategy in Table VIII), and balancing the workload between master and workers (S-SPLIT() strategy) significantly improves the performance of the solve phase. The forward substitution time with sparse RHS decreases from 30 seconds down to 19 seconds, showing a significantly larger relative gain than the one obtained during the (dense) backward substitution, from 167 down to 139 seconds. This is coherent with the observation reported in Figure 4.12 that node parallelism is more critical when sparsity is exploited.

We now consider the performance of the solve phase in an hybrid MPI-OpenMP environment, where multiple threads are used within each MPI process. In general, sparse direct solvers are used on a limited number of right-hand sides, and processing several of them together (e.g., 16 or 32) leads to better arithmetic intensity and performance thanks to the use of BLAS 3 operations at each node of the separator tree, for which one can rely on multithreaded BLAS libraries. However, CSEM applications have a much larger number of sparse right-hand sides and larger blocks of right-hand sides (ideally all of them if memory was not an issue) are needed to cover the tree and benefit from sufficient tree parallelism (see Section 4.6.1). In this context, and especially in a multithreaded environment, efficient data manipulations at each node of the tree and data locality become critical to efficiently exploit the caches and the memory bandwidth of the processors. We have thus worked on improving locality and on mul-

Table X: Time (seconds) of the analysis, factorization and solve phases on 90 MPI $\times$ 10 threads **with** and **without** all the improvements described in the study. Dense timings of the backward substitution have been divided by 1.35, see Section 4.6.1. The numbers in parenthesis indicate the percentage of T_{total}.

Statistics with improvements				
	H3	H17	S21	DB30
T_a	10 (4%)	56 (3%)	68 (2%)	106 (8%)
T_f	31 (11%)	380 (24%)	434 (16%)	510 (37%)
T_s	233 (85%)	1163 (73%)	2284 (82%)	755 (55%)
T_{fwd}	73 (27%)	289 (18%)	759 (27%)	184 (13%)
T_{root}	14 (5%)	190 (12%)	326 (12%)	80 (6%)
T_{bwd}	146 (53%)	684 (43%)	1199 (43%)	491 (36%)
T_{total}	274	1599	2786	1371
Statistics without improvements (see details in Table I)				
T_{total}	850	4567	8363	5117

tithreading memory-bound operations in both the forward and backward solve phases: arrange nested loops to match the storage of right-hand sides and intermediate solutions, introduce new OpenMP directives, improve data locality and suppress intermediate storage whenever possible.

Table IX reports the impact of these improvements on locality and multithreading (noted THREADOPT) on the solve time for the system H3 with the 1024 non-contiguous RHS corresponding to the set $\boldsymbol{S}_2$ used in Sections 4.6.1 and 4.6.1. The block size (BLK) defines the number of right-hand sides treated in one shot. We see that with a block size of 16, the improvement due to better data locality is nonexistent with one thread and relatively limited with 10 threads. However, with a block size of 1024 (*i.e.*, when all RHS of S_2 are processed in one shot), the impact of these optimizations motivated by the CSEM sparse RHS context become very large, as T_s decreases from 79 to 28 seconds.

We end the study with results on several test matrices that combine all techniques introduced previously.

4.6.3 Global resolution times

We now summarize the new results obtained on the set of systems presented in Table I, and show that the work described in this chapter has a large impact on the global resolution times. We also relate the results to previous work [60] which compared the direct approach to an iterative one.

The experimental environment is the one described in Section 4.2.3. Runs are performed on the EOS machine on 90 MPI $\times$10 threads, hence a total of 900 cores, and the solve phase is

Table XI: Extrapolation of the total resolution time (seconds) for S21 on 900 cores of the EOS machine.

Number of RHS	BLR solver ($\epsilon = 10^{-7}$)				Iterative solver
	T_a	T_f	T_s	T_{total}	
968	68	**434**	**170**	**672**	803
3784	68	**434**	**670**	**1172**	3141

performed using a blocking parameter BLK equal to 1024 columns for system H3 and to 512 for the larger systems H17, S21, and DB30. Compared to Table I, each block now consists of non-contiguous columns in order to encourage tree parallelism, and the columns within each block are postordered. The root node of the separator tree uses ScaLAPACK for both the factorization and the solve phases (as in Table I).

We report in Table X the detailed times for the solve phase, as well as the time for the analysis and factorization phases when all the improvements described in this chapter are applied. To take into account the results of Section 4.6.1 showing that sparsity can be efficiently exploited during the backward substitution, we have also divided the times for the dense backward substitution by 1.35.

Whereas the solve time represented between 83% and 95% of the complete resolution time in Table I, its weight now only represents between 55% and 85% of the resolution time. The solve time has indeed been divided by a factor between 3.4 (for H3) and 5.7 (for DB30). We note that the factorization times have slightly varied between Tables I and X. Although not expected, the modified mapping described in Section 4.4.1 also improves T_f. Overall, the time for the entire resolution has been divided by a factor between 2.8 (for H17) and 3.7 (for DB30).

Finally, we would like to compare the performance of our improved direct solver with an iterative multigrid solver. For that purpose we shall revisit results of our previous work [60, table 5] where both solvers were tested on the S21 matrix with two sets of sparse RHSs with sizes 968 and 3784. The conclusion of the previous work was that the iterative solver is always better because of slow solve phase of the direct solver that required around one second per RHS. After the improvements described in the present chapter the conclusions change dramatically. As we see from Table XI, for the moderate number of right-hand sides (< 1000), the two solvers show similar performance. However for several thousands of RHSs, which is typical for Gauss-Newton iterations, the direct solver demonstrates a superior speed.

4.7 Concluding remarks

We have shown that the performance of direct solvers for large-scale 3D EM problems is critical and we have explained why the structure of the problem can be used to significantly improve the performance. The improvements are twofold: first, we reduced the computational load through the exploitation of sparsity in RHS and solution, second we highlighted certain properties of the solve phase to drive parallel algorithms for modern parallel architectures.

On the one hand, thanks to the sparse structure of the EM sources we have been able to restrict computation during the forward substitution of the solve phase. For a matrix with 3

million unknowns and thousands of RHSs, the resulting gains in the operation count for forward substitution is a factor of $\sim$7.5x leading to a run-time reduction by a factor of $\sim$6.7x. These numbers have been achieved by exploiting both horizontal and vertical sparsity and by reordering the columns of $\boldsymbol{S}$. We have also exploited the sparsity of the solution. Indeed, in marine CSEM applications only the entries that belong to a box with a square section and centered around the source position and excluding water and air needs be computed. We have shown that exploiting this sparsity provides an opportunity to reduce also the time of the backward step. The results should be applicable to linear systems arising in other physical problems on finite-difference or finite-element grids as long as the sources are localized in space and leads to very sparse RHS.

On the other hand, since the solve phase is the most critical step for CSEM application (due to the very large number of right-hand sides) we have shown that this gives scope to design algorithms that will be driven by solve phase metrics to better balance work and data in a parallel environment. Combined to improved multithreading while exploiting large numbers of RHS, it provides an additional time reduction for the forward substitution and backward substitutions.

In the end, with all the improvements included, the overall time reduction for the solve phase is between a 3.4x and 5.7x factor for the tested CSEM problems and makes direct methods now competitive against iterative methods.

Finally, we have shown that combining the improvements on the solve phase with the use of Block Low-Rank (BLR) approximation to speed up the factorization phase makes competitive an approach based on a direct solver at least for problem with numerous RHSs occurring e.g. in the Gauss-Newton inversion. Moreover, since the solve phase is still often the most computationally intensive part of direct solver, one interesting perspective and future work would be to exploit the Block Low-Rank (BLR) format of the factors also during the solve phase. This would further decrease the number of operations during the solve phase and would also reduce memory footprint during factorization, to solve larger problems.

Chapter 5

Conclusion

In this concluding chapter, we first briefly summarize the contributions in the area of sparse systems of linear equations with *multiple sparse* right-hand sides reported in this manuscript. In Chapter 1, the necessary background to analyze the solve phase, its complexity and properties in the context of multiple sparse right-hand sides is provided. Two applications in seismic and electromagnetism modeling for which the solution phase is the most time consuming will be used to illustrate our discussions. Chapter 2 assesses the asymptotic behavior of the solve phase in the context of both sparse right-hand sides and low-rank approximation. How to improve the existing algorithms is described in Chapter 3 where an original approach leading toward an optimal solution is introduced. Finally Chapter 4 revisits our graph related algorithm to explain and justify them in terms of properties of the application. Algorithms to improve the performance in a parallel environment on a real applications is also provided.

Since Chapters 3 and 4 were developed in parallel, our application related chapter, Chapter 4 did not benefit from the new algorithms presented in Chapter 3. We show in Section 5.2 preliminary results to illustrate the gains obtained by exploiting, in a parallel environment, the algorithm presented in Chapter 3 on our two applications. Gains as introduced in the in Chapter 2 are also reported. Section 5.3 describes few perspectives for future work and closes this concluding chapter.

5.1 Contributions

The field of sparse RHS is well known [11, 36, 61]. The objective of Chapter 1 is to introduce some concepts needed in the other chapters. In particular, it provides the formalism needed for Chapter 3: we characterize sparsity and define the so-called *vertical* and *horizontal* sparsity as well as notions such as active nodes/column that are intensively used in Chapter 3. We also illustrate in this chapter the importance of choosing a good permutation of the RHS columns and introduce the formalism needed for Chapter 3.

This thesis is built around three main contributions (three chapters in the manuscript) briefly summarized in the following.

Solve complexity with multiple sparse RHS [JS1]. We emphasize that, in this context of a very large number of sparse RHS, the solve phase can become the bottleneck of the complete numerical simulation. In this chapter we investigate the asymptotic complexity of the problem taking also into account the fact the factor matrices can be represented with a low-rank format. We prove that although exploiting the sparsity of the RHS does not change the asymptotic complexity of the forward substitution this is not the case in the context of low-rank representation of the factors.

First, the use of low-rank approximation techniques brings the solution phase down to a (nearly) linear $\mathcal{O}(n)$ asymptotic complexity. Second, the sparsity of B can be exploited to reduce the cost of the forward substitution. In Chapter 2, we investigate the asymptotic gain on the complexity of the forward substitution achieved by exploiting the RHS sparsity. In particular, we study on 2D and 3D regular problems the asymptotic complexity both for traditional full-rank unstructured solvers and for the case when low-rank approximation is exploited. A significant asymptotic improvement is observed in the latter case, possibly as large as $\mathcal{O}(n^{1/2})$. We confirm the theoretical results first on regular Poisson and Helmholtz problems and second on a set of matrices coming from real-life applications. We mention that the result could be extended to the whole solve phase when only part of the solution is requested, as it is the case in Chapter 4.

As a side result, this complexity study also provides a measure of the available parallelism of the solve phase in the dense RHS case for which a comparison with the factorization shows interesting properties that should be taken into account for the design of algorithms.

Improvement of the exploitation of RHS sparsity [W2, J1]. In this chapter, we focus on the extension of current algorithms to exploit sparsity of the RHS. Based on a geometrical intuition built upon the nested dissection ordering, we first propose a more efficient and generic approach to permute RHS and to reduce the cost of the forward substitution. A second contribution is the description of a blocking algorithm that further decreases this cost by adequately choosing groups of RHS that can be processed together. Although both algorithms are motivated by geometrical observations, they are designed with an algebraic approach, giving a general scope to this work. Notions of node optimality and RHS independence were introduced and formalized, together with theoretical properties to provide a global insight and to support the proposed algorithms.

The experiments confirm the effectiveness of the proposed approach, so-called flat tree permutation with an average reduction of 13% in the number of operations. The so-called blocking algorithm further reduces the number operations. Based on a greedy approach, the blocking algorithm tries to limit the number of groups needed to reach complexity at a given distance from the optimal solution. In a sequential environment and on our real applications, we compare the performance the proposed blocking approach with regular blocking strategies and show the superiority of our blocking approach.

Application to a real-life application and solve-oriented algorithms [JS2, W1]. Controlled-Source ElectroMagnetism (CSEM) is a method of choice for oil and gaz exploration. In this context, the inversion of electromagnetic (EM) data for large-scale geophysical

applications often requires the solution of sparse systems of linear equations with a *large* number of *sparse* right-hand sides each corresponding to a source/transmitter position of the application. Sparse direct solvers are very attractive for these problems, especially when combined with low-rank approximations which significantly reduce the complexity and the cost of the factorization.

We show in Chapter 4 that exploiting the sparsity of both the RHS and the solution nicely impacts the performance of the solve phase. We explain why and how the algebraic properties and tools introduced in Chapter 1 can be used to accelerate computation in the CSEM context.

The first objective of this chapter is to propose another point of view on RHS sparsity that is more comprehensible for a reader who is not a specialist of sparse direct methods and/or graph theory. The second objective is to adapt the parallel algorithms that were designed for the factorization to solve-oriented algorithms and to describe performance optimizations particularly relevant for the case of a very large number of right-hand sides such as in the CSEM application. While Chapters 2 and 3 were targeting the number of operations, this chapter is more concerned with performance and memory issues in a parallel environment. This context motivates the improvement of algorithms to better preserve the parallelism of the solve phase. We show that both the operation count and the elapsed time for the solution phase can be very significantly reduced. The total time of CSEM simulation can be divided by approximately a factor of 3 on all linear systems from our set (from 3 to 30 million unknowns with 4 to 12 thousands RHSs).

5.2 Performance and improvements

In this section, we propose to combine the algorithms developed in Chapters 3 and 4 giving thus an overview of the possibilities offered by a combination of the main contributions described in previous chapters.

The experimental context is that of Chapter 4, *i.e.*, the nested dissection ordering is applied and we consider a limited memory that implies to select a subset of columns from matrix B. As explained in Section 4.2.3, B is processed by blocks of $BLK = 1024$ or $BLK = 512$ columns.

As explained in Chapter 4, to preserve tree parallelism, one needs to select RHS that are sufficiently distributed over the physical domain. This was obtained by selecting non-contiguous columns from matrix B. We name it here the *interleave mechanism*. The success of such an approach thus also relies on the properties of the initial ordering of the columns of the RHS. In Chapter 4, the interleaving mechanism relies on the following properties. Each block of BLK columns is composed of a set of fixed size sub-blocks of contiguous columns equally distributed onto the set of columns of B. Doing so we expect locality within the sub-blocks (efficient memory access) and a good coverage of the total space (good potential for parallelism). Postordering withing each BLK block can then be applied to further improve the data locality and the performance of the algorithms as illustrated in Chapter 4. Our simple interleave mechanism is sensitive to the original ordering of the columns of B.

In the following we compare the use of an algebraic approach based on the flat tree permutation described in Chapter 3 with the natural ordering provided by the application. As explained in Chapter 3, the flat tree permutation groups the RHS depending their position with respect

to the elimination tree so that we can expect an equilibrated distribution of the RHS over the physical domain after the use of the interleave mecanism.

To illustrate our discussion, we introduce the following strategies to permute the RHS:

- `Inter_OFF+PO`: B is split into blocks of size BLK columns. A permutation based on a Post-Ordering of the elimination tree, $\sigma_{\texttt{PO}}$, is then applied within each block;
- `Inter_ON+PO`: the blocks of BLK columns are built *with* the interleave mechanism. Then, a permutation based on a Post-Ordering of the elimination tree, $\sigma_{\texttt{PO}}$, is applied within each block;
- `FT+Inter_ON`: flat tree permutation, $\sigma_{\texttt{FT}}$, is applied first on the columns of B. The blocks of BLK columns are then built *with* the interleave mechanism.

Table I: Comparison of number of operations ($\times 10^{12}$) of the forward substitution in FR and BLR with different interleaving strategies. Root node not included.

EMGS	H3		H17		S3		S21	
OPS	FR	BLR	FR	BLR	FR	BLR	FR	BLR
`Inter_OFF+PO`	8.8	3.4	84.2	6.7	16.5	6.7	179.7	30.2
`Inter_ON+PO`	10.7	4.2	101.9	10.2	21.2	8.3	208.9	35.0
`FT+Inter_ON`	8.0	3.1	72.9	3.9	15.9	6.5	159.8	26.7

SEISCOPE	5Hz		7Hz		10Hz	
OPS	FR	BLR	FR	BLR	FR	BLR
`Inter_OFF+PO`	2.7	1.0	9.0	2.6	30.1	7.0
`Inter_ON+PO`	3.0	1.0	10.8	3.0	35.8	8.0
`FT+Inter_ON`	2.5	0.9	7.5	2.2	25.1	5.9

In Tables I and II, we compare both the number of operations and the time for the forward substitution. We also consider both full-rank (FR) and block low-rank (BLR) formats to store the factors.

Let us first analyze the case of FR factors. The application of the flat tree permutation beforehand gives quite interesting results. As expected, when the interleave mechanism is applied first then it increases the number of operations (compare lines `Inter_OFF+PO` with `Inter_ON+PO` in Table I). This increase is motivated by the better parallel behavior (compare lines `Inter_OFF+PO` with `Inter_ON+PO` in Table II). Applying first the flat tree algorithm before the interleave mechanism has positive effects on the number of operations and even more on the time for parallel execution. For example, for the large matrices H17 or 10Hz, the time with `FT+Inter_ON` is divided respectively by a factor 1.6x and 1.7x when compared to the time with `Inter_ON+PO`, whereas, the number of operations is only divided by 1.4x. Thus, on our test problems, the capacity of the interleave mechanism to expose parallelism is improved when the flat tree permutation is applied beforehand. This will have to be confirmed with a more detailed study. It also suggests to compare our `FT+Inter_ON` with the use of a regular post-ordering beforehand.

Table II: Comparison of times (s) of the forward substitution in FR and BLR with different interleaving strategies. 90 MPI $\times 1$ threads. Root node not included.

EMGS	H3		H17		S3		S21	
TIME	FR	BLR	FR	BLR	FR	BLR	FR	BLR
`Inter_OFF+PO`	230	158	1561	715	435	256	3854	1290
`Inter_ON+PO`	166	119	1339	630	404	255	3748	1386
`FT+Inter_ON`	105	76	845	386	281	189	2483	887
SEISCOPE	5Hz		7Hz		10Hz			
TIME	FR	BLR	FR	BLR	FR	BLR		
`Inter_OFF+PO`	26	18	76	45	203	91		
`Inter_ON+PO`	23	16	76	44	186	85		
`FT+Inter_ON`	16	12	40	26	109	54		

One can also see in Table I that the BLR compression of the factors leads to quite an important reduction in the number of operations with respect to full-rank (FR) storage, a reduction that is larger than the average compression of the factors (statistic not provided in the tables). This is due, as already explained in Section 2.4, to the fact that the factor compression is more effective on the top of the elimination tree which is where most of the work is concentrated in the context of sparse RHS. The time reduction of BLR with respect to FR is quite interesting (factor of 2.8x on the large matrix S21) but still quite far from the theoretical flop reduction. With a block low rank format, the matrices of factors is represented as a set of small dense matrices. Small BLAS kernels are thus involved during the forward substitution and algorithmic work is still needed to improve the performance, balance the workload in a distributed-environment context, and better capture the potential resulting from the reduction in the number of operations. Considering now the BLR (Block Low-Rank) columns, Table I and II confirm the good properties of the flat tree permutation observed for the FR case.

We make some final remarks on the evolution of the gain $\mathcal{G}^{ES}$. We recall that $\mathcal{G}^{ES}$ measures the gain resulting from exploiting the sparsity of the RHS and is defined as the ratio of the complexity of processing RHS as dense over the complexity of processing them as sparse (see Chapter 2 for a precise definition). $\mathcal{G}^{ES}$ thus depends on how the RHS are permuted. We compare, in Table III, the gains $\mathcal{G}^{ES}(\sigma_{\texttt{PO}})$ with an `Inter_ON+PO` permutation of the RHS and $\mathcal{G}^{ES}(\sigma_{\texttt{FT}})$ with a `FT+Inter_ON` permutation. Gains with respect to time for the forward substitution are also reported. For gains reported in BLR columns it is assumed that the factors are always stored in BLR format. We thus do not report here the gains of BLR format over FR format. Numbers related to $\mathcal{G}^{ES}(\sigma_{\texttt{PO}})$ are extracted from Tables V and VI of Chapter 2. In Table IV, $\mathcal{G}^{ES}(\sigma_{\texttt{FT}})$ values are computed using values reported in Tables I and II (row `FT+Inter_ON`). Also when factors are stored in BLR format, we observe that $\mathcal{G}^{ES}$ is substantially improved both in terms of operations and times. As mentioned before for the FR case, this is justified by the good properties of the flat tree permutation to both reduce operations and preserve tree parallelism. The conclusions from Chapter 2 in terms of complexity remain however unchanged.

Table III: Gains relative to the number of operations (OPS) of the forward substitution. Root node not included.

OPS	FR		BLR	
	5Hz	10Hz	5Hz	10Hz
$\mathcal{G}^{ES}(\sigma_{\texttt{PO}})$	5.1	5.1	10.5	11.7
$\mathcal{G}^{ES}(\sigma_{\texttt{FT}})$	6.8	8.0	14.0	19.5
	H3	H17	H3	H17
$\mathcal{G}^{ES}(\sigma_{\texttt{PO}})$	6.5	7.6	8.2	40.0
$\mathcal{G}^{ES}(\sigma_{\texttt{FT}})$	8.0	10.7	11.1	103.8
	S3	S21	S3	S21
$\mathcal{G}^{ES}(\sigma_{\texttt{PO}})$	6.2	7.3	4.5	12.4
$\mathcal{G}^{ES}(\sigma_{\texttt{FT}})$	8.3	9.5	5.8	16.2

Table IV: Gains relative to the time of the forward substitution. 90MPI $\times$ 1 thread on EOScomputer. Root node not included.

Times	FR		BLR	
	5Hz	10Hz	5Hz	10Hz
$\mathcal{G}^{ES}(\sigma_{\texttt{PO}})$	1.9	2.3	2.3	2.7
$\mathcal{G}^{ES}(\sigma_{\texttt{FT}})$	3.1	3.5	4.2	4.6
	H3	H17	H3	H17
$\mathcal{G}^{ES}(\sigma_{\texttt{PO}})$	2.3	4.1	2.3	4.9
$\mathcal{G}^{ES}(\sigma_{\texttt{FT}})$	3.6	6.4	3.6	8.0
	S3	S21	S3	S21
$\mathcal{G}^{ES}(\sigma_{\texttt{PO}})$	1.6	1.8	1.7	2.2
$\mathcal{G}^{ES}(\sigma_{\texttt{FT}})$	2.3	2.7	2.3	3.5

5.3 Perspectives

In this concluding section, we want to draw few perspectives or extensions to the work presented in this manuscript. This section divides the discussion into two different areas: the first related to the topic of sparse RHS and how the work presented in this thesis can be further improved, the second discusses more general features on the solve phase.

The previous section already revealed some ideas on how to pursue the work related to the solution phase with sparse RHS. As a first simple proposition, the application of the blocking algorithm has not been integrated in the performance study reported in the previous tables and this should be analyzed. But before, one may want to consider the following approach. We believe that the formalization introduced in Chapter 3 can be used to improve the interleaving mechanism designed to expose parallelism with a block of RHS. To improve tree parallelism, an alternative to our simple interleave mechanism could be to exploit the recursion tree T_{rec} since it formally represents the position of the RHS in the domain. We could then build the groups of BLK columns by choosing RHS in each of the $R[U_i]$ sets at a given depth d. In this case, we ensure to place in each group of BLK RHS, RHS that will be located in every possible part of the domain.

The second class of perspectives could have a larger scope of applications. While the thesis mainly focus on the solve phase with multiple RHS, in many applications the solve phase is performed on one RHS but *many times*. This is for example the case in non steady applications where the solution depends on the solution from the previous time step. Then, the time of the solve can also be critical or predominant. Some of the improvements introduced in Chapter 4 (load-balancing between master and slaves) can be applied, however, many other fields can be explored to adapt the current factorization-oriented algorithms toward solve-based algorithms. For example, theoretical results from Chapter 2 showed that the solve exhibits more tree paral-

lelism than the factorization; this property may be used to drive the design of new algorithms to better exploit this potential. Finally, the use of low-rank approximations has dramatically modified the behavior of the current algorithms in parallel environments (shared and/or distributed memory). Quite some algorithmic efforts have been done to efficiently exploit low-rank structures during the factorization phase and there is much scope to improve the performance of the solve phase in this context.

Bibliography

[1] A. V. Aho, M. R. Garey, and J. D. Ullman. “The transitive reduction of a directed graph.” In: *SIAM Journal on Computing* 1 (1972), pp. 131–137.

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

[3] P. R. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L’Excellent, and C. Weisbecker. “Improving multifrontal methods by means of block low-rank representations.” In: *SIAM Journal on Scientific Computing* 37.3 (2015), A1451–A1474.

[4] P. R. Amestoy, R. Brossier, A. Buttari, J.-Y. L’Excellent, T. Mary, L. Métivier, A. Miniussi, and S. Operto. “Fast 3D frequency-domain full waveform inversion with a parallel Block Low-Rank multifrontal direct solver: application to OBC data from the North Sea.” In: *Geophysics* 81.6 (2016), R363–R383.

[5] P. R. Amestoy, A. Buttari, J.-Y. L’Excellent, and T. Mary. “On the complexity of the Block Low-Rank multifrontal factorization.” In: *SIAM Journal on Scientific Computing* 39.4 (2017), A1710–A1740. DOI: 10.1137/16M1077192.

[6] P. R. Amestoy, A. Buttari, J.-Y. L’Excellent, and T. Mary. “Performance and Scalability of the Block Low-Rank Multifrontal Factorization on Multicore Architectures.” In: (2018). Accepted for publication.

[7] P. R. Amestoy, T. A. Davis, and I. S. Duff. “An approximate minimum degree ordering algorithm.” In: *SIAM Journal on Matrix Analysis and Applications* 17.4 (1996), pp. 886–905.

[8] P. R. Amestoy, I. S. Duff, A. Guermouche, and Tz. Slavova. “Analysis of the Solution Phase of a Parallel Multifrontal Approach.” In: *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.” In: *SIAM Journal on Matrix Analysis and Applications* 23.1 (2001), pp. 15–41.

[10] P. R. Amestoy, I. S. Duff, J.-Y. L’Excellent, Y. Robert, F.-H. Rouet, and B. Uçar. “On computing inverse entries of a sparse matrix in an out-of-core environment.” In: *SIAM Journal on Scientific Computing* 34.4 (2012), A1975–A1999.

[11] P. R. Amestoy, I. S. Duff, J.-Y. L’Excellent, and F.-H. Rouet. “Parallel computation of entries of A^{-1}.” In: *SIAM Journal on Scientific Computing* 37.2 (2015), pp. C268–C284.

[12] P. R. Amestoy, A. Guermouche, J.-Y. L’Excellent, and S. Pralet. “Hybrid scheduling for the parallel solution of linear systems.” In: *Parallel Computing* 32.2 (2006), pp. 136–156.

[13] P. R. Amestoy, J.-Y. L’Excellent, and G. Moreau. *On Exploiting Sparsity of Multiple Right-Hand Sides in Sparse Direct Solvers*. Research report RR-9122. INRIA, 2017.

[14] P. R. Amestoy, J.-Y. L'Excellent, F.-H. Rouet, and W. M. Sid-Lakhdar. "Modeling 1D distributed-memory dense kernels for an asynchronous multifrontal sparse solver." anglais. In: *High Performance Computing for Computational Science, VECPAR 2014 - 11th International Conference, Eugene, Oregon, USA, June 30 - July 3, 2014, Revised Selected Papers*. 2014, pp. 156–169.

[15] P. R. Amestoy and C. Puglisi. "An unsymmetrized multifrontal LU factorization." In: *SIAM Journal on Matrix Analysis and Applications* 24 (2002), pp. 553–569.

[16] P. Amestoy, A. Buttari, J.-Y. L 'excellent, and T. Mary. *Bridging the gap between flat and hierarchical low-rank matrix formats: the multilevel BLR format*. Research Report. University of Manchester, Apr. 2018. URL: https://hal.archives-ouvertes.fr/hal-01774642.

[17] A. Aminfar, S. Ambikasaran, and E. Darve. "A fast block low-rank dense solver with applications to finite-element matrices." In: *Journal of Computational Physics* 304 (2016), pp. 170–188.

[18] D. B. Avdeev. "Three-Dimensional Electromagnetic Modelling and Inversion from Theory to Application." In: *Surveys in Geophysics* 26.6 (Nov. 2005), pp. 767–799. ISSN: 1573-0956. DOI: 10.1007/s10712-005-1836-x. URL: https://doi.org/10.1007/s10712-005-1836-x.

[19] M. Bebendorf. *Hierarchical Matrices: A Means to Efficiently Solve Elliptic Boundary Value Problems*. Vol. 63. Lecture Notes in Computational Science and Engineering (LNCSE). Springer-Verlag, 2008. ISBN: ISBN 978-3-540-77146-3.

[20] M. W. Berry, B. Hendrickson, and P. Raghavan. "Sparse Marix Reordering Schemes for Browsing Hypertext." In: *Lecture notes in applied mathematic* 32 (1996), pp. 99–124.

[21] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM Press, 1997.

[22] S. Börm, L. Grasedyck, and W. Hackbusch. "Introduction to hierarchical matrices with applications." In: *Engineering analysis with boundary elements* 27.5 (2003), pp. 405–422. ISSN: 0955-7997. DOI: 10.1016/S0955-7997(02)00152-2.

[23] R.-U. Börner. "Numerical Modelling in Geo-Electromagnetics: Advances and Challenges." In: *Surveys in Geophysics* 31.2 (Mar. 2010), pp. 225–245. ISSN: 1573-0956. DOI: 10.1007/s10712-009-9087-x. URL: https://doi.org/10.1007/s10712-009-9087-x.

[24] S. Constable. "Ten years of marine CSEM for hydrocarbon exploration." In: *GEOPHYSICS* 75.5 (2010), 75A67–75A81. DOI: 10.1190/1.3483451. URL: https://doi.org/10.1190/1.3483451.

[25] E. Cuthill and J. McKee. "Reducing the bandwidth of sparse symmetric matrices." In: *Proceedings 24th National Conference of the Association for Computing Machinery*. New Jersey: Brandon Press, 1969, pp. 157–172.

[26] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. “Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms.” In: *ACM Transactions on Mathematical Software* 16 (1990), pp. 1–17.

[27] I. S. Duff. “On the number of nonzeros added when Gaussian elimination is performed on sparse random matrices.” In: *mathematics of computation* 28.125 (1974), pp. 219–230.

[28] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices, Second Edition*. London: Oxford University Press, 2017.

[29] I. S. Duff and J. K. Reid. “A note on the work involved in no-fill sparse matrix factorization.” In: *IMA Journal of Numerical Analysis* 18 (1983), pp. 1145–1151.

[30] I. S. Duff and J. K. Reid. “The multifrontal solution of indefinite sparse symmetric linear systems.” In: *ACM Transactions on Mathematical Software* 9 (1983), pp. 302–325.

[31] I. S. Duff and J. K. Reid. “The multifrontal solution of unsymmetric sets of linear systems.” In: *SIAM Journal on Scientific and Statistical Computing* 5 (1984), pp. 633–641.

[32] S. C. Eisenstat and J. W. H. Liu. “The theory of elimination trees for sparse unsymmetric matrices.” In: *SIAM Journal on Matrix Analysis and Applications* 26 (2005), pp. 686–705.

[33] S. Ellingsrud, T. Eidesmo, S. Johansen, M. C. Sinha, L. M. MacGregor, and S. Constable. “Remote sensing of hydrocarbon layers by seabed logging (SBL): Results from a cruise offshore Angola.” In: *The Leading Edge* 21.10 (2002), pp. 972–982. DOI: 10.1190/1.1518433. URL: https://doi.org/10.1190/1.1518433.

[34] J. A. George. “Nested dissection of a regular finite-element mesh.” In: *SIAM Journal on Numerical Analysis* 10.2 (1973), pp. 345–363.

[35] J. R. Gilbert. “Predicting structure in sparse matrix computations.” In: *SIAM Journal on Matrix Analysis and Applications* 15 (1994), pp. 62–79.

[36] J. R. Gilbert and J. W. H. Liu. “Elimination structures for unsymmetric sparse LU factors.” In: *SIAM Journal on Matrix Analysis and Applications* 14 (1993), pp. 334–352.

[37] J. R. Gilbert, E. G. Ng, and B. W. Peyton. “An efficient algorithm to compute row and column counts for sparse cholesky factorization.” In: *SIAM Journal on Scientific Computing* 15 (1994), pp. 1075–1091.

[38] G. H. Golub and C. F. Van Loan. *Matrix Computations. 4th ed.* Baltimore, MD.: Johns Hopkins Press, 2012.

[39] W. Hackbusch. “A sparse matrix arithmetic based on $\mathcal{H}$-matrices. Part I: introduction to $\mathcal{H}$-matrices.” In: *Computing* 62.2 (1999), pp. 89–108. ISSN: 0010-485X. DOI: http://dx.doi.org/10.1007/s006070050015.

[40] P. Hanssen, A. K. Nguyen, L. T. T. Fogelin, H. R. Jensen, M. Skaro, R. Mittet, M. Rosenquist, L. O. Suilleabhain, and P. van der Sman. "The next generation offshore CSEM acquisition system." In: *SEG Technical Program Expanded Abstracts 2017*. 2017, pp. 1194–1198. DOI: 10.1190/segam2017-17725809.1. URL: https://library.seg.org/doi/abs/10.1190/segam2017-%2017725809.1.

[41] M. Hiner, Y. Martinez, and S. Sun. "Delineating salt bodies with 3D CSEM technology." In: *Salt Challenges in Hydrocarbon Exploration, SEG Annual Meeting Post-convention Workshop, New Orleans, 2015*. 2015.

[42] B. M. Irons. "A frontal solution program for finite-element analysis." In: *Int. Journal of Numerical Methods in Engineering* 2 (1970), pp. 5–32.

[43] G. Karypis and V. Kumar. METIS – *A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices – Version 4.0*. University of Minnesota. Sept. 1998.

[44] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. "Basic Linear Algebra Subprograms for Fortran Usage." In: 5 (1979), pp. 308–323.

[45] J. W. H. Liu. "Modification of the Minimum Degree Algorithm by Multiple Elimination." In: *ACM Transactions on Mathematical Software* 11.2 (1985), pp. 141–153.

[46] J. W. H. Liu. "On the storage requirement in the out-of-core multifrontal method for sparse factorization." In: 12 (1986), pp. 127–148.

[47] J. W. H. Liu. "The multifrontal method for sparse matrix solution: Theory and Practice." In: 34 (1992), pp. 82–109.

[48] J. W. H. Liu. "The Role of Elimination Trees in Sparse Factorization." In: *SIAM Journal on Matrix Analysis and Applications* 11 (1990), pp. 134–172.

[49] T. Mary. "Block Low-Rank multifrontal solvers: complexity, performance, and scalability." PhD Thesis. Université de Toulouse, Nov. 2017.

[50] E. G. Ng and P. Raghavan. "Performance of greedy heuristics for sparse Cholesky factorization." In: *SIAM Journal on Matrix Analysis and Applications* 20 (1999), pp. 902–914.

[51] A. K. Nguyen, J. I. Nordskag, T. Wiik, A. K. Bjorke, L. Boman, O. M. Pedersen, J. Ribaudo, and R. Mittet. "Comparing large-scale 3D Gauss-Newton and BFGS CSEM inversions." In: *SEG Technical Program Expanded Abstracts 2016*. 2016, pp. 872–877. DOI: 10.1190/segam2016-13858633.1. URL: https://library.seg.org/doi/abs/10.1190/segam2016-%2013858633.1.

[52] F. Pellegrini. SCOTCH *and* LIBSCOTCH 5.0 *User's guide*. Technical Report. LaBRI, Université Bordeaux I, 2007.

[53] A. Pothen and C. Sun. "A Mapping Algorithm for Parallel Sparse Cholesky Factorization." In: *SIAM Journal on Scientific Computing* 14(5) (1993), pp. 1253–1257.

[54] J. K. Reid and J. A. Scott. "Reducing the Total Bandwidth of a Sparse Unsymmetric Matrix." In: *SIAM Journal on Matrix Analysis and Applications* 28.3 (2006), pp. 805–821.

[55] D. J. Rose. “A Graph-Theoretic Study of the Numerical Solution of Sparse Positive Definite Systems of Linear Equations.” In: *Graph Theory and Computing*. Ed. by R. C. Read. New York: Academic Press, 1972.

[56] D. J. Rose, R. E. Tarjan, and G. S. Lueker. “Algorithmic aspects of vertex elimination on graphs.” In: *SIAM Journal on Computing* 5.2 (1976), pp. 266–283.

[57] F.-H. Rouet. “Memory and performance issues in parallel multifrontal factorizations and triangular solutions with sparse right-hand sides.” PhD Thesis. Institut National Polytechnique de Toulouse, Oct. 2012.

[58] R. Schreiber. “A new implementation of sparse Gaussian elimination.” In: *ACM Transactions on Mathematical Software* 8 (1982), pp. 256–276.

[59] J. A. Scott and J. D. Hogg. *A note on the solve phase of a multicore solver*. Tech. rep. RAL-TR-2010-07. Rutherford Appleton Laboratory, 2010.

[60] D. Shantsev, P. Jaysaval, S. de la Kethulle de Ryhove, P. Amestoy, A. Buttari, J.-Y. L’Excellent, and T. Mary. “Large-scale 3D EM modeling with a Block Low-Rank multifrontal direct solver.” In: *Geophysical Journal International* 209.3 (2017), pp. 1558–1571.

[61] Tz. Slavova. “Parallel triangular solution in the out-of-core multifrontal approach for solving large sparse linear systems.” Available as CERFACS Report TH/PA/09/59. Ph.D. dissertation. Institut National Polytechnique de Toulouse, Apr. 2009.

[62] R. Streich. “Controlled-Source Electromagnetic Approaches for Hydrocarbon Exploration and Monitoring on Land.” In: *Surveys in Geophysics* 37.1 (Jan. 2016), pp. 47–80. ISSN: 1573-0956. DOI: 10.1007/s10712-015-9336-0. URL: https://doi.org/10.1007/s10712-015-9336-0.

[63] A. Tarantola. “Inversion of seismic reflection data in the acoustic approximation.” In: *Geophysics* 49.8 (1984), pp. 1259–1266.

[64] C. Weisbecker. “Improving multifrontal solvers by means of algebraic block low-rank representations.” PhD Thesis. Institut National Polytechnique de Toulouse, Oct. 2013. URL: http://ethesis.inp-toulouse.fr/archive/00002471/.

[65] J. Xia. “Efficient Structured Multifrontal Factorization for General Large Sparse Matrices.” In: *SIAM Journal on Scientific Computing* 35.2 (2013), A832–A860. DOI: 10.1137/120867032.

[66] J. Xia, S. Chandrasekaran, M. Gu, and X. S. Li. “Fast Algorithms for hierarchically semiseparable matrices.” In: *Numerical Linear Algebra with Applications* 17.6 (2010), pp. 953–976.

[67] I. Yamazaki, X. S. Li, F.-H. Rouet, and B. Uçar. “On Partitioning and Reordering Problems in a Hierarchically Parallel Hybrid Linear Solver.” In: *2013 IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*. Cambridge, MA, United States, 2013, pp. 1391–1400.

[68] M. Yannakakis. “Computing the Minimum Fill-In is NP-Complete.” In: *SIAM Journal on Algebraic and Discrete Methods* 2 (1981), pp. 77–79.

[69] Y.-H. Yeung, J. Crouch, and A. Pothen. "Interactively Cutting and Constraining Vertices in Meshes Using Augmented Matrices." In: *ACM Trans. Graph.* 35.2 (Feb. 2016), 18:1–18:17. ISSN: 0730-0301. DOI: 10.1145/2856317. URL: http://doi.acm.org/10.1145/2856317.

[70] Y. H. Yeung, A. Pothen, M. Halappanavar, and Z. Huang. "AMPS: An Augmented Matrix Formulation for Principal Submatrix Updates with Application to Power Grids." In: *SIAM Journal on Scientific Computing* 39.5 (2017), S809–S827.

[71] J. Zach, A. Bjorke, T. Storen, and F. Maao. "3D inversion of marine CSEM data using a fast finite-difference time-domain forward code and approximate hessian-based optimization." In: *SEG Technical Program Expanded Abstracts 2008*. 2008, pp. 614–618. DOI: 10.1190/1.3063726. URL: https://library.seg.org/doi/abs/10.1190/1.3063726.

List of publications[1]

Articles in International Refereed Journals (accepted with minor revision)

[J1] P. Amestoy, J.-Y. L'Excellent, and G. Moreau. "On Exploiting Sparsity of Multiple Right-Hand Sides in Sparse Direct Solvers." In: *SIAM Journal on Scientific Computing,* also available as a research report RR-9122 (accepted with minor revision, 2018).

Articles in International Refereed Journals (to be submitted)

[JS1] P. Amestoy, A. Buttari, J.-Y. L'Excellent, T. Mary, and G. Moreau. "Exploiting the sparsity of right-hand sides to improve the asymptotic complexity of low-rank sparse direct solvers." In: *SIAM Journal on Scientific Computing* (to be submitted).

[JS2] P. Amestoy, S. de la Kethulle de Ryhove, J.-Y. L'Excellent, G. Moreau, and S. Daniil. "Efficient use of sparsity by direct solvers applied to 3D controlled-source EM problems." In: (To be submitted).

Abstracts in International Refereed Workshops

[W1] P. Amestoy, S. de la Kethulle de Ryhove, J.-Y. L'Excellent, G. Moreau, and S. Daniil. "Fast direct solver for 3D marine controlled-source EM problems based on sparsity utilization and BLR approximation." In: *24th EM Induction Workshop EMIW2018*. Aug. 2018.

[W2] P. Amestoy, J.-Y. L'Excellent, and G. Moreau. "Elimination Tree Flattening to Exploit Right-Hand Sides Sparsity." In: *SIAM Workshop on Combinatorial Scientific Computing 2018*. 2 pages extended abstract. Bergen, Norway, June 2018.

Communications in International Conferences

[C1] P. Amestoy, A. Buttari, J.-Y. L'Excellent, T. Mary, and G. Moreau. "Complexity and parallelism of the solution phase in sparse direct solvers." In: *10th International Workshop on Parallel Matrix Algorithms and Applications PMAA18*. June 2018.

[1] Authors are listed in alphabetical order.

[C2] P. Amestoy, J.-Y. L'Excellent, and G. Moreau. "Direct solution of sparse systems of linear equations with sparse multiple right-hand sides." In: *Sparse Days*. Cerfacs, Toulouse, France, Sept. 2017.

[C3] P. Amestoy, J.-Y. L'Excellent, and G. Moreau. "On the Solution Phase of Sparse Direct Solvers with Many Right-Hand Sides." In: *SIAM Conference on Computational Science and Engineering (SIAM CSE17)*. Atlanta, USA, Feb. 2017.

Other Communications

[O1] P. Amestoy, J.-Y. L'Excellent, and G. Moreau. "Avancées récentes et prochains défis de la phase de résolution pour les solveurs directs creux." In: *Journées MUMPS au Pôle Scientifique de Modélisation Numérique*. ENS de Lyon, Lyon, France, Mar. 2017.

[O2] P. Amestoy, J.-Y. L'Excellent, and G. Moreau. "Performance of the Solution Phase: Recent Work and Perspectives." In: *Internal communication*. Grenoble, France, May 2017.

[O3] P. Amestoy, J.-Y. L'Excellent, and G. Moreau. "Recent advances on the solution phase of sparse solvers with multiple sparse right-hand sides." In: *MUMPS User's days*. Inria, Montbonnot, France, June 2017.

[O4] P. Amestoy, J.-Y. L'Excellent, and G. Moreau. "When the Solution Phase is the Most Critical Part of Computation." In: *Solveurs linéaires HPC pour les études industrielles*. EDF Lab Paris-Saclay, Paris, France, Mar. 2017.