Combler l'écart entre H-Matrices et méthodes directes creuses pour la résolution de systèmes linéaires de grandes tailles

Aurélien Falco

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Aurélien Falco**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Bridging the Gap Between $\mathcal{H}$-Matrices and Sparse Direct Methods for the Solution of Large Linear Systems

---

Date de soutenance : 2019/06/24

Devant la commission d'examen composée de :

Emmanuel Agullo	Chargé de recherche, Inria	Encadrant
François Alouges	Professeur, École Polytechnique	Président du jury
Luc Giraud	Directeur de recherche, Inria	Directeur de thèse
Sabine Le Borne	Professeur, Technische Universität Hamburg	Rapporteuse
David Levadoux	Professeur associé, ONERA	Examinateur
Esmond Ng	Professeur, Lawrence Berkeley National Laboratory	Rapporteur
Grégoire Pont	Ingénieur de recherche, Airbus	Examinateur
Guillaume Sylvand	Ingénieur de recherche, Airbus	Co-directeur de thèse

– 2019 –

Titre Combler l'écart entre $\mathcal{H}$-Matrices et méthodes directes creuses pour la résolution de systèmes linéaires de grandes tailles

Résumé De nombreux phénomènes physiques peuvent être étudiés au moyen de modélisations et de simulations numériques, courantes dans les applications scientifiques. Pour être calculable sur un ordinateur, des techniques de discrétisation appropriées doivent être considérées, conduisant souvent à un ensemble d'équations linéaires dont les caractéristiques dépendent des techniques de discrétisation. D'un côté, la méthode des éléments finis conduit généralement à des systèmes linéaires creux, tandis que les méthodes des éléments finis de frontière conduisent à des systèmes linéaires denses. La taille des systèmes linéaires en découlant dépend du domaine où le phénomène physique étudié se produit et tend à devenir de plus en plus grand à mesure que les performances des infrastructures informatiques augmentent. Pour des raisons de robustesse numérique, les techniques de solution basées sur la factorisation de la matrice associée au système linéaire sont la méthode de choix utilisée lorsqu'elle est abordable. A cet égard, les méthodes hiérarchiques basées sur de la compression de rang faible ont permis une importante réduction des ressources de calcul nécessaires pour la résolution de systèmes linéaires denses au cours des deux dernières décennies. Pour les systèmes linéaires creux, leur utilisation reste un défi qui a été étudié à la fois par la communauté des matrices hiérarchiques et la communauté des matrices creuses. D'une part, la communauté des matrices hiérarchiques a d'abord exploité la structure creuse du problème via l'utilisation de la dissection emboitée. Bien que cette approche bénéficie de la structure hiérarchique qui en résulte, elle n'est pas aussi efficace que les solveurs creux en ce qui concerne l'exploitation des zéros et la séparation structurelle des zéros et des non-zéros. D'autre part, la factorisation creuse est accomplie de telle sorte qu'elle aboutit à une séquence d'opérations plus petites et denses, ce qui incite les solveurs à utiliser cette propriété et à exploiter les techniques de compression des méthodes hiérarchiques afin de réduire le coût de calcul de ces opérations élémentaires. Néanmoins, la structure hiérarchique globale peut être perdue si la compression des méthodes hiérarchiques n'est utilisée que localement sur des sous-matrices denses. Nous passons en revue ici les principales techniques employées par ces deux communautés, en essayant de mettre en évidence leurs propriétés communes et leurs limites respectives, en mettant l'accent sur les études qui visent à combler l'écart qui les séparent. Partant de ces observations, nous proposons une classe d'algorithmes hiérarchiques basés sur l'analyse symbolique de la structure des facteurs d'une matrice creuse. Ces algorithmes s'appuient sur une information symbolique pour grouper les inconnues entre elles et construire une structure hiérarchique cohérente avec la disposition des non-zéros de la matrice. Nos méthodes s'appuient également sur la compression de rang faible pour réduire la consommation mémoire des sous-matrices les plus grandes ainsi que le temps que met le solveur à trouver une solution. Nous comparons également des techniques de renumérotation se fondant sur des propriétés géométriques ou topologiques. Enfin, nous ouvrons la discussion à un couplage entre la méthode des éléments finis et la méthode des éléments finis de frontière dans un cadre logiciel unique.

Abstract Many physical phenomena may be studied through modeling and numerical simulations, commonplace in scientific applications. To be tractable on a computer, appropriated discretization techniques must be considered, which often lead to a set of linear equations whose features depend on the discretization techniques. Among them, the Finite Element Method usually leads to sparse linear systems whereas the Boundary Element Method leads to dense linear systems. The size of the resulting linear systems depends on the domain where the studied physical phenomenon develops and tends to become larger and larger as the performance of the computer facilities increases. For the sake of numerical robustness, the solution techniques based on the factorization of the matrix associated with the linear system are the methods of choice when affordable. In that respect, hierarchical methods based on low-rank compression have allowed a drastic reduction of the computational requirements for the solution of dense linear systems over the last two decades. For sparse linear systems, their application remains a challenge which has been studied by both the community of hierarchical matrices and the community of sparse matrices. On the one hand, the first step taken by the community of hierarchical matrices most often takes advantage of the sparsity of the problem through the use of nested dissection. While this approach benefits from the hierarchical structure, it is not, however, as efficient as sparse solvers regarding the exploitation of zeros and the structural separation of zeros from non-zeros. On the other hand, sparse factorization is organized so as to lead to a sequence of smaller dense operations, enticing sparse solvers to use this property and exploit compression techniques from hierarchical methods in order to reduce the computational cost of these elementary operations. Nonetheless, the globally hierarchical structure may be lost if the compression of hierarchical methods is used only locally on dense submatrices. We here review the main techniques that have been employed by both those communities, trying to highlight their common properties and their respective limits with a special emphasis on studies that have aimed to bridge the gap between them. With these observations in mind, we propose a class of hierarchical algorithms based on the symbolic analysis of the structure of the factors of a sparse matrix. These algorithms rely on a symbolic information to cluster and construct a hierarchical structure coherent with the non-zero pattern of the matrix. Moreover, the resulting hierarchical matrix relies on low-rank compression for the reduction of the memory consumption of large submatrices as well as the time to solution of the solver. We also compare multiple ordering techniques based on geometrical or topological properties. Finally, we open the discussion to a coupling between the Finite Element Method and the Boundary Element Method in a unified computational framework.

Keywords Sparse Matrices, $\mathcal{H}$-Matrices, Low-Rank Compression, Linear Algebra, Finite Elements, FEM/BEM Coupling

Mots-clés Matrices Creuses, $\mathcal{H}$-Matrices, Compression de Rang Faible, Algèbre Linéaire, Éléments Finis, Couplage FEM/BEM

Laboratoire d'accueil HiePACS team, Inria Bordeaux Sud-ouest, 200, avenue de la Vieille Tour, 33405 TALENCE

Résumé long L'utilisation de simulations numériques permet l'étude de phénomènes complexes en limitant, voire en évitant, le recours à l'expérimentation réelle souvent coûteuse. De telles simulations sont utilisées depuis des décennies dans de nombreux domaines scientifiques. Dans l'industrie aéronautique, elles sont par exemple utilisées afin d'étudier des phénomènes physiques tels que la propagation d'ondes électromagnétiques ou acoustiques. Un exemple de ce dernier domaine est l'étude du bruit généré par les moteurs d'un avion au décollage. Certains aéroports périurbains sont aujourd'hui sujets à des réglementations de plus en plus strictes visant à réduire les nuisances sonores autour des pistes de décollage pour le bien-être des personnes résidant dans la proximité immédiate de l'aéroport. Les constructeurs sont donc obligés de réduire le bruit généré par leurs avions et exploitent intensivement des simulations numériques pour améliorer le design des aéronefs afin d'atteindre cet objectif. Le bruit peut être représenté par une onde acoustique se propageant autour de l'appareil. Un modèle numérique peut être utilisé pour étudier sa propagation et son interaction avec l'écoulement généré par le moteur.

Un phénomène physique tel ce problème d'aéroacoustique mentionné ci-dessus peut-être modélisé par des équations aux dérivées partielles linéaires (EDP) définies dans un domaine englobant l'espace d'étude de la propagation de l'onde. Cette EDP, dont la solution est une fonction souvent continue, est transformée en un problème discret calculable sur un ordinateur via des techniques de discrétisation, telles que la méthode des éléments finis (FEM : Finite Element Method) ou la méthode des éléments finis de frontière (BEM : Boundary Element Method). Dans ce cas, le problème discret prend la forme d'un système linéaire. Dans cette thèse, nous nous intéressons à la résolution de problèmes discrétisés par une technique qui couple la FEM et la BEM. La FEM est utilisée sur un maillage volumique qui discrétise l'espace 3D, la BEM est utilisée sur un maillage surfacique. Le système linéaire associé peut être écrit sous la forme $Ax = b$, où la matrice A possède naturellement une structure bloc 2×2 :

$$A = \begin{bmatrix} A_{vv} & A_{vs} \\ A_{sv} & A_{ss} \end{bmatrix}.$$

Le bloc (1,1) A_{vv} est une matrice de grande taille, car définie sur un maillage volumique 3D, et creuse (très peu de coefficients non-nuls), de par la formulation FEM, alors que le bloc (2,2) A_{ss} est une matrice de plus petite dimension, car définie sur un maillage surfacique, et pleine (très peu de coefficients nuls), de par la formulation BEM. Les matrices extra-diagonales A_{vs} et A_{sv} ont une structure creuse. La résolution efficace du problème complet nécessite donc l'utilisation de techniques adaptées aux caractéristiques très différentes des blocs constituant la matrice complète.

Deux grandes classes de méthodes peuvent être utilisées pour la résolution de systèmes linéaires, les méthodes directes et les méthodes itératives. Nous nous concentrons dans cette thèse sur les méthodes dites directes qui s'appuient sur l'écriture de la matrice A sous la forme d'un produit de matrices ayant des structures simples autorisant des résolutions

aisées de systèmes linéaires associés à chacune d'elles. La plus connue et générale des factorisations est la factorisation $A = LU$, où L une matrice triangulaire inférieure et U une matrice triangulaire supérieure. Le système linéaire peut alors être résolu en calculant successivement une solution pour les deux systèmes linéaires triangulaires associés à la matrice L pour le premier et U pour le second. Leurs propriétés de robustesse et de précision combinées à leur comportement prédictibles rendent ces méthodes très populaires et usitées dans les codes de simulations.

En aéroacoustique, la taille des matrices croît en fonction de la fréquence des ondes étudiées ou de la taille du domaine par rapport à la longueur d'onde ; ce qui nécessite le développement de solveurs linéaires efficaces et adaptés à la taille toujours croissante des problèmes à résoudre avec des moyens de calcul limité.

Dans ce contexte, les matrices hiérarchiques ($\mathcal{H}$-Matrices) [112] ont permis de réduire significativement les contraintes calculatoires (mémoire et temps de calcul) pour des classes de matrices pleines qui peuvent être approchées par des matrices de rang faible. Cette représentation sous forme de rang faible permet d'exprimer les calculs matriciels élémentaires et rendent en particulier possible le calcul des factorisations à moindre coût.

En ce qui concerne les matrices creuses, la procédure de factorisation d'une telle matrice introduit de nouveaux coefficients. Ce phénomène est appelé remplissage (fill-in en anglais). Les solveurs directs creux utilisent généralement des techniques de renumérotation d'inconnues (reordering) telles que la dissection emboitée [90] pour réduire ce remplissage. Ils s'appuient également sur la notion de factorisation symbolique afin de déterminer la structure de la matrice factorisée avant le calcul effectif de la factorisation numérique.

Alors que les $\mathcal{H}$-Matrices étaient à l'origine utilisées pour résoudre des problèmes pleins, leur utilisation a été étudiée pour des systèmes linéaires creux par l'introduction de la dissection emboitée dans l'arithmétique $\mathcal{H}$ [143] (remplaçant la bissection récursive utilisée sur des problèmes pleins). D'autre part, la compression de rang faible a également été introduite dans les solveurs directs creux [191] pour la compression des sous-matrices pleines qui interviennent dans la factorisation de matrices creuses. Nous nous posons donc ici la question de l'utilisation efficace des techniques de compression de rang faible et des techniques creuses (renumérotation, factorisation symbolique) pour la résolution d'un système linéaire creux.

Dans le contexte de la résolution d'un couplage FEM-BEM en utilisant des $\mathcal{H}$-Matrices, nous nous concentrons principalement sur l'exploitation du caractère creux de la structure des non-zéros et du remplissage pour la factorisation du bloc A_{vv} (FEM-FEM). Nous avons ainsi étudié l'intégration de la dissection emboitée au sein des $\mathcal{H}$-Matrices, se reposant sur des informations géométriques ou topologiques, et confirmé les précédentes tendances observées par la communauté $\mathcal{H}$. Nous avons aussi introduit de nouvelles techniques permettant d'éviter l'apparition de blocs longs et fins lors de l'utilisation de la dissection emboitée dans la construction d'une $\mathcal{H}$-Matrice. Les principales contributions de cette thèse concernent l'introduction de nouveaux algorithmes permettant l'utilisation d'une information symbolique dans la hiérarchie d'une $\mathcal{H}$-Matrice. Pour cela, les algorithmes développés approchent le problème de deux façons. Dans un premier temps, nous

introduisons des méthodes qui permettent de localiser le remplissage dû à la factorisation numérique et d'éviter le stockage de coefficients nuls dans la $\mathcal{H}$-Matrice. Nous avons regroupé ces méthodes sous le nom de Factorisations Symboliques Hiérarchiques. Nous discutons de plusieurs possibilités afin d'accomplir une telle factorisation symbolique et quantifions leur impact sur la mémoire et le temps requis pour effectuer une factorisation numérique. Dans un deuxième temps, nous introduisons aussi des méthodes permettant un meilleur groupement d'inconnues situées au sein des séparateurs calculés par la dissection emboitée en utilisant l'information symbolique de leurs interactions avec d'autres inconnues.

Dans le chapitre 1, nous présentons le contexte théorique et industriel de cette thèse. Nous développons d'abord le contexte industriel et la transformation d'un problème physique d'aéroacoustique en un système linéaire utilisant la FEM et la BEM. Nous discutons ensuite de la solution des systèmes linéaires pleins utilisant des techniques de compression telles que les $\mathcal{H}$-Matrices. La solution des systèmes linéaires creux est ensuite discutée à travers le prisme des techniques de renumérotation, des structures supernodales (ou quotientes) et de la factorisation symbolique. La solution de l'ensemble du couplage FEM/BEM est finalement traitée en présentant les différentes méthodes sur lesquelles nous nous concentrerons dans le reste de ce document.

Dans le chapitre 2, nous discutons de la factorisation du sous-système creux par l'utilisation de la compression de rang faible. Un état de l'art est d'abord établi, en énumérant les travaux de la communauté $\mathcal{H}$, qui a étendu les $\mathcal{H}$-Matrices en utilisant la technique de dissection emboitée, et les travaux de la communauté creuse, qui a introduit des techniques de compression de rang faible pour les sous-matrices pleines résultant de la factorisation d'une matrice creuse. Nous présentons ensuite les diverses techniques de renumérotation sur lesquelles nous nous appuyons et que nous comparons dans cette thèse ; c'est-à-dire des dissections emboitées basées sur des informations géométriques et d'autres utilisant des informations topologiques. Nous discutons aussi de comment adapter efficacement les dissections emboitées pour les $\mathcal{H}$-Matrices, par exemple par la prévention de l'apparition des blocs longs et fins qui résultent naturellement de la combinaison entre ces deux méthodes. Le critère d'admissibilité lié à la décision de quels blocs d'une $\mathcal{H}$-Matrice seront stockés dans un format de rang faible est également discuté dans un contexte creux. L'utilisation des informations symboliques est discutée en § 2.4. Nous détaillons diverses méthodes pour tirer parti du caractère creux de la répartition des non-zéros dans cette section. Le premier ensemble de méthodes repose sur différentes façons de calculer une information symbolique utilisable dans un contexte hiérarchique, c'est-à-dire effectuer une Factorisation Symbolique Hiérarchique. Le second ensemble de méthodes considère la division des séparateurs issus de la dissection emboitée, parmi lesquelles nous proposons des méthodes conscientes des interactions (interactions-aware) entre supernœuds reposant sur des informations symboliques. La discussion est ouverte au couplage FEM/BEM dans la dernière section de ce chapitre.

Enfin, les expériences et les résultats liées aux méthodes présentées auparavant sont discutés dans le chapitre 3. Nous nous concentrons d'abord sur l'efficacité des étapes d'analyse symbolique hiérarchique (de prétraitement) que nous avons proposées. Nous

montrons ainsi qu'une approche *ascendante* (bottom-up) calculant des informations *vers la droite* (right-looking) permet de déterminer plus rapidement les informations symboliques de toute la hiérarchie d'une $\mathcal{H}$-Matrice. Leur impact sur la factorisation numérique de la matrice creuse A_{vv} est ensuite discuté. Nous montrons que l'utilisation de techniques de factorisation symbolique réduit la consommation mémoire et de temps de notre solveur hiérarchique par rapport à sa version sans factorisation symbolique. Nous confirmons aussi que l'utilisation de la dissection emboitée au sein des $\mathcal{H}$-Matrices permet des gains de performance par rapport à la bissection récursive usuellement employée pour des matrices pleines. L'efficacité parallèle de la factorisation numérique est ensuite validée, afin de pouvoir effectuer des tests sur des problèmes plus grands, avec plus de cent millions d'inconnues sur une machine ayant une plus grande capacité mémoire. Nous observons ainsi une stabilité et la complexité de nos algorithmes à plus large échelle. Les meilleures méthodes que nous avons développées sont ensuite mises en perspective avec un solveur direct creux de référence, solveur utilisant aussi de la compression de rang faible. Nous montrons que nous avons une consommation mémoire théorique et pratique proche de ce solveur référence lors de la désactivation de la compression, validant notre implémentation de techniques creuses telles que la factorisation symbolique ou la dissection emboitée. L'emploi de la compression de rang faible permet ensuite de réduire la consommation mémoire du solveur hiérarchique. Cependant nous exhibons des problèmes de performance en terme de nombre d'opérations et de temps de calcul, qui ne semblent pas correspondre à la complexité promise par les $\mathcal{H}$-Matrices, certainement dûs à un défaut d'implémentation. Enfin, nous montrons des résultats préliminaires concernant l'impact de nos méthodes sur le couplage FEM/BEM. Nos méthodes parviennent ainsi à réduire le coût d'une partie du calcul global, plus particulièrement impliquant la sous-matrice A_{vv}, et permet ainsi une solution plus rapide du couplage. De futures travaux concernant l'utilisation de l'information symbolique non limitée à la sous-matrice A_{vv} mais aussi étendue à la sous-matrice A_{sv} devrait de même permettre un large gain de performance quant à la résolution du couplage FEM/BEM.

Acknowledgement

I would like to thank, first of all, my supervisors Emmanuel Agullo, Luc Giraud and Guillaume Sylvand for their support over these years. Each of you has contributed to the thesis from his own perspective and has given a particular relish to this work. I also thank Esmond Ng and Sabine Le Borne for reviewing the manuscript and their useful remarks on the document. I would also like to thank the other members of the jury, François Alouges, for presiding it, but also David Levadoux and Grégoire Pont. Thank you all for granting me the title of Doctor. I will try to be worthy of this honor you have bestowed upon me!

I thank Airbus, as well as the Conseil Régional d'Aquitaine, for granting the funds for this thesis. Thanks also to Jerome Robert and Denis Barbier for our fruitful exchanges over the implementation of the methods developed in this thesis in the Airbus software package.

I thank Marc Sergent, Nathalie Furmento, Samuel Thibault, and more generally the ex-RUNTIME team, for their technical support over the StarPU software package. I am also grateful to the development team of the MUMPS solver for their insights and constructing remarks in order to establish fair comparisons, including Abdou Guermouche, Théo Mary, Patrick Amestoy, Alfredo Buttari and Jean-Yves L'Excellent. I thank the PlaFRIM and GENCI facilities for allowing us to use their platform for numerical experiments. Finally, many thanks to Florent Pruvost, François Rué, Luka Stanisic, and numerous others, for their help on technical problems, sometimes resulting from a dysfunction with the chair-to-keyboard interface.

Et enfin, la douce langue française portera (quand même !) mes remerciements plus personnels : je remercie donc mes collègues d'Inria pour les moments de détente, les parties de baby-foot, les parties de cartes, ou bars à vins, ainsi que les collègues de l'Enseirb-Matmeca (quand ce ne sont pas les mêmes), pour ces découvertes partagées ensemble. Si l'on pouvait remercier les montagnes, les Pyrénées en feraient partie.

Pour conclure, je remercie mes amis les plus proches (anonymes pour éviter les incidents diplomatiques) ainsi que ma famille, qui commence à déborder de docteurs.

Au vu du déroulé des évènements, je voudrais dédicacer cette thèse à ma grand-mère Lise et à ma nièce Juliette.

Contents

List of Algorithms

Glossary

ACA	Adaptive Cross Approximation. A method for compression. 22–24, 26
Adjacency Graph	Graph representing the interactions between unknowns. An edge in the graph corresponds to an entry in the associated matrix. 45
Admissibility Condition	Boolean function used in the $\mathcal{H}$-Matrix technique to decide the storage of a submatrix, i.e., $\mathcal{R}k$-Matrix or Full-Matrix. 3, 22, 25–27, 29, 32, 35–37, 85, 106, 107, 109, 110, 144, 154, 157, 174, 177, 183, 201, 203
BEM	Boundary Element Method. A discretization method leading to dense linear systems. 4, 5, 8–12, 15, 20, 21, 24, 27, 28, 35, 36, 72, 73, 75, 80, 83, 151, 155, 192, 195, 196, 198, 200, 202, 203, 207–210
Block Cluster Tree	The skeleton of an $\mathcal{H}$-Matrix. 22, 27, 36, 37, 39, 90, 92, 101, 103, 105, 106, 113, 114, 116, 118, 134, 154, 158
BLR	Block Low Rank. 42, 84, 94, 95, 110, 134, 157, 159, 160, 196
Bounding Box	Cuboid around a cluster of points used to compute dimensions of the cluster more easily (diameter, ...). 29, 30, 32, 35, 97, 157, 174, 177
Cluster Tree	A tree used in the construction of an $\mathcal{H}$-Matrix. xii, 22, 27–30, 36–40, 84–93, 96, 100–103, 106, 111–113, 115–117, 119, 121–130, 132–136, 138, 144, 147, 158, 159, 161, 163, 164, 201, 206

FEM	Finite Element Method. A discretization method leading to sparse linear systems. 4, 5, 8–12, 15, 42, 43, 68, 69, 72, 73, 75, 83, 84, 151, 155, 159, 192, 195, 196, 200, 202, 203, 207–210
Fill-in	Phenomenon that occurs when computing a LU decomposition, when a zero entry becomes non-zero. iv, 2, 44–46, 48, 50, 51, 58, 61, 64, 69, 84–86, 88, 110, 114, 115, 117, 120, 121, 129, 177, 200, 201, 203
FMM	Fast Multipole Method. A numerical technique to speed up calculations, in particular in the n-body problem. 16, 20, 21, 30, 32, 71, 75
$\mathcal{H}$-Matrix	Hierarchical matrix. iv–vi, viii–xi, 2, 3, 5, 12, 13, 15, 16, 21–23, 27, 30, 36–42, 47, 73, 75, 80–86, 88–93, 95, 96, 101, 104, 106, 107, 110, 111, 113, 115–118, 120, 121, 126–128, 130, 131, 133–135, 140, 142, 145, 146, 151–154, 157–159, 168, 172, 174, 177, 178, 180, 183, 191–196, 198–202, 207
HPC	Also called supercomputing, it is the art of using supercomputers or computer clusters efficiently to obtain the best performance of the hardware. 21, 73
LU factorization	Also referred to as LU decomposition. A matrix factorization involved in solving systems of linear equations. Other decompositions include QR, Cholesky, LDL^t. 15–17, 19, 21, 36, 40, 44, 45, 53, 69–71, 75, 107, 110, 132
Nested Dissection	A divide and conquer heuristic used in graph partitioning. 2, 3, 50–53, 56, 57, 61, 62, 65, 83–86, 88–92, 96, 98–100, 110, 111, 121, 128–131, 133, 135, 136, 142, 144, 149–154, 156, 159, 160, 170, 172, 174, 177, 180, 183, 191, 192, 195, 196, 198–204, 207
Out-of-core	Method involving the storage of matrix blocks on disk to avoid running out-of-memory. 75, 76, 78, 196, 198, 199
PDE	Partial Differential Equations. 4, 8, 9, 21

Preconditioner	Used to condition a problem so that fewer iterative steps need to be performed. 14, 15, 21
Quotient Graph	Aggregated form of an adjacency graph over supernodes. 56
RAM	Random-Access Memory. 157, 160, 180, 185, 199
RHS	Right-Hand Side of a linear system. 13, 15, 43
$\mathcal{R}k$-Matrix	A method for matrix compression, which transforms a matrix into a product of two smaller matrices. 22, 25, 26, 39, 85, 86, 107, 128
Schur complement	A matrix block that arises when performing block Gaussian elimination. 40, 70, 71, 74–79, 159, 192, 194–196, 198–200, 202
Separator	The subdomain of nodes separating a graph in two independent subdomains while computing a nested dissection. 106
Supernode	Group of nodes of the adjacency graph, usually corresponding to nodes with the same outer interactions. 53
SVD	Singular Value Decomposition. A canonical linear algebra decomposition of matrices usable for compression. 22, 23, 25, 26
Symbolic factorization	A method to locate the non-zeros in the factors of a matrix, often combined with nested dissection in sparse solvers to optimize the number of stored non-zeros. 44, 50, 57, 58, 61–66, 69, 85, 86, 92, 110, 111, 114, 121, 124, 125, 128, 129, 135, 136, 159, 163, 167, 170, 171, 174, 178, 195

List of Abbreviations and Symbols

G^*	Elimination of graph G. 45
G/P	Quotient graph of G using partition P. 61
M_{*j}	Column j of matrix M. 24
M_{i*}	Row i of matrix M. 24
$\|S\|$	Cardinality of a set S, i.e., the number of elements it contains. 24
$\|M\|$	Normal norm of a matrix M. 13
$\|M\|_2$	Spectral norm of a matrix M. 23
$\wedge$	Logical conjunction (AND). 36, 40, 44, 47, 60, 61, 63–65, 67, 68, 91, 97, 101, 116–118, 120, 148, 149
$\vee$	Logical disjunction (OR). 36, 44, 101, 117, 120, 145
$A \geq B$	Greater or equal relation operator. For a set, a vector, or a cluster of elements, it is defined as true if: $\forall (a, b) \in (A, B),\ a \geq b$. 17
$A > B$	Strict greater relation operator. For a set, a vector, or a cluster of elements, it is defined as true if: $\forall (a, b) \in (A, B),\ a > b$. 118
$A \leq B$	Lower or equal relation operator. For a set, a vector, or a cluster of elements, it is defined as true if: $\forall (a, b) \in (A, B),\ a \leq b$. 23
$A < B$	Strict Lower relation operator. For a set, a vector, or a cluster of elements, it is defined as true if: $\forall (a, b) \in (A, B),\ a < b$. 118
M^H	Conjugate (or Hermitian) transpose of M. The notation M^* is not used to avoid confusion with the notation G^*. 16
$\overline{M}$	Conjugate of M. 26
M^T	Transpose of M. 16

Introduction

The use of numerical simulations in the industry grants scientists the ability to study complex phenomena by reducing, or even removing, the need for physical, and often costly, experiments. Such simulations have been used for decades in many scientific fields. In the aircraft industry, they are for instance used to study physical phenomena such as the propagation of electromagnetic or acoustic waves. An example of this last field is the study of the noise generated by the engines of an aircraft at takeoff. Some suburban airports are nowadays ensuing strict regulations intended to reduce the noise in the area surrounding the runways used by airplanes for the welfare of the inhabitants living in the immediate vicinity of the airport. Aircraft manufacturers are thus enticed to reduce the noise generated by an aircraft and heavily rely on numerical simulations to enhance their aircrafts for this purpose. The noise may be represented by an acoustic wave propagating around the plane. To study its propagation, a numerical model may be used.

A physical phenomenon such as the aeroacoustic problem mentioned just above is expressed under the form of Partial Differential Equations (PDE) over a domain covering the propagation space. For a solution on a computer, the continuous formulation of PDEs is transformed into a discrete problem through discretization techniques, such as the Finite Element Method (FEM) or the Boundary Element Method (BEM). This discrete problem is written under the form of a linear system. We study in this thesis the solution of a coupling between the FEM and the BEM. The FEM is used on a volume mesh to discretize a 3D space whereas the BEM is used on a surface mesh. The associated linear system may be formulated as $Ax = b$, where the 2×2 matrix A may be decomposed into:

$$A = \begin{bmatrix} A_{vv} & A_{vs} \\ A_{sv} & A_{ss} \end{bmatrix}.$$

The (1,1) block A_{vv} is a large matrix due to the 3D volume mesh, which is also sparse (with few non-zero entries), due to the FEM formulation, whereas the (2,2) block A_{ss} is a smaller matrix as it corresponds to a surface mesh, and is dense (with few zeros entries) due to the BEM formulation. The efficient solution of the overall system therefore must rely on different techniques suited for the very different characteristics of these subsystems.

Two main classes of methods may be used for the solution of linear systems, the *direct* methods and the *iterative* methods. We focus in this thesis on direct solutions relying on factorization methods decomposing a matrix A into a product of matrices with simpler structures leading to an easier solution. The most common factorization of a matrix A is the decomposition into an LU form, with L a lower triangular matrix and

U an upper triangular matrix. The solution of the linear system associated with A can then be decomposed into the solution of two triangular linear systems associated with the matrix L, firstly, and U, secondly. The robustness, numerical stability and accuracy of direct methods have led them to be very popular and common in numerical simulation softwares.

In the aeroacoustic field, the size of the studied matrices grows with the frequency of the problem or the size of the domain compared to the wavelength. Large-scale solvers must consequently be implemented to manage an increase in terms of computational requirements with a limited amount of computational resources. Hierarchical matrices ($\mathcal{H}$-Matrices) [112] have drastically reduced the (time and memory) requirements of the computation of solutions involving dense matrices with low-rank characteristics. The representation of a matrix using a low-rank format relies on an approximation of this matrix with a controlled accuracy leading to a smaller arithmetic and memory complexity of the factorization.

Regarding the case of the factorization of a sparse matrix, new entries are generated by the procedure. This phenomenon is referred to as fill-in. Sparse direct solvers usually rely on reordering techniques such as the nested dissection [90] to reduce this fill-in. They also rely on symbolic factorization to determine the structure of the factorized matrix before the actual numerical factorization.

While $\mathcal{H}$-Matrices were originally used for the solution of dense problems, their use has been studied for sparse linear systems by the introduction of nested dissection in the $\mathcal{H}$ framework [143] (replacing the recursive bisection usually used in the case of dense linear systems). On the other hand, low-rank compression has also been introduced in sparse direct solvers [191] for the compression of dense submatrices that arise in the factorization of sparse matrices. We therefore investigate here the efficient use of techniques arising from the $\mathcal{H}$-Matrix community (hierarchical low-rank compression) as well as from the Sparse-Direct community (reordering, symbolic factorization) for the solution of large sparse linear systems.

In the context of the computation of the solution of a FEM/BEM coupling using $\mathcal{H}$-Matrices, we mainly focus on exploiting the sparsity of the pattern of non-zeros and fill-in for the factorization of the (FEM-FEM) block A_{vv}. We have indeed studied the introduction of the nested dissection in the $\mathcal{H}$ arithmetic, using geometric of topological information, and confirmed previous trends shown by the $\mathcal{H}$-Matrix community. We have also introduced new techniques to prevent the formation of tall & skinny blocks when nested dissection is used in the construction of a $\mathcal{H}$-Matrix. The main contributions of this thesis are the introduction of novel algorithms for the symbolic analysis of the sparsity of a matrix that may be used in the hierarchy of a $\mathcal{H}$-Matrix. These algorithms may be divided in two main categories. The first category of methods aim to locate the fill-in generated by the numerical factorization and avoiding the storage of zero coefficients in the $\mathcal{H}$-Matrix. We name these methods Hierarchical Symbolic Factorizations (HSF). We discuss several possibilities for achieving such a symbolic factorization and quantify their impact on memory and the time required to perform a numerical factorization. The second category of methods aim to cluster unknowns located within the separators calculated by

the nested dissection by using the symbolic information of their interactions with other unknowns. We name them Interactions-Aware (IA) separator clusterings.

In Chapter 1, we present the theoretical and industrial background of this thesis. We first develop the industrial context and the transformation of a physical aeroacoustic problem into a linear system using the FEM and the BEM. We then discuss the solution of dense linear systems using techniques of compression such as the $\mathcal{H}$-Matrices. The solution of sparse linear systems is then discussed through the prism of reordering techniques, supernodal (or quotient) structures and symbolic factorization. The solution of the overall FEM/BEM coupling is finally addressed by presenting the methods we will focus on in the rest of this document.

In Chapter 2, we discuss the solution of the sparse subsystem using low-rank compression. A state of the art is first provided, listing works of the $\mathcal{H}$-Matrix community, which has extended $\mathcal{H}$-Matrices using nested dissection, and works from the Sparse-Direct community, which has introduced low-rank techniques of compression for dense submatrices arising in the sparse factorization. We then present the multiple reordering techniques we rely on and compare in this thesis, i.e., a nested dissection based on geometric information and nested dissections based on topological information. We also discuss how to efficiently adapt the nested dissection for $\mathcal{H}$-Matrices, for example by the prevention of tall & skinny blocks that naturally arise from the combination of these two methods. The admissibility condition involved in the decision of which blocks of a $\mathcal{H}$-Matrix will be stored in a low-rank format is also briefly discussed in a sparse context, of which preliminary results raise questions about its formulation. The use of symbolic information is discussed in § 2.4. We detail multiple methods to take advantage of the sparsity pattern of non-zeros in this section. The first set of methods relies on different ways of computing the symbolic information and using it in a hierarchical context, i.e., computing a HSF. The second set of methods considers the division of separators arising from nested dissection, among which we propose Interactions-Aware methods relying on symbolic information. The discussion is open to the FEM/BEM coupling in the last section of this chapter.

Finally, experiments and results are discussed in Chapter 3. We first focus on the efficiency of the (pre-processing) hierarchical symbolic analysis steps we have proposed. Their impact on the numerical factorization of the sparse matrix A_{vv} is then discussed, as well as the parallel efficiency of the factorization. Once the parallel efficiency has been validated, we have run tests on larger problems with more than one hundred million unknowns on a machine with a larger memory capacity, in an effort to understand the behavior of our methods on large-scale problems. The best methods we have developed are then put in perspective with a reference sparse direct solver, also relying on low-rank compression. Finally, we show preliminary results of the impact of our methods on the FEM/BEM coupling.

Chapter 1

General Introduction

In this chapter, we present the general context of this thesis by explaining the phenomena we want to examine, their implications and their mathematical formulations. We first cover the physical and mathematical context in § 1.1, explaining why and how we need to study wave propagation via simulations in § 1.1.1 and what are the numerical computational methods used for the discretization of this type of physical systems, i.e., Finite Element Method (FEM) and Boundary Element Method (BEM), in § 1.1.2. Then we detail the solution of the linear systems arising from these discretization schemes in §§ 1.2 and 1.3. Finally, the global coupled system, its formulation and its solution, are further discussed in § 1.4.

1.1 General Framework

Numerical simulations have been used for decades in industry to safely design new products. They can be used for a great range of problems: testing, safety engineering, quality measurements, and more generally for model validation. Motivated by the industrial context of this thesis, involving a partnership with Airbus, we are more interested in the design of aircrafts and the physical problems involved, such as electromagnetic and aeroacoustic. Designing based on a numerical prototype has many advantages, as it can be less expensive, easier, safer, and faster than creating and testing a physical one. For example, in the framework we are interested in, it can be used to study the performance and efficiency of an airplane before its effective assembly. In order to perform such simulations, several steps are required. The problem is first modeled into a system of Partial Differential Equations (PDE) such as for example the Helmholtz equation for acoustic problems or Maxwell equations for electromagnetic problems. When analytical solution cannot be exhibited, the continuous quantities involved in the PDEs have to be represented by discrete values that can further be handled on a computer. Moving from continuous to discrete representation is referred to as discretization. The FEM [50, 82, 168, 201] and the BEM [33, 167, 175, 187] are two techniques involving the discretization of the space variable. It should be noted that BEM can be categorized as a special case of FEM. They differ in their mathematical formulation of a physical

problem and, due to this formulation, in the properties the studied medium is required to possess for the method to be applicable. A medium represents the characteristics of the space domain in which the problem is set. The resulting formulation is a linear system of n algebraic equations with different properties depending on the method used. The associated matrix is either sparse for a FEM discretization or dense for a BEM discretization. The adjective "sparse" means a matrix is filled with a lot of zero coefficients whereas "dense" means it is filled with only very few zeros. The system can thus be formulated in the matrix form $Ax = b$, where A is a matrix of size $n \times n$ and the vectors x and b are of size n. Such a system can be solved using *direct methods*, based on a factorization of A, or *iterative methods*, in which matrix-vector products are used to build a series of iterates that should converge to the solution [146].

We consider here three-dimensional electromagnetic and aeroacoustic problems, or, for simpler test cases, simpler objects such as cubes or spheres, with a 7-point stencil for a 3D Laplace's equation for example. In the latter case, if N is the number of discretization points on each edge of the cube, then the matrix A has N^3 rows and N^3 columns with about $7N^3$ nonzero coefficients; even a sparse-optimized factorization would lead to $\mathcal{O}(N^6)$ floating-point operations [146]. One of the challenges of numerical simulations is therefore to make the large computational costs of such problems match modern architectures, either by reducing the amount of computation or by exploiting the hardware architectures to parallelize them.

In the context of this work, various methods may be used for the solution of a linear system arising from a discretization using the FEM and the BEM of an electromagnetic or aeroacoustic problem. We focus in this thesis on algorithmic optimizations for these methods, and more specifically, we will be interested in the solution of the overall system using $\mathcal{H}$-Matrices [112]. This hierarchical format is introduced in § 1.2.3 and will be the main subject of Chapter 2.

After a presentation of the application context of this work in § 1.1.1, we introduce the reader in § 1.1.2 to the numerical approximation methods used in this thesis, i.e., the FEM and the BEM. Finally, we discuss the linear systems arising from these methods, their characteristics and their solutions in § 1.1.3.

1.1.1 Industrial Context

Numerical simulations can be used to study many physical problems, such as electromagnetic [139, 144] or aeroacoustic [32, 49, 160, 185] problems, which are the categories of problems we focus on, in the context of the industrial partnership of this thesis. Accurately analyzing electromagnetic and aeroacoustic waves is a major challenge for aeronautics and implies a need for precise simulations. Related applications include the conception and installation of antenna for the electromagnetic case, as well as the study of the noise generated by an airplane engine for aeroacoustic problems. This may be studied for an airplane during takeoff, landing, cruise or taxi (movement of the aircraft on the ground).

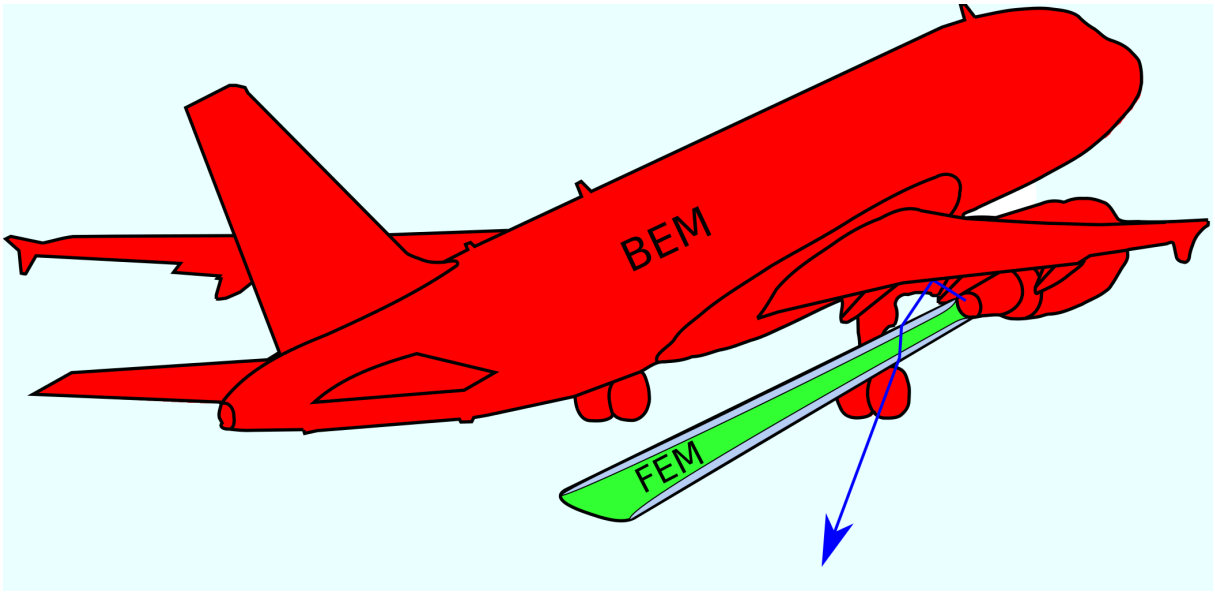
Let us focus on one example of aeroacoustic problem. Why do we need to study the noise generated at takeoff? The answer lies in the effect of noise pollution on health and behavior of human beings, to the point where categorizing airplanes according to the noise they produce is a part of regulations issued by governments or airports. Possible effects of exposure to chronic environmental noise have been reported through a number of studies [51, 83], more particularly on children, and especially on cognition, motivation or the cardiovascular system. But mainly, noise exposure causes annoyance [182], for example through sleep disturbance. Therefore, measuring and classifying airplanes according to the amount of noise they produce is important, especially for airports, as they must carry out special operations and issue regulations for aircraft engines to address this problem. Heathrow Airport for example, being the busiest airport of Europe, has issued a report named "A quieter Heathrow" presenting the operations conducted to "tackle aircraft noise". Special restrictions [1] like the "Quota Count" (QC) system have been implemented. The QC system has been introduced in 1993 and is now used in London's Heathrow, Gatwick and Stansted airports. In this system, each aircraft is awarded a quota count value depending on the amount of noise it generated under controlled certification conditions both at takeoff and landing; airports have a fixed quota to observe. For example, the Airbus A380 uses no more than half of the quota of a Boeing 747, while being larger [2] [3], therefore providing airlines with an incentive to operate quieter aircrafts like the A380. It seems the QC system is somewhat efficient to provide a measure on the harm caused by the noise generated by an aircraft [123]. However, later studies regarding the London Heathrow airport still suggest that "high levels of aircraft noise are associated with an increased risk of stroke, coronary heart disease, and cardiovascular disease" [118], while recommending more work to be done on this field of research to avoid statistical bias due to the uncertainty of some data "such as ethnicity and smoking". All of this leads to the conclusion that categorizing airplanes according to the noise generated should not to be taken lightly and is a necessity for airports and thus aeronautics companies.

In our Airbus framework, the software used to solve aeroacoustic problems is named Actipole and its counterpart for electromagnetism is named Elfipole. For the rest of this document, an aeroacoustic study case which can be solved using Actipole is considered as being the main objective to achieve. Let us consider this situation: we want to analyze the propagation of an acoustic wave emitted by the engine of a plane during takeoff (Fig. 1.1a). We are not interested here in how the noise is generated inside the engine (modeled as an acoustic waveguide) but only in how it propagates outside of it. Should we measure the noise emitted by an airplane in another context, at landing for example, other sources of noise might appear, like the landing gear, typically wheels, as they produce noise mainly due to turbulences around the tires (and possibly the flaps, when extended). While the problem setting is slightly changed, the need for simulation still holds: the system would simply be more complex.

[1] http://www.heathrow.com/noise/heathrow-operations/night-flights
[2] http://www.globalaircraft.org/planes/airbus_a380.pl
[3] http://www.planenation.com/2007/11/28/airbus/airbus-a380.htm

(a) When a plane takes off, a propelling jet of hot and cold air is produced by the engine in the opposite direction of the movement of the plane. Noise is also produced by the engines and emitted in all directions. [a]

[a] Picture By Sebaso (Own work) [CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0) or CC BY-SA 4.0 (http://creativecommons.org/licenses/by-sa/4.0)], via Wikimedia Commons



(b) Example of an acoustic wave (one sound ray is here highlighted in blue) emitted from the jet engine and running through the jet. The colors indicate here the domain where BEM discretization is applied (red surface) and the domain where FEM discretization is applied (green volume). BEM is also used on the outer surface of the FEM domain.

Figure 1.1: Modeling of a plane with the BEM-FEM coupling.

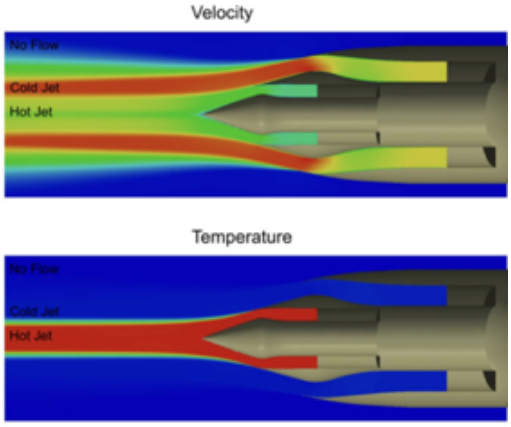


Figure 1.2: Engine based on the turbofan model. Part of the flow of air passes through the core of the engine while another part is *bypassed* into a ducted fan. The bypass stream creates differences of velocity and temperature in the jet. The jet coming from the bypass stream is cold and fast while the jet coming from the core stream is hot and a bit slower.

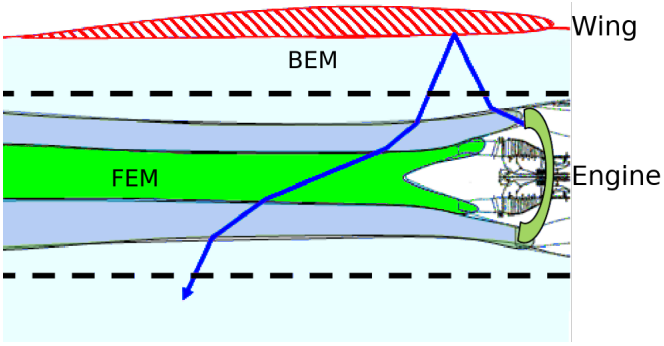


Figure 1.3: Example of an acoustic wave that propagates through multiple media. FEM is used between the dashes, i.e., on the green area (hot air), the dark blue area (cold air) and part of the light blue area (in order to obtain a simple cylinder form in a 3D context). BEM is used on the boundary of the red striped domain and the outer surface of the FEM domain.

On the study case of Fig. 1.1b, the sound wave is illustrated by a group of sound rays emitted in all directions, one of which is heading for the wing (blue ray on Fig. 1.1b and 1.3). Some rays run only through the ambient air outside the jet of exhaust gas, approximated as a homogeneous (uniform) medium, while others, like the blue ray of our study case, run, after a reflection under the wing, through the jet, consequently being deviated of their courses towards the ground. This setup is slightly more complex due to the non-homogeneous flow of hot and cold air constituting the jet, such as depicted in Fig. 1.2. Indeed, when the medium of propagation is not at rest, the convected Helmholtz equation is the simplest model for the study of trajectories of sound rays. However, two distinct domains can be identified in our setup: the interior of the jet, which is non-uniform, and the exterior medium, which is assumed to be uniform far from the jet. The convected Helmholtz equation has been reformulated in [58] to match these conditions: the classical Helmholtz equation is used in the uniform domain and an anisotropic second-order PDE in the non-uniform domain. One of the advantages of this formulation is the possibility to use the BEM which only involves integral operators for the classical Helmholtz equation on the uniform domain. On the non-uniform (interior) medium, the FEM is used to discretize the anisotropic second-order PDE. It should be noted that electromagnetic waves, governed by Maxwell's equations [131], can lead to similar systems using BEM and FEM.

These numerical methods are introduced in § 1.1.2, followed by the FEM/BEM coupling developed by Fabien Casenave [58] within the Airbus framework.

1.1.2 Numerical Approximations and Linear Systems

Numerical methods rely on the notion of discretization of a domain into smaller elements, **Finite Elements** in the case of the **FEM**, and **Boundary Elements** in the case of the **BEM**.

The **FEM** is a numerical discretization technique usually involved in the modeling of a three-dimensional space and used for solving engineering problems such as those described in the previous sections. Applications of the FEM in electromagnetics are detailed for instance in [129]. The invention of the method cannot exactly be dated (papers as early as the 1940s [124] were published on such methods), though it has originated from problems encountered in civil and aeronautics engineering. Since then, it has been studied, developed and/or extended by many researchers [50, 82, 168, 201] to provide different methods with the same core idea: a mesh discretization of the domain of interest into small cells. The space is discretized, i.e., subdivided into smaller parts called *finite elements*. In the case of the Airbus Actipole solver, the elements used to discretize a three-dimensional space are tetrahedrons. To each element is associated a local equation that approximates the original PDE, so that the method leads to a linear system of differential equations which represents the original mathematical problem. Such linear systems can be characterized by the number of *unknowns* they possess, which is related to the number of elements of the discretized mesh. Fig. 1.4b is an example of a cylinder mesh that may be used to discretize the jet of exhaust gas.

The **BEM** [33, 167, 175, 187] is closely related to the FEM, though these methods are not usually used on the same problems. BEM is applied on homogeneous media while FEM is preferred for heterogeneous media. Indeed, BEM is used when PDEs are formulated as integral equations, i.e., in boundary integral equation forms. It aims to solve a problem using only the boundary values of the physical domain, thus avoiding the need of a mesh over the entire domain. In order for this approximation to be applicable, the medium in the domain must be homogeneous. This translates as defining a mesh only over the boundaries of the considered domain and the discretization of a 3D problem using the BEM is therefore 2D. An example of such a mesh is shown in Fig. 1.4a. The linear systems arising from BEM (2D) are smaller than those arising from FEM (3D) because of the surface characteristic of their mesh. The mathematical formulation of BEM used here is further detailed in [155, 184].

In the case of Actipole, FEM is used inside the jet (green and dark blue areas in Fig. 1.1b and 1.3, with the addition of the light blue areas between the dashes to form a cylinder) because of the non-uniformity of the medium. The ambient air is approximated as being a uniform medium. Discretization in the BEM may therefore be computed over the surface of the plane. The two methods can effectively be connected using a coupling technique [56, 58], granted that the mathematical formulation of the two problems satisfy

(a) 2D surface mesh (BEM).

(b) 3D volume mesh (FEM).

Figure 1.4: Meshes on on an aircraft and a cylinder (representing the jet of exhaust gas), respectively.

the same properties on the interface between the BEM and FEM domains. In that case, the unknowns located on the boundaries of the domain discretized with the FEM are linked to the unknowns from the domain discretized using the BEM. This way, it is possible to benefit from both the volume (FEM) solution and the integral equation formulation from the BEM. As stated earlier, the classical Helmholtz equation is used in the uniform domain (discretized using the BEM) and an anisotropic second-order PDE is used in the non-uniform domain (discretized using the FEM). The transmission condition at the coupling boundary then naturally fits the boundary condition from the classical Helmholtz equation (here, a Neumann boundary condition). Dirichlet-to-Neumann maps are used for the coupling. The flow inside the jet is considered constant through time and is not disrupted by acoustic waves propagation whereas the flow in the uniform domain is discretized using larger elements and a modified frequency to adapt for the possible changes in the flow via a Lorentz transformation. For more details about this coupling and its applications to aeroacoustics, we refer the reader to [57].

Finally, the unknowns may be grouped in three main categories: (1) the unknowns associated with the formulation of FEM on the interior of the jet (heterogeneous); (2) a coupling where unknowns are shared between the BEM discretization and the FEM discretization on the outer surface of the jet; and (3) the unknowns associated with the use of BEM on the surface of the plane (Fig. 1.4a). The global algebraic system may thus

be formulated as:

$$\begin{bmatrix} A_{11} & A_{12} & 0 \\ A_{21} & A_{22} & A_{23} \\ 0 & A_{32} & A_{33} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}. \tag{1.1}$$

In practice, the unknowns from (2) and (3) are grouped together into the same surface mesh s associated with the BEM while the unknowns from (1) are grouped together into a volume mesh v associated with the FEM.

Therefore the entire system can be formulated using a 2×2 block matrix, as follows:

$$\begin{bmatrix} A_{vv} & A_{vs} \\ A_{sv} & A_{ss} \end{bmatrix} \times \begin{bmatrix} x_v \\ x_s \end{bmatrix} = \begin{bmatrix} b_v \\ b_s \end{bmatrix}. \tag{1.2}$$

The matrix system is composed of the four following blocks:

- A_{vv} is the action of the volume part on itself. This matrix is symmetric.
- A_{sv} represents the action of the volume part on the exterior surface. This action is calculated in the elements (tetrahedrons) in contact with the exterior surface.
- A_{vs} represents the exterior surface action on the volume part, it is the transpose of A_{sv}.
- A_{ss} is the matrix containing the action of the exterior surface on itself. This action is calculated in the outer domain (usual interactions) and in the inner domain (through the tetrahedrons).

Moreover, we define the sizes of each mesh as follows:

Definition 1.1. n_{FEM} is the number of elements in the mesh discretized with FEM. n_{BEM} is the number of elements in the mesh discretized with BEM.

We use here a frequency-domain BEM and FEM which lead to a linear system with complex entries.

1.1.3 Linear System Solution

We have presented the different linear systems arising from FEM, BEM and the FEM/BEM coupling and shown how they can be arranged into the global system of Eq. (1.2). We now discuss how this linear system is solved in the framework of this thesis. The 2×2 block system can be synthesized as a global one:

$$Ax = b.$$

The dimensions of the matrix A are chosen as $n \times n$ for this work and consequently x and b are vectors of size n. Moreover, the matrix is complex in our application, that is, $A \in \mathbb{C}^{n \times n}$.

Linear systems can be classified in two classes: *sparse linear systems* are mainly constituted of null coefficients, and *dense linear systems* are the opposite, meaning no, or hardly any, null coefficient. A dense matrix is sometimes called *full* or *fully populated*. Sparse matrices are often categorized according to the involved number of non-zeros,

noted nnz, and where $nnz \ll n^2$, whereas dense matrices are often qualified with their order n (nnz then satisfying $nnz = \mathcal{O}(n^2)$). Regarding the linear systems arising from the discretization techniques introduced above, each method leads to problems with very distinct properties. In particular they can lead to either sparse linear systems or dense linear systems. Linear systems arising from FEM in our application are categorized as sparse: each element mainly interacts with its close neighbors. On the contrary, linear systems arising from BEM are categorized as dense because of the integral formulation of the problem. Finally, the coupling leads to sparse matrices. To summarize, in the 2×2 block matrix from Eq. (1.2), A_{vv}, A_{sv} and A_{vs} are sparse matrices whereas A_{ss} is dense.

In this thesis, we consider the whole coupling system (1.2) and we study the impact of $\mathcal{H}$-Matrices on the computation of its solution. To understand the problematics of computing a solution in the case of dense and sparse problems, we discuss in this chapter methods that may be used in each case. We will detail in the following sections the methods used to find a solution for the dense problem arising from BEM in § 1.2, then their counterparts for the sparse problem arising from FEM in § 1.3, and finally, the solution of the entire system in § 1.4.

The methods used to find a solution for linear systems, whether they are sparse or dense, can be categorized in two main classes, as mentioned earlier: *direct methods* often based on a factorization of A into a product of matrices; and *iterative methods* that usually resort to matrix-vector multiplications to iterate over a sequence of approximation converging to the solution.

Before getting into more details on these methods, we first define some key terms we will rely on in the rest of this document. When solving $Ax = b$, the *condition number* $\kappa(A) = \frac{\sigma_{max}(A)}{\sigma_{min}(A)}$ [8], where $\sigma_{max}(A)$ and $\sigma_{min}(A)$ are the maximum and minimum singular values (further detailed p. 23 in Theorem 1.1) of A, indicates how much a perturbation in A or b can be amplified in the solution x, regardless of the possible error because of the finite precision calculation performed by the algorithm chosen to solve the problem. When a condition number is large ($\kappa(A) >> 1$), the solution may therefore be inaccurate and the problem is said to be *ill-conditioned*. On the contrary, if the condition number is small ($\kappa(A) = \mathcal{O}(1)$), the problem is said *well-conditioned*.

1.1.3.1 Direct Methods

Direct methods factorize the initial matrix A into a product of matrices so that the transformed system of equations is then simpler to solve. They generally follow three main steps. Firstly, the problem is analyzed and assembled into an acceptable computational (matrix-like) form. Secondly, the matrix is factorized under the form LU, LDL^H or LL^H, depending on the properties of the matrix, where L and U are lower and upper triangular matrices, respectively, and D diagonal. Thirdly, the solution is found through a sequence of triangular or diagonal system solve steps. The use of direct methods for the solution of dense and sparse problems is detailed in § 1.2.1 and § 1.3.1, respectively. One difference between the two is an additional step (reordering) during the sparse solution to take

advantage of the sparsity of the structure, to attempt preserving sparsity in the matrix factors.

Direct methods have several advantages: they are deterministic, as we usually know exactly how much calculations and memory we need in order to compute the solution; accurate in the backward error sense [122]; and, once a matrix is factorized, it can be used to efficiently solve multiple right-hand sides at a relatively low cost. If the matrix is *well-conditioned*, the accuracy of the solution is only limited by the accuracy of the machine, often referred to as *machine epsilon*, which is the distance ϵ_M from 1.0 to the next larger floating point number [122]. For example, if the solution was computed on a machine with double precision (64 bits), the machine epsilon is $\epsilon_M \approx 2^{-53} \approx 1.11 \times 10^{-16}$ [26, 99]. However, a drawback of direct methods is their large computational requirements to compute the factors, which, may be problematic for large matrices (see complexities in § 1.2.1.1). Iterative methods can alleviate this problem. Some direct methods such as the $\mathcal{H}$-Matrices [112] compute an approximate solution and thus require less memory and time. Such methods take advantage of the low-rank properties of some matrices or matrix blocks to find an approximation of a solution with a tolerated error. The $\mathcal{H}$-Matrix technique, on which we will rely on in Chapter 2 is detailed in § 1.2.3.

1.1.3.2 Iterative Methods

The second class of methods, iterative methods, aims to find a good approximation of the solution x, using a sequence of approximates closer and closer to x at each step of the *iterative* process. We focus here on iterative procedures where each step mainly involves a matrix-vector multiplication, which is much less costly than a matrix-matrix operation: for a matrix of size $n \times n$, a matrix-vector operation has an arithmetic complexity of $\mathcal{O}(n^2)$, while the matrix-matrix operation costs $\mathcal{O}(n^3)$ for dense problems. A sparse matrix-vector operation can be computed in $\mathcal{O}(nnz)$. Therefore, one might rather use an iterative method than a direct one if it takes only a few steps before convergence. However, one cannot predict a priori the number of steps required to reach convergence and for many problems the sequence may not converge at all [172]. Many of the following explanations come from [172]. More specifically, an iterative method is a procedure used to find an approximate solution of a system by generating a sequence of which the k-th term is calculated from the previous terms. In other words, to solve $Ax = b$, iterative methods rely on matrix-vector operations to construct a sequence x_k that converges to x. The iterative process computes n_{iter} steps with matrix-vector operations before finally be close enough to the solution and stop. To check the quality of the approximate solution, the *backward error* [189] is commonly used:

$$\frac{\|Ax_k - b\|}{\|b\|} \quad or \quad \frac{\|Ax_k - b\|}{\|A\| \cdot \|x_k\| + \|b\|}.$$

The number n_{iter} is in general not known a priori. If this number is lower than n, the iterative solution is more efficient than a direct solution (otherwise they both have an arithmetic complexity of $\mathcal{O}(n^3)$).

Iterative methods can be categorized in two main classes : stationary methods and Krylov subspace methods. The modern iterative principle has been introduced by Gauss in the early 19th century and has later been formally expressed in the mid-20th century [28, 121, 142, 197]. Since then, they have been extensively used for the solution of linear systems [88, 174].

Stationary methods are also called relaxation or fixed-point methods. Using the initial approximate solution x_0, they modify slightly the approximation until convergence is reached, based on a measurement of the error in the result (the residual). For the system $Ax = b$, the residual vector at step k is $r_k = b - Ax_k$. Then x_{k+1} is computed from x_k by $x_{k+1} = x_k + f(r_k)$, with f a linear function. Main stationary methods are the Jacobi method, the Gauss-Seidel method and Sucessive Over-Relaxation (SOR) method [197]. However these methods are not very popular anymore, at least when used alone. They are more often combined with other more efficient methods.

Krylov subspace methods form a basis of the sequence of vectors of the form $p(A)r_0$, where p is polynomial, aiming to approximate $e_0 = x - x_0 = A^{-1}r_0$ by $p(A)r_0$ with a good polynomial p, where r_0 is the residual vector equal to $b - Ax_0$. Therefore the solution is approximated by

$$x_k = x_0 + p_{k-1}(A)r_0,$$

where p_{k-1} is a polynomial of degree $k-1$ [172]. At each step of the iterative process, a Krylov subspace method computes x_{k+1} based on all the elements of the sequence $(x_j)_{j\in[0,k]}$: $x_{k+1} = f(x_k, x_{k-1}, ..., x_0)$ by means of projection. Krylov subspace methods include the Lanczos algorithm [142], the Arnoldi iteration [28], Conjugate Gradient (CG) [121], GMRES [173], GCR [80], or QMR [89]. Krylov subspace methods may be used on either dense or sparse problems [172]. Within the Airbus framework, we will rely on GMRES and Block GCR.

Yet, while they need less memory than their direct method counterpart, a drawback of these methods is the possible slow convergence rate. To prevent a slow convergence rate (or divergence), preconditioners may be used to lower the number of iterative steps (n_{iter}), meaning a faster convergence for an iterative method. The quality of an iterative solver is often determined by its preconditioner.

In the iterative procedure, linear systems involving the preconditioner P will need to be solved at each step. Hence, the properties of a good preconditioner can be summarized by:

- P is a good approximation of A;
- The cost of the construction of P is small;
- $Pz = r$ is easier to solve than the original system.

When P has been computed, it can be applied either on the right or on the left side of the original matrix A. A left preconditioner leads to the preconditioned system:

$$P^{-1}Ax = P^{-1}b.$$

A right preconditioner leads to the system:

$$AP^{-1}u = b,\ x \equiv P^{-1}u.$$

Finally, it is possible to precondition the system from both the left and the right. If the preconditioner is of the factored form

$$P = P_L P_R,$$

where P_L and P_R are for example triangular matrices, then the system can be formulated as:

$$P_L^{-1} A P_R^{-1} u = P_L^{-1} b, \; x \equiv P_R^{-1} u.$$

In the case of Actipole, a right preconditioner is used [12, § 5.1.5]. The technique used to find a preconditioner is the SPAI (SParse Approximate Inverse) [13, 55, 54, 184]. $\mathcal{H}$-Matrices can be used as preconditioners [35, 107]; however they are currently used only as direct solvers within the Airbus framework.

1.2 Dense Linear Systems

As discussed above, in the case of BEM, the plane surface is discretized based on the surface mesh shown in Fig. 1.4a (see § 1.1.2). We consider exclusively this mesh in this section, disregarding the mesh associated with the FEM. From the global linear system described in Eq. (1.2), we thus focus here on the subsystem associated with A_{ss}, which corresponds to the interactions of the surface unknowns from the surface mesh with themselves. Consequently, the linear system considered in this section is of the form

$$A_{ss} x_s = b'_s. \tag{1.3}$$

The exact formulation of b'_s will be detailed in § 1.4. For now, we assume it is a given right-hand side. BEM leads to dense linear systems (see § 1.1.3). The focus of this section is therefore the solution of a dense linear system such as Eq. (1.3) , formulated as

$$Ax = b$$

in the rest of this section for the sake of convenience. Such a system is characterized by the order n of the dense matrix A. The usual storage for such a dense matrix corresponds to n^2 coefficients.

For dense linear operations, we usually rely on the BLAS (Basic Linear Algebra Subprograms) interface [74], which has become a standard for dense linear algebra. The first version of BLAS is dated of 1979. This specification prescribes many low-level routines for vector-vector (level 1), matrix-vector (level 2) and matrix-matrix (level 3) operations. LAPACK (Linear Algebra PACK) [4] [26, 41] is mostly based on BLAS-3 and provides routines of a higher level such as LU factorization. Larger blocks may provide more efficient operations through the use of BLAS 3 [74] and may therefore be preferred. This will have consequences on the discussion in Chapter 2.

[4] LAPACK has been introduced in 1992

Let us now recall there are two classes of methods to solve linear systems: direct methods and iterative methods. We give an overview of the algorithms involved in direct methods applied on dense problems in § 1.2.1. Regarding iterative methods, the involved principles aforementioned remain essentially the same for a dense solution or a sparse solution. However, the matrix-vector product is quite costly for dense problems and a method called the Fast Multipole Method (FMM) may be used to speed up this product. The FMM is briefly introduced in § 1.2.2 for completeness but is not central in this thesis. Eventually, $\mathcal{H}$-Matrices rely on local compression of submatrices to lower direct methods computational requirements. They are introduced in § 1.2.3.

1.2.1 Direct Methods

For a dense problem with n unknowns, the cost of computing the LU factorization of a dense matrix A is $\mathcal{O}(n^3)$. Though it has been shown that this complexity can be lowered to $\mathcal{O}(n^\gamma)$, with $\gamma < 2.376$ [63, 183], we consider the complexity of a dense LU factorization to be $\mathcal{O}(n^3)$ in this thesis for the sake of clarity. As explained in § 1.1.3.1, direct methods are based on the factorization of a matrix to transform the system solution into two triangular system solutions that are easier to solve. For example with the LU factorization, the system $Ax = b$ becomes $LUx = b$, whose solution can be decomposed as follows:

$$Ly = b \; ; \quad Ux = y.$$

The solutions of these triangular systems are then largely less costly than the factorization, the complexity of a triangular solutions being $\mathcal{O}(n^2)$ compared to the complexity of $\mathcal{O}(n^3)$ of the factorization. Some factorization algorithms will be detailed in § 1.2.1.1 (for instance the LU factorization), followed by the solution of the resulting triangular systems, detailed in § 1.2.1.2.

1.2.1.1 Factorization Techniques

The factorization of a matrix A into a product of matrices having a desired structure (e.g., LU decomposition) or numerical properties (e.g., QR decomposition) is often called decomposition. An example of such a decomposition is the Cholesky or LL^H decomposition, applicable to symmetric matrices. This decomposition is based on the works of André-Louis Cholesky published posthumously (as he was killed on the battlefield in the first World War) by a fellow officer in 1924 [178]. This procedure can be viewed as the matrix form of Gaussian elimination, said to be the most natural way for solving simultaneous linear equations, and of which an example may be found in an over-2000 years old source [108]. The generalization of the algorithm to unsymmetrical matrices, the LU factorization, was introduced by Banachiewicz in 1938. Many variants have since then appeared, like the LDL^H decomposition for complex Hermitian matrices or LDL^T (usually) for real symmetric matrices [52] or the QR decomposition. An example of an implementation of the LU, QR and Cholesky factorizations can be found in the LAPACK library [26]. Even though M^H reduces to M^T if M is real, our application

may involve symmetric complex matrices (not necessarily Hermitian) so we can use the LDL^T decomposition on complex matrices. Factorization methods may lead to products of different forms and are to be used on matrices with specific properties. Some of them are listed below, followed by the required specifications of the matrix for the method to be applicable:

- LL^H (or Cholesky) decomposition: this method decomposes a matrix $A \in \mathbb{C}^{n\times n}$ that is Hermitian ($A = A^H$) positive definite into a product of the form LL^H;
- LL^T decomposition: related to the Cholesky decomposition, it may be applied on a symmetric matrix $A \in \mathbb{C}^{n\times n}$;
- LDL^H decomposition: applicable to any Hermitian matrix $A \in \mathbb{C}^{n\times n}$. The matrix is decomposed as the product of three matrices, L a lower unit triangular matrix, D a diagonal matrix, and L^H the conjugate transpose of L. The Cholesky decomposition can be seen as a special case of this decomposition, where D is the identity matrix;
- LDL^T decomposition: applicable to any symmetric matrix $A \in \mathbb{C}^{n\times n}$. This is the same algorithm as LDL^H, except the conjugate transpose is replaced by a transpose;
- LU factorization: a general matrix $A \in \mathbb{C}^{n\times n}$ is decomposed into a product of two triangular matrices L and U. L is a lower triangular matrix and U an upper triangular matrix;
- QR decomposition: with this method, any non-square matrix $A \in \mathbb{C}^{m\times n}$ (with $m \geq n$) can be decomposed intro a product of Q, an orthonormal matrix, and R, an upper triangular matrix.

Depending on the properties of the matrix and the problem under consideration, the factorization will be computed using one or the other decomposition. For example, any matrix of dimensions $m \times n$ can be factorized using QR decomposition. However, the cost of this method is greater than the cost of LU factorization so the latter may be preferred for acceptable problems like square matrices. Table 1.1 gives a quick overview of the complexities of each decomposition for square matrices. The QR decomposition would however require $4(m^2n - mn^2 + n^3/3)$ flops on a rectangular matrix $A \in \mathbb{C}^{m\times n}$ [100]. In this work, we focus on the LU factorization, which can

Method	Flop	Storage
Cholesky	$\frac{1}{3}n^3$	$\frac{n(n+1)}{2}$
LDL^H	$\frac{1}{3}n^3$	$\frac{n(n+1)}{2} + n$
LU	$\frac{2}{3}n^3$	n^2
QR	$\frac{4}{3}n^3$	$\frac{3}{2}n^2$

Table 1.1: Theoretical complexity of factorizations for square dense problems ($m = n$). The complexity of the solution step depends on the storage of the factorization considered.

be computed for the square matrices arising from BEM. The algorithm of the LU factorization is given in Algorithm 1. The notation a_{ij} is used to describe a scalar entry

(i,j) of the matrix A to avoid confusion with future notations A_{ij} for block algorithms.

Algorithm 1: LU decomposition of a matrix A of size $n \times n$.

Function `LuFactorization`(A, n)

1 **for** $k = 1$ **to** $n - 1$ **do**
2 $\ell_{kk} \leftarrow 1$
3 **for** $i = k + 1$ **to** n **do**
4 $\ell_{ik} \leftarrow \frac{a_{ik}}{a_{kk}}$
5 $u_{ki} \leftarrow a_{ki}$
6 **for** $j = k + 1$ **to** n **do**
7 $a_{ij} \leftarrow a_{ij} - \ell_{ik} a_{kj}$

In fact, this algorithm can be computed in-place when the matrix A is no longer used by the calling algorithm: we do not need to store two extra matrices L and U, as the values of A can be overwritten by those of L and U, which leads us to Algorithm 2.

Algorithm 2: LU decomposition of a matrix A of size $n \times n$ overwritten by L and U.

Function `LuFactorization`(A, n)

1 **for** $k = 1$ **to** $n - 1$ **do**
2 **for** $i = k + 1$ **to** n **do**
3 $a_{ik} \leftarrow \frac{a_{ik}}{a_{kk}}$
4 **for** $j = k + 1$ **to** n **do**
5 $a_{ij} \leftarrow a_{ij} - a_{ik} a_{kj}$

Factorizations have first been computed in a scalar manner, entry by entry, as in the algorithms above, and later been expressed in a vectorized form, column by column or block columns by block columns. A matrix may be also decomposed into a block matrix, meaning a matrix decomposed into a list of blocks (also called tiles) as illustrated in Fig. 1.5. The matrix is decomposed into $n_{blocks} \times n_{blocks}$ blocks, each of size $n_b \times n_b$,

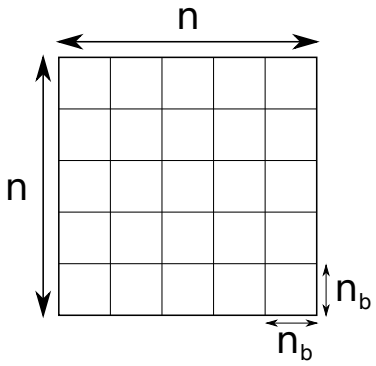


Figure 1.5: Matrix subdivided into blocks (or tiles).

so that $n_{blocks} \times n_b = n$. Factorizations can use this representation and be expressed as block algorithms and especially the LU factorization can be formulated as a block LU decomposition [71, 72]. At first, it seems n could be substituted with n_{blocks} in the original algorithm. However, replacing n by n_{blocks} is not sufficient: the scalar and block

Algorithm 3: Block LU decomposition of a matrix A with $n_{blocks} \times n_{blocks}$ blocks.

```
Function BlockLuFactorization(A, n_blocks)
1   for k = 1 to n_blocks − 1 do
2       LuFactorization(A_kk)
3       for i = k + 1 to n_blocks do
4           SolveLowerTriangular(A_kk, A_ki)
5           SolveUpperTriangular(A_kk, A_ik)
6           for j = k + 1 to n_blocks do
7               A_ij ← A_ij − A_ik A_kj
```

operations are different. A_{kk} will need to be factorized to proceed with the subsequent solutions. Therefore, the slightly different algorithm of block LU factorization is detailed in Algorithm 3, where `SolveLowerTriangular` and `SolveUpperTriangular` are methods to solve triangular systems, about which more details can be found in § 1.2.1.2. One could see the original algorithm (Algorithm 1) as a special case of the block algorithm, where each block consists in one entry only, as the LU factorization of a one-entry matrix would remain the same matrix. This block algorithm is the basis that will allow us to better understand the modified algorithm $\mathcal{H}$-LU presented in § 2.1.2.

1.2.1.2 Solution of Triangular Systems

Once the LU factorization is completed, we have two triangular systems to solve, L being a lower triangular matrix and U an upper triangular matrix; the system $LUx = b$ can be decomposed into two equations:

$$Ly = b \; ; \quad Ux = y.$$

The solution is found using the process of *forward substitution* (for the solution of the first equation) and *backward substitution* (for the second equation) in $\mathcal{O}(n^2)$. Indeed, for a lower triangular matrix of this form:

$$\begin{pmatrix} \ell_{11} & 0 & \dots & 0 & \\ \ell_{21} & \ell_{22} & \dots & 0 & \\ \vdots & & \vdots & & \vdots \\ \ell_{n1} & \ell_{n2} & \dots & \ell_{nn} & \end{pmatrix} \times \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix},$$

we substitute the solution y_1 of the first equation ($y_1 = \frac{b_1}{\ell_{11}}$) *forward* into the next equation ($\ell_{21}y_1 + \ell_{22}y_2 = b_2$) to find y_2, and so on until we have found the solution of all y_i

for $i = 1, ..., n$. Therefore, the solution can be found via Algorithm 4, with a complexity of $\mathcal{O}(n^2)$.

Algorithm 4: In-place solution of a lower triangular system. b, of size n, is overwritten by the solution.

```
Function SolveLowerTriangular(L, b, n)
1   for i = 1 to n do
2       for k = 1 to i − 1 do
3           b_i ← b_i − ℓ_ik b_k
4       b_i ← b_i / ℓ_ii
```

For upper triangular systems, substitutions are performed in a similar fashion, but this time they are done *backwards*, as the first equation solved is the last one ($u_{nn}x_n = y_n$), giving us x_n, and each solution is substituted into the previous equation. Finally, after the *forward* and *backward* solves, the solution x is computed.

In Algorithm 3, `SolveLowerTriangular`(A_{kk}, A_{ki}) finds the solution of the lower triangular system $A_{kk}X = A_{ki}$ (only the lower triangular part of A_{kk} is considered), while `SolveUpperTriangular`(A_{kk}, A_{ik}) finds the solution of the upper triangular system $XA_{kk} = A_{ik}$ (only the upper triangular part of A_{kk} is considered) using the method we have just detailed. In both cases, X replaces the value of either A_{ik} or A_{ki}.

The arithmetic and memory complexities of the solution step are equal to $\mathcal{O}(n^2)$, regardless of the factorization technique. In fact the number of operations of the solution step increases linearly with the number of entries in the factorized matrix and may, as such, be deduced from Table 1.1.

1.2.2 Fast Multipole Method (FMM)

We have seen that direct methods and iterative methods may be used to solve dense problems. While direct methods have an arithmetic complexity of $\mathcal{O}(n^3)$ for a problem with n unknowns, one might prefer to use iterative methods if the successful convergence of the algorithm leads to a lower complexity. However, the complexity of each step of the iterative method relies heavily on the matrix-vector operation. It is possible to accelerate this operation by means of the FMM. With this method, the complexity of each matrix-vector product can be reduced from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log(n))$ operations for problems arising from the BEM.

The FMM is a hierarchical method based on the approximation of distant interactions. Introduced by Rokhlin and Greengard for n-body problems [109], it has later been adapted for electromagnetism by Rokhlin [169] and Chew [181]. Details about a numerical implementation of FMM for electromagnetic problems can be found in [64, 65, 66]. In the first years of the 21st century, FMM has also been optimized and parallelized, allowing for larger problems arising from BEM to be solved using this technique. An effective survey

of its implementation and parallelization within the Airbus framework can be found in [184].

A lot of similitudes can be found between the FMM technique and the $\mathcal{H}$-Matrix technique detailed in the following section. The main difference between both formats is that $\mathcal{H}$-Matrices rely on an algebraic construction while the FMM relies on analytic arguments. This property allows a greater flexibility among the problems solvable using this technique. This is interesting for our application due to the plurality of the problems encountered in aeronautics.

1.2.3 Hierarchical Matrices for Dense Problems

Hierarchical matrices, most commonly referred to as $\mathcal{H}$-Matrices, have been introduced by Hackbusch in 1999 [112]. They were initially introduced in the context of integral equations arising from elliptic PDEs [112, 195] and have since then been further developed through many studies [34, 39, 40, 47, 48, 114], parallelized [137, 136] and can be used either as direct solvers or as preconditioners [35, 107] for iterative methods. They have later been extended to numerous formats such as $\mathcal{H}^2$ [43, 116], HSS [60, 87], HODLR [135, 141]. While all these formats have the common property of being hierarchical, they have very different structures that will be discussed in § 1.2.3.4. Later developments include directional $\mathcal{H}^2$-Matrices [42], which try to solve the problem of the growth of the ranks of submatrices for high-frequency problems using a decomposition based on plane waves, and butterfly factorizations [148], which factorize a $N \times N$ matrix into a product of $\mathcal{O}(\log(N))$ sparse matrices, each with $\mathcal{O}(N)$ non-zeros. They have also been extended to neural network structures for nonlinear problems [85] though this is out of the scope of this thesis due to the linear characteristics of the problems we are interested in. In the rest of this section, we mainly focus on the original format introduced by Hackbusch, which is also the one implemented by Benoit Lizé for Airbus [155]. The present work is based on this implementation, of which an open-source sequential version is available in [3], and many elements of this section come from [155]. For other implementations, we refer the reader to $\mathcal{H}$-Lib^{pro} [4], AHMED [1], H2Lib [5].

In the scope of solving very large dense systems of equations, one can rely on High Performance Computing (HPC) to speed-up calculations. However, the large number of operations necessary for such computations (multiplying two dense matrices of order n has a complexity of $\mathcal{O}(n^3)$) may be too large so that one has to consider $\mathcal{H}$-Matrices to speed up calculations. While an exact computation of a dense matrix-matrix product cannot be linear in the general case, with the use of $\mathcal{H}$-Matrices and its use of data compression (that may be with accuracy loss but nevertheless acceptable), the complexity can effectively be lowered to $\mathcal{O}(n \log(n))$ for those classes of matrices. This is not true for all matrices but is valid for matrices arising from standard discretizations of elliptic partial differential equations or related integral equations [114] such as the BEM. In fact, $\mathcal{H}$-Matrices provide several matrix operations with the same almost linear complexity: matrix addition and multiplication, LU factorization, or even matrix inversion. To do so, it must yield approximations through compression. Indeed, an $\mathcal{H}$-Matrix is an algebraic

hierarchical structure composed of submatrices (also called blocks) that can be stored in a low-rank or full-rank format.

We detail here the basic concepts involved in the construction of such a structure. They can be categorized in two main components:

- Low-rank matrices;
- Structured hierarchy.

Low-rank matrices imply the possibility of data compression. One of the main reasons $\mathcal{H}$-Matrices are so efficient is that some submatrices of an $\mathcal{H}$-Matrix are stored in a low-rank format. The second component, the structured hierarchy, is the skeleton over which is built the $\mathcal{H}$-Matrix. We will later see that this list can be extended in Chapter 2.

To better understand the construction of an $\mathcal{H}$-Matrix, we first detail how a low-rank matrix can be approximated in § 1.2.3.1. Then, the partition of the matrix through a hierarchy is explained in § 1.2.3.2, starting with the cluster tree structure, detailed in § 1.2.3.2.1, using the domain division explained in § 1.2.3.2.2, followed by the admissibility condition in § 1.2.3.2.3, which is used to decide whether a submatrix should be subdivided. An admissible submatrix not based on leaves of the cluster tree is stored in a low-rank format; the other leaves are stored in a dense format (see § 1.2.3.1.2). With the help of this cluster tree, a block cluster tree (§ 1.2.3.2.4) can finally be constructed. From this block cluster tree, the $\mathcal{H}$-Matrix format finally emerges. Basic hierarchical operations are also presented in § 1.2.3.3.

1.2.3.1 Low-Rank Matrices

Low-rank matrices are one of the major key ingredients of $\mathcal{H}$-Matrices as they allow information compression reducing the memory footprint and the number of computations necessary to compute operations on this class of matrices. They will be represented here with the Rank-k ($\mathcal{R}k$)-Matrix format, a format that will be available for the storage of the submatrices of the global $\mathcal{H}$-Matrix. In this work, we consider only two formats for submatrices: the $\mathcal{R}k$-Matrix format and the usual dense entry-wise format, named here Full-Matrix (short for Fully-Populated Matrix). We now detail what are the $\mathcal{R}k$-Matrices in § 1.2.3.1.2, format used to represent low-rank matrices through the use of data compression algorithms such as SVD (Singular Value Decomposition) [100] or Adaptive Cross Approximation (ACA) [101], discussed in § 1.2.3.1.1.

1.2.3.1.1 Compression

As stated earlier, compression is essential to the almost linear complexity in time and memory of $\mathcal{H}$-Matrices. It is also the tool that controls the precision obtained after the computation of a solution. We are interested here in the compression of a matrix $M \in \mathbb{C}^{m \times n}$. Several methods [134] can be used to find the $\mathcal{R}k$-Matrix representation of a matrix, with a suitable rank k satisfying the "error" specified by the user.

The best numerical approximation in 2-norm of a general matrix by a $\mathcal{R}k$-Matrix is found using the **Singular Values Decomposition (SVD)** [79], which is the most

reliable algorithm to find an approximation for a precision specified by the user. The SVD is a factorization applicable to any matrix $M \in \mathbb{C}^{m\times n}$ defined by:

Theorem 1.1. Singular Values Decomposition. *Let $M \in \mathbb{C}^{m\times n}$, there exists a factorization of M under the form*

$$M = U\Sigma V^H$$

where $U \in \mathbb{C}^{m\times m}$ is a unitary matrix, $\Sigma \in \mathbb{R}_+^{m\times n}$ is a matrix rectangular diagonal matrix with non-negative real numbers on the diagonal, and $V \in \mathbb{C}^{n\times n}$ is a unitary matrix.

The diagonal entries Σ_{ii} are known as the singular values of the matrix M, which are also the square roots of the eigenvalues of $M^H M$. The corresponding columns of U and V, respectively U_i and V_i, are called left-singular and right-singular vectors of M. When Σ is ordered according to its singular values, arranged in decreasing order, it is possible to approximate M by a new matrix called the truncated SVD, which corresponds to the matrix M but without the smallest singular values and the associated left and right singular vectors. Only the k largest singular values are kept:

$$M_k = \tilde{U}\tilde{\Sigma}\tilde{V^H} = \sum_{i=1}^{k} U_i \Sigma_{ii} V_i^H. \tag{1.4}$$

The SVD necessitates however to factorize the whole matrix, in addition to its high complexity (equal to $\mathcal{O}(mn^2 + nm^2)$), enticing us to resort to heuristics. Whereas SVD is deterministic, heuristics are however not always ensured to find a solution. An important class of heuristic methods is known as the **Adaptive Cross Approximation (ACA)**. This class of method is based on the theory developed in [101], referred to as a (pseudo-)skeleton approximation or decomposition [134]. An adaptive version of these methods has later been included into the $\mathcal{H}$-Matrix framework in [38] and further developed in [37] under the name of ACA. Since then, variants and extensions of the ACA methods, such as the total/full or partial pivoting variants or the ACA+ method, have been studied [36, 103]. The common principle of these methods is to find three matrices U, S and V so that

$$\tilde{M} = USV^T \quad , \quad \left\| \tilde{M} - M \right\|_2 \leq \epsilon, \tag{1.5}$$

with a chosen threshold $\epsilon > 0$ and $U \in \mathbb{C}^{m\times k}$, $S \in \mathbb{C}^{k\times k}$, $V \in \mathbb{C}^{n\times k}$, where $k \leq \min(m, n)$. The format is in practice reduced to

$$\tilde{M} = BC^T, \tag{1.6}$$

where $B \in \mathbb{C}^{m\times k}$, $C \in \mathbb{C}^{n\times k}$, such as given in the example of Algorithm 5.

Algorithm 5: Adaptive Cross Approximation (with total pivoting) of a matrix $M \in \mathbb{C}^{m\times n}$.

Function `ACA`$(M \in \mathbb{C}^{m\times n})$

1 $k \leftarrow 1$
2 **while** $\neg$ Criterion **do**
3 $(i^p, j^p) \leftarrow \arg\max_{i,j} |M_{ij}|$
4 $\delta \leftarrow M_{i^p j^p}$
5 **if** $\delta = 0$ **then**
6 $B \leftarrow \begin{bmatrix} b_1 & b_2 & \dots & b_{k-1} \end{bmatrix}$
7 $C \leftarrow \begin{bmatrix} c_1 & c_2 & \dots & c_{k-1} \end{bmatrix}$
8 **return** BC^T ▷ The rank if equal to $k-1$
9 **else**
10 $b_k \leftarrow M_{*j^p}$
11 $c_k \leftarrow M_{i^p *}/\delta$
12 $M \leftarrow M - b_k c_k^T$
13 $k \leftarrow k+1$
14 $B \leftarrow \begin{bmatrix} b_1 & b_2 & \dots & b_k \end{bmatrix}$
15 $C \leftarrow \begin{bmatrix} c_1 & c_2 & \dots & c_k \end{bmatrix}$
16 **return** BC^T ▷ The rank if equal to k

This algorithm details the total pivoting variant, which searches for pivot rows and columns (line 3) and constructs the solution one pivot at a time. At each iteration, the algorithm will take the row i^p and column j^p corresponding to the entry which has the maximum absolute value in the matrix M and use this row and column as the pivot to insert in the low-rank representation of the matrix BC^T. The stopping `Criterion` may be chosen either as:

- $\tilde{M}$ approximates M within the required threshold ϵ;
- a fixed rank $k = k_{max}$ has been reached.

Assuming we focus on the second criterion, the arithmetic complexity of this algorithm is then $\mathcal{O}(mn + k^2mn + kmn)$. The partial pivoting method aims to reduce this complexity to $\mathcal{O}(k^2(m+n))$. However this method is unstable and does not succeed for some specific situations [155] so it has been adapted and transformed into another variant named ACA+ (with the same complexity) to overcome such difficulties. For more information about the ACA techniques, we refer the reader to [36, 37, 38, 155].

Another method is used in the literature for finding such an approximation: the **Hybrid Cross Approximation (HCA)**, a non-heuristic method that has been proven to converge [46]. However, this method necessitates to be able to evaluate the analytical function involved in BEM (the kernel function), while the other methods introduced above are fully algebraic and do not explicitly depend on the kernel. From the industrial use meant for this application, some versatility is needed; the approach may indeed be used

in an electromagnetic as well as an aeroacoustic context, thus modifying the physics and the kernel properties. Due to these considerations, either the SVD or the ACA with Full Pivoting will be used in this thesis.

1.2.3.1.2 $\mathcal{R}k$-Matrices

This format uses compression to reduce the memory cost of storage of a matrix, as well as reducing the number of operations necessary to compute matrix operations like addition, multiplication. However, any matrix may not necessarily be represented by this format, which is one of the reasons it is used only on submatrices, as the matrix approximated using this technique must match specific criteria. Therefore a good admissibility condition is needed to determine if a submatrix matches those criteria and such a function is defined in § 1.2.3.2.3.

Let us consider a mesh of points scattered in a three-dimensional space. The concept of $\mathcal{R}k$-Matrices is based on the compression of the interactions between two clusters of points. The interactions between two clusters of size m and n are represented by $m \times n$ coefficients in the matrix arising from this problem, each coefficient (i, j) being a non-zero if there is an existing interaction between i and j. To compress this information, it is possible to reduce all the interactions of a cluster to a limited number of variables, and propagate the gathered information through a transfer function in the same manner as it is done in the case of FMM (§ 1.2.2). This is illustrated in Fig. 1.6, where all the interactions information is gathered to one point only and transferred towards the other cluster. In the formulation of the SVD in Eq. (1.4), we may identify Σ as the transfer

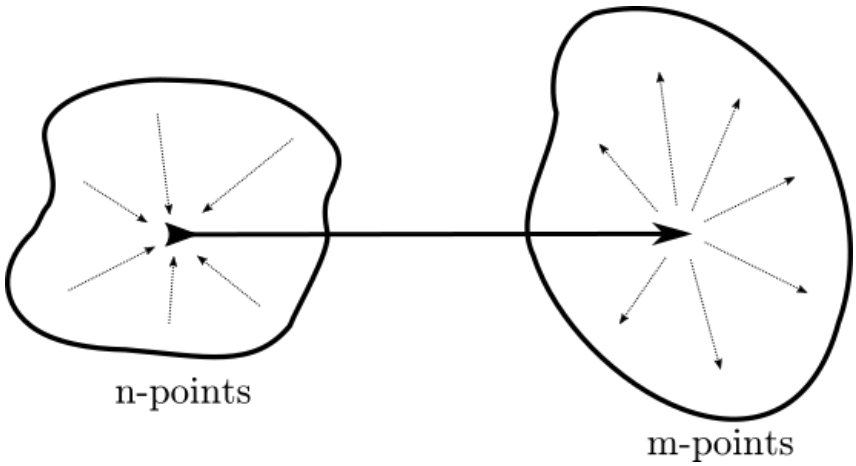


Figure 1.6: Compressed information between two clusters.

matrix as well as U and V as the gather and scatter matrices.

From the algebraic point of view, $\mathcal{R}k$-Matrices are a format using two matrix blocks to represent the original matrix. This representation of a $\mathcal{R}k$-Matrix is illustrated in Fig. 1.7.

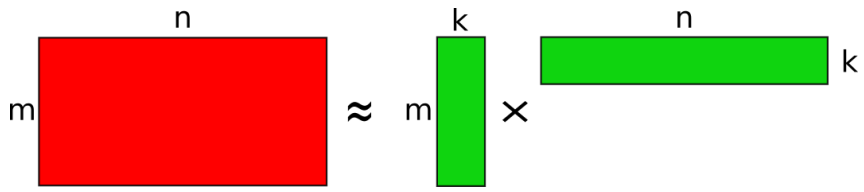


Figure 1.7: Low-rank format ($\mathcal{R}k$-Matrix). We approximate a dense matrix (red) of dimension $m \times n$ by a multiplication of two smaller matrices (green) of size $m \times k$ and $k \times n$, where k is called the rank of the $\mathcal{R}k$-Matrix.

Let $m, n \in \mathbb{N}$ and $M \in \mathbb{C}^{m \times n}$ be a matrix and let us suppose an admissibility condition is well defined. M can then be approximated with a multiplication of $B \in \mathbb{C}^{m \times k}$ and $B^T \in \mathbb{C}^{k \times n}$:

$$M \rightarrow BC^T.$$

This format is interesting only if k is small compared to m and n. k is referred to as the rank of the $\mathcal{R}k$-Matrix. Therefore, if k is small, the $\mathcal{R}k$-Matrix is defined as a low-rank matrix. The decomposition of M under a $\mathcal{R}k$-Matrix form BC^T may be computed from the SVD form in Eq. (1.4) with:

$$B = \tilde{U}\tilde{\Sigma} \quad , \quad C = \overline{\tilde{V}},$$

with $\overline{\tilde{V}}$ the conjugate of $\tilde{V}$, or, for numerical considerations [155, § 2.4.1] :

$$B = \tilde{U}\sqrt{\tilde{\Sigma}} \quad , \quad C = \overline{\sqrt{\tilde{\Sigma}}\tilde{V}},$$

where $\sqrt{\tilde{\Sigma}}$ is the diagonal matrix of which the diagonal entries are the square roots of each singular value of $\tilde{\sigma}$. Using the ACA procedure, we may use Eq. (1.6) directly, for example through Algorithm 5.

1.2.3.1.3 Operations on $\mathcal{R}k$-Matrices

Let us make a quick note on the operations on such matrices. Indeed, operations on $\mathcal{R}k$-Matrices cannot be computed through usual dense calculations. We will show a brief example regarding the addition of two $\mathcal{R}k$-Matrices. For the other operations, we refer the reader to [155]. The addition of two $m \times n$ $\mathcal{R}k$-Matrices $R_1 = B_1C_1^T$ and $R_2 = B_2C_2^T$ consist in concatenating B_1 and B_2 together and C_1 and C_2 together:

$$B_3 = [B_1B_2] \quad , \quad C_3 = [C_1C_2].$$

However, this might mean an increase in the rank of $\mathcal{R}k$-Matrices. A recompression suited for low-rank matrices can then be computed. To this end, we use two QR decompositions and a SVD [100, 155] to compute a recompression that can be decomposed into four steps. For a $\mathcal{R}k$-Matrix $R = BC^T$ of rank k, $B \in \mathbb{C}^{m \times k}$,$C \in \mathbb{C}^{n \times k}$, we get the recompressed $\mathcal{R}k$-Matrix R' using Algorithm 6.

Algorithm 6: Recompression of a $\mathcal{R}k$-Matrix $R = BC^T$.

Function `Recompression`$(R = BC^T)$

1. $B \rightarrow Q_B R_B$ ▷ Reduced QR decomposition (Definition 1.2) of B
2. $C \rightarrow Q_C R_C$ ▷ Reduced QR decomposition of C
3. $X \leftarrow R_B R_C^T$
4. $X \rightarrow \tilde{X} = \tilde{U}\tilde{\Sigma}\tilde{V}^H$ ▷ Truncated SVD of X (Eq. (1.4))
5. $B' \leftarrow Q_B \tilde{U}\sqrt{\tilde{\Sigma}}$
6. $C' \leftarrow Q_C \tilde{V}^H \sqrt{\tilde{\Sigma}}$
7. **return** $B'C'^T$ ▷ Final recompressed $\mathcal{R}k$-Matrix R' of rank k'

Definition 1.2. *Reduced QR decomposition.* A matrix $M \in \mathbb{C}^{m\times n}$, with $m \geq n$, can be decomposed as a product QR, with $Q \in \mathbb{C}^{m\times m}$ a unitary matrix and $R \in \mathbb{C}^{m\times n}$ upper triangular, i.e., a QR decomposition. Moreover, Q and R can also be partitioned:

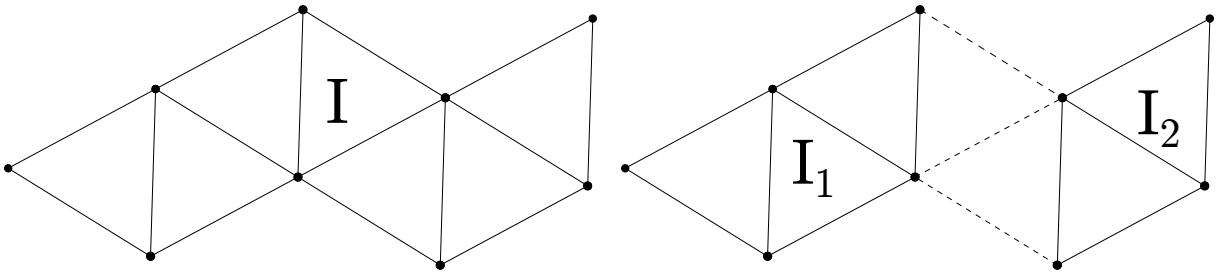
$$M = QR = Q \begin{pmatrix} R_1 \\ 0 \end{pmatrix} = \begin{pmatrix} Q_1 Q_2 \end{pmatrix} \begin{pmatrix} R_1 \\ 0 \end{pmatrix} = Q_1 R_1,$$

where $Q_1 \in \mathbb{C}^{m\times n}$ has orthonormal columns and $R_1 \in \mathbb{C}^{n\times n}$ is upper triangular with positive diagonal entries. The decomposition $Q_1 R_1$ is unique and is called the *reduced QR decomposition* or *thin QR factorization* [100, Theorem 5.2.3].

The final rank of R' is equal to k'. The (re)compression is ensured by step 4 (truncated SVD) and is done with respect to a chosen precision $\epsilon' > 0$ that can be different from the ϵ chosen for the initial compression. In practice, we consider for this work $\epsilon = \epsilon'$. The overall cost of this recompression algorithm is non-negligible : $\mathcal{O}((m+n)k^2 + k^3)$ but is reasonable for our application due to the low value of k.

1.2.3.2 Hierarchical Matrix Partition

To create an $\mathcal{H}$-Matrix, we need to partition a matrix into a list of blocks by following a block cluster tree. The first step in the construction of a block cluster tree is to order the unknowns of the problem, the initial index set. Each index (or variable or unknown) is associated with a physical vertex (also called point) of the initial mesh associated with the BEM. An example of a very simple mesh is shown in Fig. 1.8a. We consider here the sets I, representing the row indices of the considered matrix, and J, representing the column indices. The ordering may be organized in a tree called cluster tree. A cluster tree $\mathcal{T}(I)$ provides blocks of vectors for $\mathbb{R}^I$, so that a block cluster tree $\mathcal{T}(I \times J)$ relies on two cluster trees, a row cluster tree $\mathcal{T}(I)$ and a column cluster tree $\mathcal{T}(J)$ that order the index sets I and J. The construction of a cluster tree is detailed in § 1.2.3.2.1. We then introduce the concept of admissibility condition in § 1.2.3.2.3, necessary to finally obtain a block cluster tree, discussed in § 1.2.3.2.4.



(a) Whole mesh I.

(b) Subdivision of a mesh into two parts I_1 and I_2 (the edge separator is indicated by dashed edges).

Figure 1.8: Example of a two-dimensional mesh being subdivided using bisection.

1.2.3.2.1 Cluster Tree

For the construction of the cluster tree $\mathcal{T}(I)$, we restrict ourselves to the index set I. We define a cluster tree as a (most often binary but not necessarily) tree whose nodes are associated with a subset of I. The nodes of a cluster tree are often referred to as "clusters".

Each level of the cluster tree forms a partition of the index set I. For example, the root cluster contains all the indices of the set: $\tau = I$. Then, recursively, a bisection is used to find a new partition of the index set. An example of one bisection of the mesh in Fig. 1.8a is shown in Fig. 1.8b, leading to the division of the set I into I_1 and I_2. Each new index subset arising from this partition is assigned to one child of the current cluster: while there are more elements in the current set τ than a fixed threshold N_{leaf}, a domain division method is performed to create a list of children, dividing τ into subsets (two in the case of BEM).

Definition 1.3. N_{leaf} is the maximum number of elements in a leaf cluster.

As further discussed in § 1.2.3.2.2, the two main division techniques considered are the median bisection, dividing a domain into two subdomains with the same cardinality, and the geometric bisection, based on the geometry of the problem and dividing a domain into two subdomains with the same physical size and shape. The first method therefore preserves an algebraic balance between the clusters, i.e., the number of unknowns they contain, whereas the second preserves a geometric balance between the size of the clusters. The creation of a cluster tree is formally expressed in Algorithm 7, where the list of children of τ, expressed as `Children(`τ`)`, is recursively constructed. With this construction, each leaf is ensured to contain no more than N_{leaf} elements. In the case of median bisection, leaves have furthermore necessarily more than $\frac{N_{leaf}}{2}$ elements. Typically, N_{leaf} is generally set to values between 20 and 100, which is empirically a fine range in our experiments. However, the value of this variable may be discussed: for example, N_{leaf} is set to 32 in [47, 106], while it is set to 20 in [138].

Algorithm 7: Construction of a cluster tree.

```
Function CreateClusterTree(τ)
1   if |τ| ≤ N_leaf then
2       Children(τ) ← ∅                          ▷ The cluster τ will be a leaf
3   else
4       (τ_1, τ_2) ← Divide(τ)                   ▷ One of the domain division method
5       CreateClusterTree(τ_1)
6       CreateClusterTree(τ_2)
7       Children(τ) ← {τ_1, τ_2}
```

In practice, instead of copying the information about each index in every cluster it is associated with, it is stored only once, as a list of all indices $i \in I$. This is due to the fact that a parent cluster contains the union of all indices of its children. Consequently, assuming a continuous ordering of the indices in the leaves, only two informations need to be stored for each cluster: the offset of the first index of its corresponding index subset, and the size of the subset.

The exact computation of geometry properties like the diameter of a cluster of point $diam(\sigma)$, or the distance between two clusters $dist(\sigma, \tau)$ would be rather costly so that one may prefer to use an approximation of the cluster's dimensions [114, § 5.2.1]. A cluster of points is then approximated by its bounding box.

Bounding Box

A bounding box Q_σ is defined as an axis-aligned bounding box that contains all the points in the cluster σ. This box can be characterized using its two extrema points x_{max} and x_{min}. As Q_σ contains all points of the cluster σ, the diameter of the bounding box is larger than the diameter of σ and the distance between two clusters is longer than the distance between the two corresponding bounding boxes. These dimensions are indicated in Fig. 1.14 for the bounding boxes of two clusters σ and τ. The diameter of Q_σ is far much simpler to compute than the diameter of a general polyhedron associated with the points contained in the cluster σ. The dimensions can be computed in constant time [155, 3.1.4.2.3]. This is very useful for the calculations of the admissibility condition introduced in § 1.2.3.2.3. The bounding box also changes the way domain division techniques are computed, as they can consequently rely on the simplifications derived from the usage of a bounding box. This will be further developed in the following section.

1.2.3.2.2 Domain Division

The procedure of domain division is also referred to as clustering. In the case of a binary cluster tree, the axis along which is done the division is chosen as the longest

dimension of the bounding box, that is, for $x = (x^1, x^2, x^3) \in \mathbb{R}^3$, where x^d is the coordinate of x in the dimension d,

$$i^* \leftarrow \arg\max_{i=1,2,3} (x^i_{max} - x^i_{min}).$$

The unknowns are then ordered following this axis. This ensures an efficient separation into smaller cuboids, closer to cubes.

There are usually two types of domain division used in $\mathcal{H}$-Matrices: median bisection and geometric bisection. The purpose of these methods is to split a set of points into two subsets, based either on the number of points in each subset or on the geometry of the problem. Each of these methods uses the same spatial axis defined above to split a domain.

Median Bisection This method consists in dividing a set of points into two partitions with an equal cardinality, i.e., an equal number of points, (with a difference of at most one point) and is therefore also referred to as *cardinality-based* bisection. In particular, this will lead to two clusters σ and τ such that $-1 \leq |\sigma| - |\tau| \leq 1$. Following the ordering along the axis i^* previously chosen, a plane is chosen as the median of the initial set so that the number of points on each side of the division are equal:

$$x^{i^*}_{sep} \leftarrow median\left((x^{i^*}_i)_{i=1,\dots,|\tau|}\right).$$

This method results in a balanced, uniform tree (Fig. 1.10). Particularly, the height of the tree is bounded above by:

$$h(n) \leq \lfloor \log_2(n) \rfloor + 2, \tag{1.7}$$

where n is the cardinality of I. As there are n operations at each level of recursion in the tree to find the position of the division axis, the cost of its construction is $\mathcal{O}(n \log_2(n))$. Fig. 1.9 illustrates an example of median bisection, applied on a plane.

This partitioning method also ensures for each leaf of the cluster tree a maximum size of N_{leaf} , and, in addition, a minimum size of $\frac{N_{leaf}}{2}$ elements (the latter is not guaranteed by geometric bisection). However, a downside of this method is the possibility of leaves having a very large diameter if they are sparsely populated compared to other denser clusters.

Geometric Bisection This method consists in dividing a domain in two partitions with the same diameter, the cut made along the direction i^* (the same as for median bisection), halving the bounding box in the direction of the longest dimension:

$$x^{i^*}_{sep} \leftarrow \frac{1}{2}\left(\max_{j=1,\dots,|\tau|} x^{i^*}_j - \min_{j=1,\dots,|\tau|} x^{i^*}_j\right),$$

This method results in an unbalanced, possible non uniform tree (Fig. 1.11). The construction of a tree through this method can be compared to the construction of FMM,

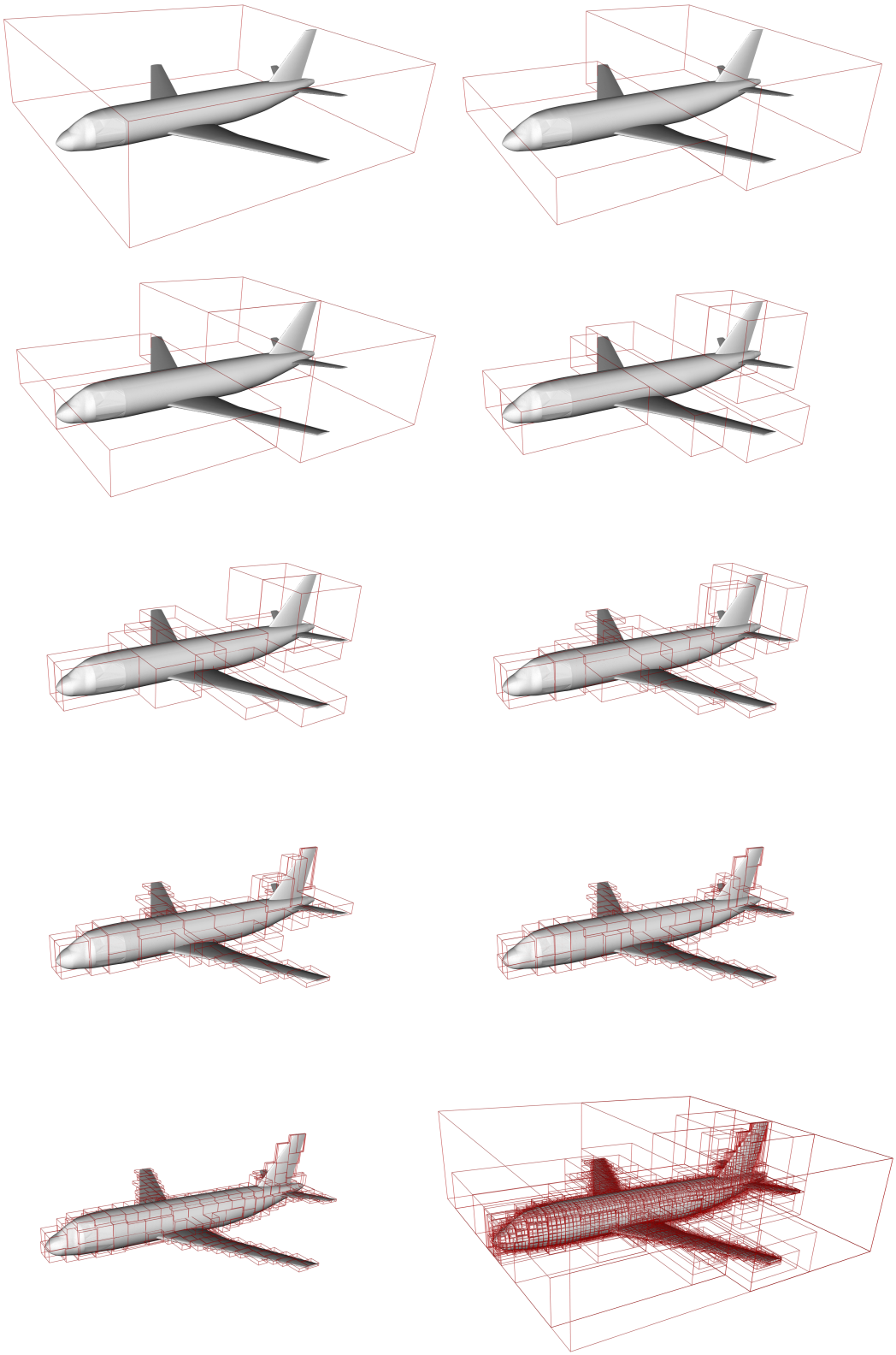


Figure 1.9: Median bisection applied on a plane mesh.

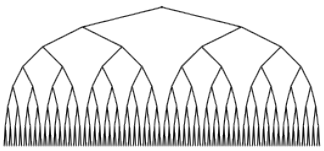


Figure 1.10: Tree based on median bisection.

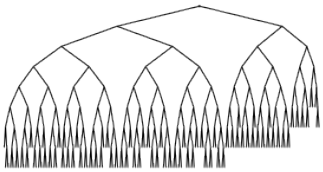


Figure 1.11: Tree based on geometric bisection.

as FMM geometrically divides a space using the middle of each dimension of a problem, leading to a quadtree (each vertex has four children) in a two-dimensional space and to an octree (each vertex has eight children) in a three-dimensional space, the general structure of a FMM tree being 2^d for d dimensions. A possible problem is the creation of clusters containing only one point if there are only one point located in one half of the physical domain (potentially located at the extreme border of the bounding box used in the formula so that no optimization is possible) and all the other points are located in the other half. The worst case scenario happens if each division leads to such two disproportionate subdomains, one of which contains only one point, thus leading to an extremely unbalanced tree. Then the height of the tree is n and the cost of its construction cannot be bounded above by anything less than $\mathcal{O}(n^2)$, with however an average case of $\mathcal{O}(n \log_2(n))$.

However, this method ensures for the clusters to have an equivalent volume, contrary to median bisection. Fig. 1.12 illustrates an example of geometric bisection, applied on a plane.

Other Domain Divisions Other algorithms can be used to partition the mesh. An example could be a hybrid version, implemented in the Airbus framework, that uses median bisection but switches to geometric bisection if the volume of one of the two subsets is too large compared to the other one (Fig. 1.13). The recursion could divide each set into more than two subsets, for instance leading to octrees as for the FMM. Finally, one could use topological tools like SCOTCH [62] or METIS [132], which are programs for partitioning graphs. These tools are mainly used on sparse graphs and we will see how to use them in Chapter 2. An algebraic clustering method has also been introduced in [107] for sparse matrices, based on a notion of algebraic distance relying on the length of the shortest path between two nodes.

1.2.3.2.3 Admissibility Condition

The admissibility condition is defined as a binary function that decides the storage of a submatrix block (σ, τ). Either we recurse on its children ($Admissible(\sigma, \tau) = False$) or we stop the recursion and assemble the block ($Admissible(\sigma, \tau) = True$). If the block is assembled, it will also determine the format chosen for a submatrix, i.e., $\mathcal{R}k$-Matrix or

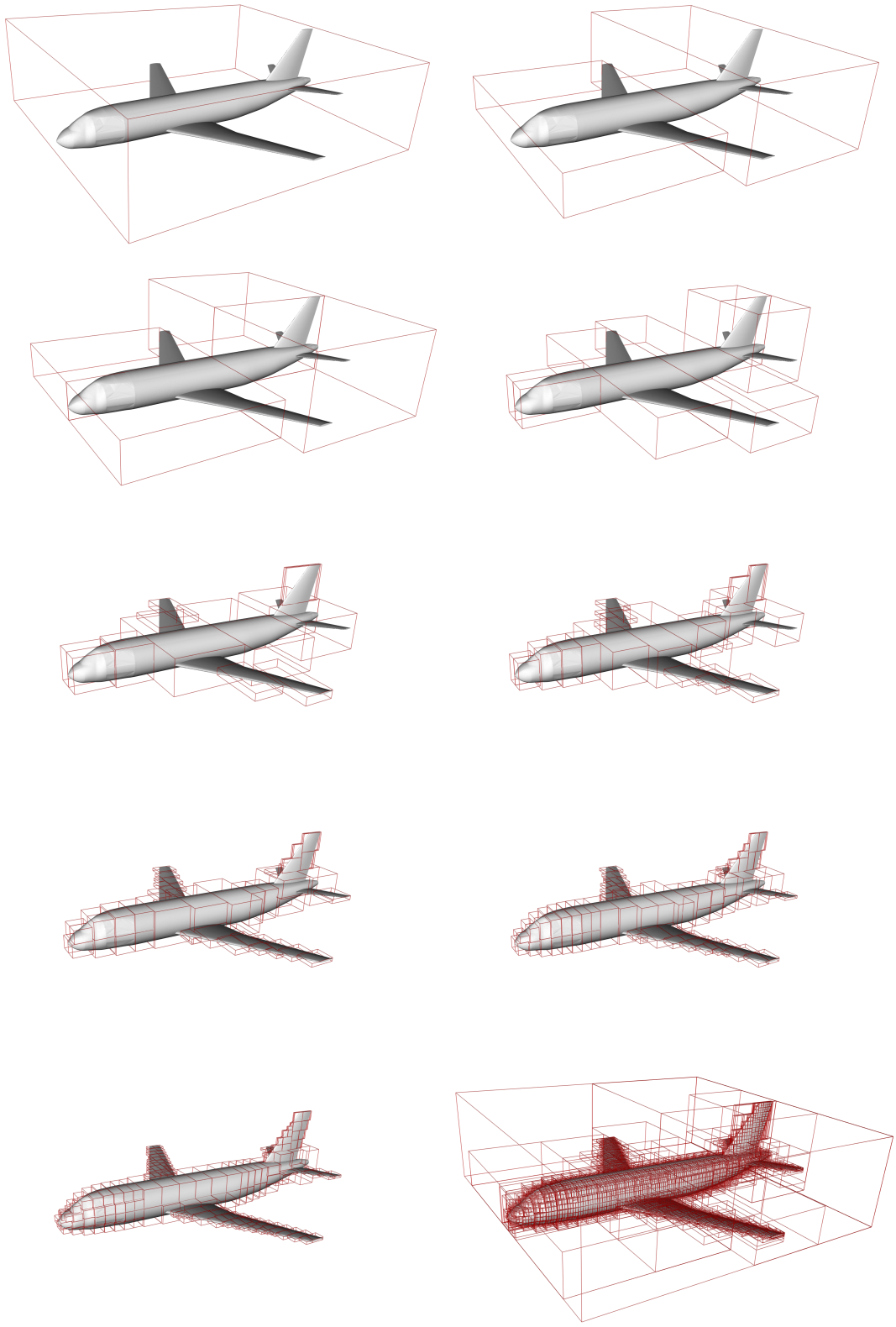


Figure 1.12: Geometric bisection applied on a plane mesh.

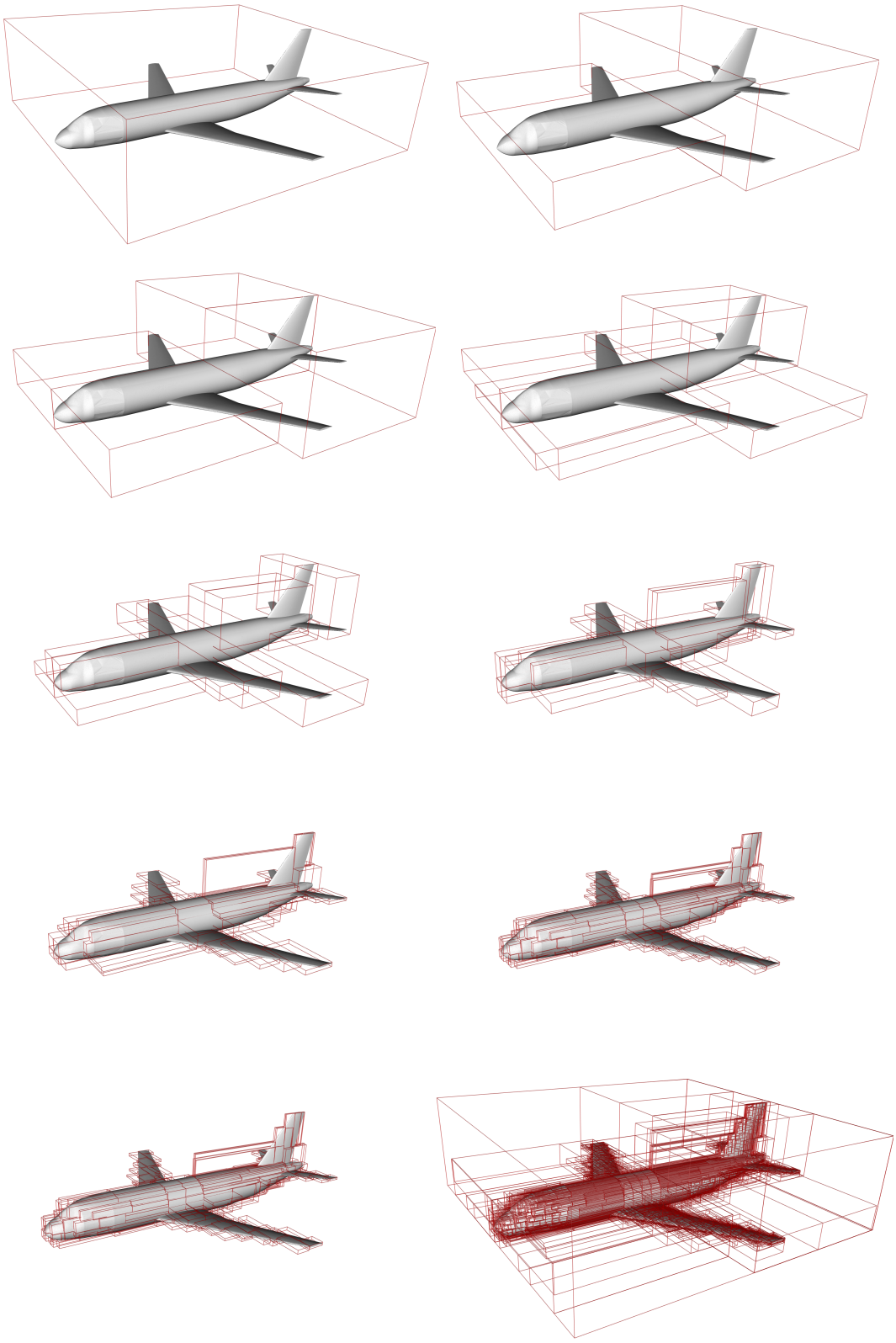


Figure 1.13: Hybrid bisection applied on a plane mesh.

Full-Matrix. The first proposed admissibility condition is based on the formulation of the integral equations of BEM [38, 45, 113].

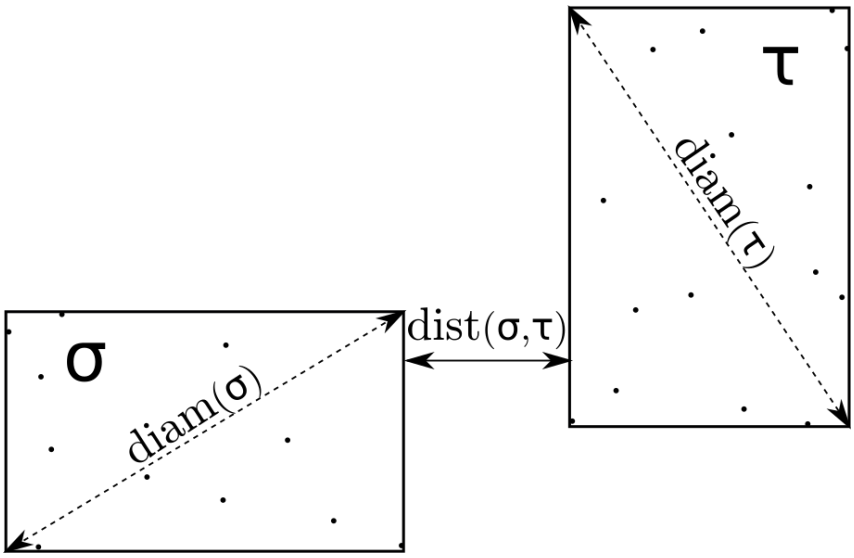


Figure 1.14: Dimensions used for the admissibility condition. If this condition is true, the interactions between σ and τ will be compressed. We can see here the diameters of each cluster and the distance between them.

Let σ and τ be two clusters and let us consider the submatrix $M|_{\sigma\times\tau}$. Then the admissibility condition of the submatrix $M|_{\sigma\times\tau}$ is defined by the function formulated in (1.8).

$$Admissible(\sigma,\tau) = \begin{cases} True & if\ \min(diam(\sigma), diam(\tau)) \leq \eta \cdot dist(\sigma,\tau), \\ False & otherwise. \end{cases} \tag{1.8}$$

The parameter η is often set to 1 or 2 in the literature. In the rest of this document, we do not investigate the impact of its value and set it to 2.

We have seen that the diameter of the bounding box (defined in § 1.2.3.2.1) is larger than the true diameter of a cluster, whereas the distance between two bounding boxes is shorter than the distance between their corresponding clusters. Therefore, if the admissibility condition applied on the bounding boxes is true, then the admissibility condition actually applied on the clusters is also true. We can thus benefit from a fast approximation of the dimensions of a cluster and compute a valid admissibility condition. Nonetheless, this also means that the condition applied on bounding boxes will ignore some admissible submatrices. The criterion we have just detailed (defined by Eq. (1.8)) is called *strong* [114, § 9.3] admissibility in opposition to the simpler *weak* admissibility defined by Eq. (1.9).

$$Admissible(\sigma,\tau) = \begin{cases} True & if\ \sigma \cap \tau = \emptyset, \\ False & otherwise. \end{cases} \tag{1.9}$$

This weak admissibility condition is the basis for the Hierarchical Off-Diagonal Low-Rank (HODLR) format (discussed in § 1.2.3.4) and represents the fact that every off-diagonal block is compressed.

It should be noted that other functions could be chosen as an admissibility condition for specific problems and that this is still an on-going investigation [199, III.C].

1.2.3.2.4 Block Cluster Tree

From the previous components we can now deduce the block cluster tree structure. The block cluster tree $\mathcal{T}(I \times J)$ is computed from the cluster trees $\mathcal{T}(I)$ and $\mathcal{T}(J)$. In the usual BEM and "level-conserving" [114] case, it may simply be expressed as the product, level by level, of cluster trees $\mathcal{T}(I)$ and $\mathcal{T}(J)$, i.e.,

$$\begin{aligned} &\forall(\sigma \times \tau) \in \mathcal{T}(I \times J) \Rightarrow \\ &\sigma \in \mathcal{T}(I) \wedge \tau \in \mathcal{T}(J) \wedge \texttt{Depth}(\sigma \times \tau) = \texttt{Depth}(\sigma) = \texttt{Depth}(\tau). \end{aligned} \tag{1.10}$$

The resulting tree is a 2×2 tree (a quadtree). Therefore, each vertex of the block cluster tree represents a pair (σ, τ) of vertices of the original cluster trees and represents a submatrix of the global matrix. For each block we have a set of rows and columns respectively defined by σ and τ. For now, each leaf block of this structure is left empty. It will be filled with an appropriate format like a $\mathcal{R}k$-Matrix or a Full-Matrix when creating the final $\mathcal{H}$-Matrix. Algorithm 8 describes the procedure to create a block cluster tree.

Algorithm 8: Construction of a block cluster tree.

```
Function CreateBlockClusterTree(σ × τ)
1   if IsLeaf(σ) ∨ IsLeaf(τ) ∨ Admissible(σ, τ) then
2       Children(σ × τ) ← ∅                                  ▷ is a leaf (σ × τ)
3   else
4       Children(σ × τ) ← {(σ', τ') | σ' ∈ Children(σ), τ' ∈ Children(τ)}   ▷
          One of the domain division method
5       for (σ', τ') ∈ Children(σ × τ) do
6           CreateBlockClusterTree(σ', τ')
```

1.2.3.3 $\mathcal{H}$-Matrix & Hierarchical Operations

Now that we have defined all the components of the hierarchical framework, we can define an $\mathcal{H}$-Matrix using the block cluster tree. This step is called *assembly*. It can be considered as the first hierarchical operation and is presented in § 1.2.3.3.1. Usual dense operations are not directly applicable to this framework. Operations like the addition, multiplication or the LU factorization must be rewritten to follow the hierarchy of the structure and are consequently recursive. The same and simple principle is used for

each operation: the algorithms are recursive until we find a satisfying condition (such as working on leaves), then, following the nature of the format used on the current block ("$\mathcal{H}$-Matrix" for non-leaves nodes, $\mathcal{R}k$-Matrix or Full-Matrix), operations of a lower level are computed, such as the addition of two dense blocks or the addition of a compressed block and a dense block.

The set of hierarchical operations constituting the $\mathcal{H}$-Matrix arithmetic can be compared to the set of operations defined by BLAS and LAPACK so that the $\mathcal{H}$-arithmetic may be referred to as $\mathcal{H}$-BLAS or $\mathcal{H}$-LAPACK. Numerous studies have been conducted to optimize this arithmetic [104, 114, 155]. We therefore detail here only two routines of both $\mathcal{H}$-BLAS and $\mathcal{H}$-LAPACK and refer an interested reader to these studies for more information on the $\mathcal{H}$-arithmetic.

1.2.3.3.1 Creation of an $\mathcal{H}$-Matrix

We mentioned earlier that the creation of an $\mathcal{H}$-Matrix could be considered as a hierarchical operation. Indeed, this function is recursive like all the other hierarchical operations, and applies a specific operation on the leaves of the block cluster tree. In the

Algorithm 9: Creation (or assembly) of an $\mathcal{H}$-Matrix.

```
Function CreateHMatrix(σ × τ)
1   if IsLeaf(σ × τ) then
2       if Admissible(σ, τ) then
3           Compress(σ × τ)                    ▷ Creation of a Rk-Matrix
4       else
5           CreateFullBlock(σ × τ)             ▷ Creation of a Full-Matrix
6   else
7       for (σ', τ') ∈ Children(σ × τ) do
8           CreateHMatrix(σ', τ')
```

case of *assembly*, the operation on each leaf is the assembly of this block, i.e., the creation of either a Full-Matrix (the usual entrywise format for dense matrices) or a $\mathcal{R}k$-Matrix (the compressed format of a low-rank matrix). Algorithm 9 shows the steps required to create an $\mathcal{H}$-Matrix. It should be noted that the decision of creating a $\mathcal{R}k$-Matrix or a Full-Matrix depends on the admissibility condition. An example of an $\mathcal{H}$-Matrix, constructed using two cluster trees based on median bisection, is shown in Fig. 1.15. Also, the $\mathcal{H}$-Matrices resulting from Fig. 1.9, 1.12 and 1.13 are shown in Fig. 1.16.

With the $\mathcal{H}$-Matrix defined, we focus now on the definition of hierarchical algorithms used to compute some of the standard operations like the addition, multiplication or factorization.

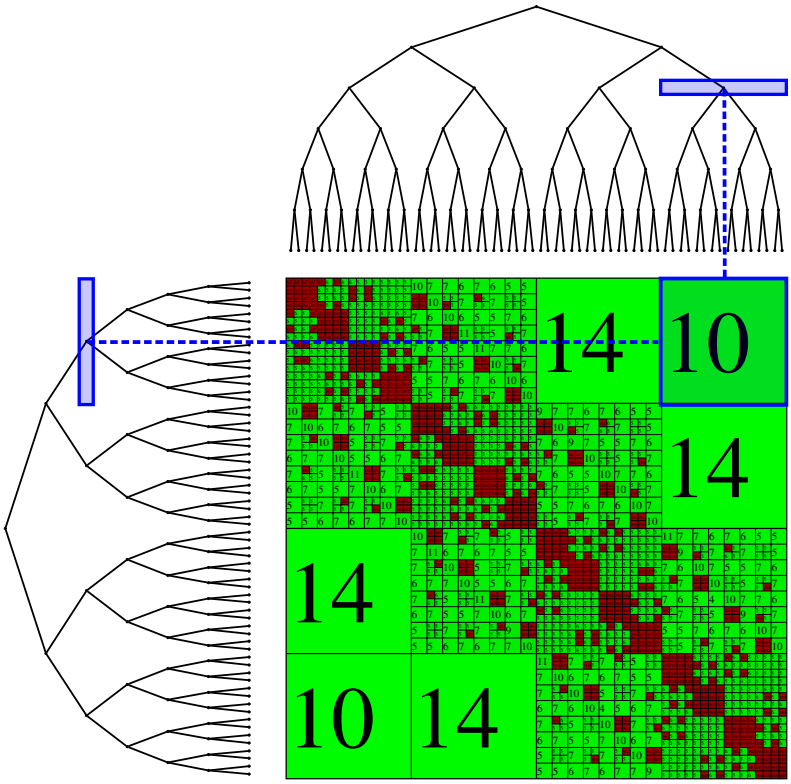
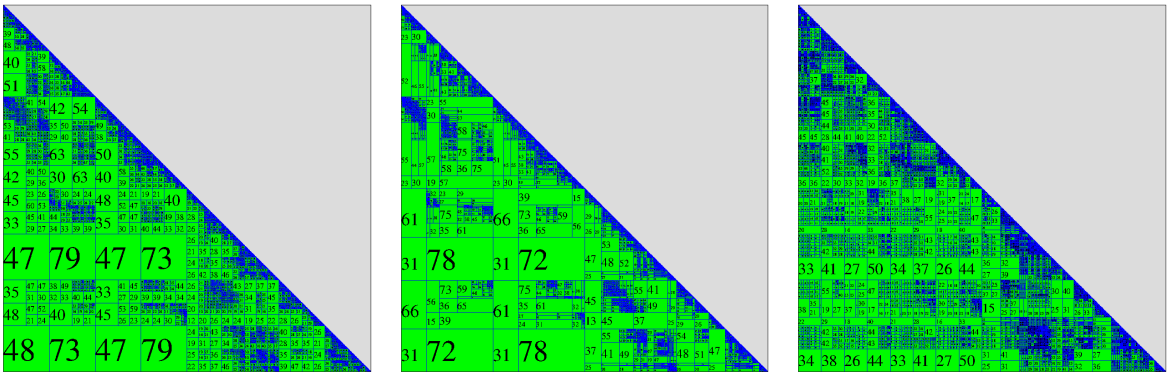


Figure 1.15: An $\mathcal{H}$-Matrix with the corresponding row and column cluster trees. On each intersection of clusters we can decide if we want to (1) recurse on both the row cluster tree and the column cluster tree, (2) compress (for example the clusters in blue rectangles that intersect to form the compressed top right block), or (3) store the block as dense if the row or column cluster tree is a leaf. Red blocks are Full-Matrices whereas green blocks are $\mathcal{R}k$-Matrices, in which the number indicates the rank.



(a) Median bisection. (b) Geometric bisection. (c) Hybrid bisection.

Figure 1.16: Examples of symmetric $\mathcal{H}$-Matrices resulting from median, geometric and hybrid clusterings.

1.2.3.3.2 Hierarchical Addition

The addition of two matrices is usually done by the routine called AXPY from BLAS. Therefore, the addition of two $\mathcal{H}$-Matrices are computed through a $\mathcal{H}$-AXPY routine, defined in Algorithm 10. This routine computes the operation $H_1 \leftarrow \alpha H_2 + H_1$, where H_1

Algorithm 10: Addition of two $\mathcal{H}$-Matrices.

```
Function H-AXPY(H_1, H_2, α, σ × τ)
1   if IsLeaf(σ × τ) then
2       if IsCompressed(σ × τ) then
3           Rk-AXPY(H_1, H_2, α, σ × τ)
4       else
5           AXPY(H_1, H_2, α, σ × τ)
6   else
7       for (σ', τ') ∈ Children(σ × τ) do
8           H-AXPY(H_1, H_2, α, σ', τ')
```

and H_2 must be based on compatible cluster trees both on rows and columns. Note that the algorithm induces that H_1 and H_2 have the same format at each step of the recursion. This is due to the fact that they share the same partition and represent the same subsystem of the overall matrix. This simplifies the algorithm to three main operations: either (1) recurse on children, (2) add two $\mathcal{R}k$-Matrices (§ 1.2.3.1.3), or (3) add two dense matrices, in which case the AXPY routine from regular dense BLAS is used.

1.2.3.3.3 Hierarchical Multiplication

The goal of the GEMM routine is to perform the multiplication of two matrices A and B and store the result into a third matrix C. This operation is more complex to handle due to the large number of cases arising from the different formats of each involved matrix (low-rank format, dense format, hierarchical format). Considering only three formats, the number of different cases to implement is already equal to $3^3 = 27$ [155]. Algorithm 11 illustrates the recursion of the operation. $\mathcal{T}_I$ represents the row cluster tree shared by A and C, $\mathcal{T}_J$ the column cluster tree shared by B and C and $\mathcal{T}_K$ the common-dimension cluster tree between A and B. σ is the current node of recursion in the cluster tree $\mathcal{T}_I$, τ the current node of recursion in the cluster tree $\mathcal{T}_J$ and ρ the current node of recursion in the cluster tree $\mathcal{T}_K$. All other 26 cases must be handled in `Leaf-GEMM`$(C, \alpha, A, B, \beta, \sigma', \tau', \rho')$, including multiplication of low-rank matrices with dense matrices, hierarchical matrices with non-hierarchical matrices for example. This operation may need to be adapted to different techniques, such as discussed in [114, §7.8.3] in the case of "non-conserving level" block cluster trees. Other details will be

Algorithm 11: Multiplication of two $\mathcal{H}$-Matrices A and B into C.

Function $\mathcal{H}$-GEMM$(C, \alpha, A, B, \beta, \sigma \in \mathcal{T}_I, \tau \in \mathcal{T}_J, \rho \in \mathcal{T}_K)$

```
if IsRoot(C) then
    C ← βC
if ¬IsLeaf(σ × τ) ∧ ¬IsLeaf(σ × ρ) ∧ ¬IsLeaf(ρ × τ) then
    for σ' ∈ Children(σ) do
        for τ' ∈ Children(τ) do
            for ρ' ∈ Children(ρ) do
                H-GEMM(C, α, A, B, 1, σ', τ', ρ')
else
    Leaf-GEMM(C, α, A, B, β, σ, τ, ρ)
```

provided in § 2.4.3.4 in the case of hierarchical constructions that do not exclusively rely on cluster trees.

1.2.3.3.4 Hierarchical LU Factorization ($\mathcal{H}$-LU)

As for usual matrices, the solution of $Ax = b$, with A being an $\mathcal{H}$-Matrix, is not usually computed through the inversion of the matrix A. The LU factorization may be used (introduced in § 1.2.1.1) to express A under the form of a product of matrices (here LU), leading to the solution of triangular systems (§ 1.2.1.2). It is computed hierarchically for the specific case of $\mathcal{H}$-Matrices. The modified algorithm is detailed in Algorithm 12 for dense problems (with the specificity of being a quadtree). The other

Algorithm 12: In-place $\mathcal{H}$-LU factorization or $\mathcal{H}$-GETRF (A is overwitten by L and U).

Function $\mathcal{H}$-LU(A)

```
if IsLeaf (A) then
    LuFactorization(A)
else
    A = [A11  A12]
        [A21  A22]
    H-LU(A11)
    H-SolveLowerTriangular(A11,A12)
    H-SolveUpperTriangular(A11,A21)
    A22 ← A22 − A21A12          ▷ Schur complement (H-GEMM)
    H-LU(A22)
```

hierarchical operations involved in this algorithm ($\mathcal{H}$-GEMM, $\mathcal{H}$-SolveLowerTriangular) are not detailed here as they are based on the same principles. We refer the reader to [112, 114, 155] for more information on these operations, and to [137, 155] for the parallelization of these algorithms. The $\mathcal{H}$-GEMM operation corresponds to the hierarchical multiplication of two $\mathcal{H}$-Matrices. $\mathcal{H}$-SolveLowerTriangular solves the lower triangular system $A_{11}X = A_{12}$, where only the lower part of A_{11} is considered and A_{12} is replaced by the new value X. This also applies to $\mathcal{H}$-SolveUpperTriangular which solves the upper triangular system $XA_{11} = A_{21}$ (only the upper part of A_{11} is considered) and A_{21} is replaced by the new value X. We will see a generalization of the $\mathcal{H}$-LU algorithm to any number of children in Chapter 2.

1.2.3.4 Other Formats related to the $\mathcal{H}$-Matrices

A lot of derived types of $\mathcal{H}$-Matrices have appeared since the first introduction of $\mathcal{H}$-Matrices in 1999. They are often categorized by their construction [14], which may use nested basis or not, a weak or a strong admissibility (leading to strong and weak hierarchies, respectively).

One of the first format to have emerged from $\mathcal{H}$-Matrices is $\mathcal{H}^2$-Matrices introduced in [117] and later extended in [43, 116]. $\mathcal{H}^2$-Matrices can also be related to the FMM as $\mathcal{H}^2$-Matrices can be seen as an algebraic version of the matrices encountered in the FMM [16]. The difference between $\mathcal{H}^2$-Matrices and $\mathcal{H}$-Matrices is their use of nested bases. The principle of nested bases methods is to add a hierarchy constraint over the computed $\mathcal{R}k$-Matrices. A matrix block is then approximated under the following form:

$$M_{\sigma\times\tau} \approx V_\sigma S_{\sigma\times\tau} V_\tau^H,$$

where $S_{\sigma\times\tau}$ is called a coupling matrix, V_σ only depends upon σ and V_τ only upon τ. V_σ and V_τ matrices are called *cluster bases* of σ and τ, respectively. Let σ' and σ'' be the children of σ. Then, V_σ can be described by the basis vector of $V_{\sigma'}$ and $V_{\sigma''}$ and the corresponding transfer matrices $T_{\sigma'\sigma}$ and $T_{\sigma''\sigma}$:

$$V_\sigma = \begin{bmatrix} V_{\sigma'}T_{\sigma'\sigma} \\ V_{\sigma''}T_{\sigma''\sigma} \end{bmatrix} = V_{\sigma'}T_{\sigma'\sigma} + V_{\sigma''}T_{\sigma''\sigma}.$$

This can be extended for more than two children. Therefore we can store cluster bases matrices V on the leaf clusters only and the transfer matrices T for all upper clusters, thus reducing even more the memory consumption of the algorithm. The double hierarchy of this format (hierarchical partition and hierarchical bases) gives the exponent 2 to the name $\mathcal{H}^2$-Matrices.

Both $\mathcal{H}$ and $\mathcal{H}^2$ matrices use the strong admissibility defined in § 1.2.3.2.3. Other formats use what is called the weak admissibility, which, paradoxically, leads to what is called a strong hierarchy or a strong low-rank structure. Among those formats the Hierarchically Semi-Separable (HSS) matrices [60, 87] are the equivalent of $\mathcal{H}^2$-Matrices: they have the same properties, except HSS-Matrices are constructed using a weak admissibility whereas $\mathcal{H}^2$-Matrices use the strong admissibility condition. This also

applies to Hierarchically Off-Diagonal Low-Rank (HODLR) matrices [10, 25, 135, 141], which are the equivalent of $\mathcal{H}$-Matrices with a weak admissibility. In those cases, further optimizations can be performed, but this is orthogonal to this thesis and we refer the reader to [15, 192] for further details.

Finally, a non-hierarchical (or single-level) format has also been introduced, namely the Block Low Rank (BLR) format [17, 18, 27, 179]. This format is based on the same principle of compression as $\mathcal{H}$-Matrices. However, the partition of the matrix is not hierarchical. This leads to simpler implementation and operations, traded for a loss in terms of complexity. When using this technique, a matrix is divided into smaller blocks of the same size, some of which are chosen as acceptable for being low-rank. [18, 157] show a comparison between complexities for a BLR approach and for hierarchical techniques, in which it appears the lower complexity of $\mathcal{H}$-Matrices may not be so significant. Especially, [188, 2.3.1] shows a lower storage and a lower number of operations for BLR compared to $\mathcal{H}$ and HSS in the case of two dense matrices arising from Poisson and Helmholtz equations. This format has also been extended in [19] to a multilevel format in an attempt to bridge the gap between the flat (BLR) and hierarchical ($\mathcal{H}$) formats. However, it should be noted that BLR has been introduced in the context of sparse matrices. Though it could be used on dense matrices, it remains essentially used to compress dense submatrices during the factorization of large sparse matrices.

Table 1.2, inspired from [14, 16, 25], lists the formats discussed above according to their characteristics. While some of these formats are simpler in terms of implementation

	BLR	FMM	$\mathcal{H}$	$\mathcal{H}^2$	HODLR	HSS
Hierarchical		✓	✓	✓	✓	✓
Algebraic	✓		✓	✓	✓	✓
Nested Basis		✓		✓		✓
Weak Admissibility					✓	✓

Table 1.2: Comparison of the characteristics of formats related to the $\mathcal{H}$-Matrix taxonomy.

and development than the original $\mathcal{H}$-Matrix, this latter format is still very much used due to its genericity and reliability, and is the format considered in the rest of this document.

1.3 Sparse Linear Systems

This section presents classic methods for the solution of sparse linear systems. It is motivated by the fact that the linear system that arises from the FEM on the jet illustrated in Fig. 1.1b, 1.3 and 1.4b is sparse. However, we must emphasize that this presentation is generic and not specific to this particular context.

Let us remind the reader that the underlying mesh is three-dimensional and each element interacts with its neighbors. Two remote unknowns are therefore usually not interacting with each other and thus the coefficient of the matrix representing this interaction is equal to zero. For this section, we take the example of a cube (Fig. 1.17)

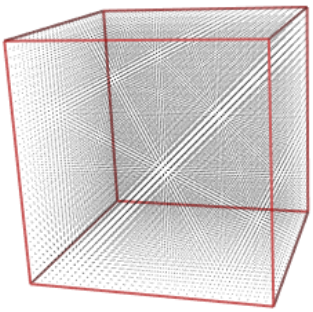


Figure 1.17: Cubic mesh over which is applied Laplace's equation with a 7-point stencil of 8000 unknowns.

with a 7-point stencil for a 3D Laplace's equation. Each unknown only interacts with its six direct neighbors. All other interactions are null.

The matrix block A_{vv} from the global system of Eq. (1.2) has already been defined as the interaction of the volume unknowns from the FEM discretization with each other. In this section, we abstract the original system of equations to a simpler form and give ourselves a new goal, which is to solve a sparse system formulated as

$$Ax = b,$$

where, consequently, $A \in \mathbb{C}^{n \times n}$ is sparse and b is a right-hand side. This matrix can thus be characterized by the number of non-zero entries it contains, denoted nnz, and x and b are both complex vectors of size n: $x, b \in \mathbb{C}^n$.

Because of the sparsity of such problems, they are not usually solved using the same techniques previously introduced in § 1.2. This section is dedicated to the presentation of sparse methods optimized to efficiently process sparse problems. For our problem $Ax = b$, we consider the number of non-zeros nnz to be sufficiently low ($nnz \ll n^2$) to seek an efficient storage in $\mathcal{O}(nnz)$ (rather than $\mathcal{O}(n^2)$). Primarily, sparse methods aim to avoid any unnecessary storage of zeros. Just as for dense problems, there are two main classes of methods to solve sparse problems: *direct methods* based on the factorization of A into a product of matrices; and *iterative methods* that resort to matrix-vector products to iterate over a sequence of approximation converging to the solution. We first present the sparse direct methods and techniques in § 1.3.1 and then clarify the iterative methods for the solution of sparse systems in § 1.3.2.

1.3.1 Sparse Direct Methods

We have discussed direct solvers in § 1.1.3.1 and how they are to be used for dense problems in § 1.2.1. However, to take the sparsity of the problem into consideration, we must rely on somewhat different algorithms. We give here an overview of techniques used to optimize computations for direct methods applied on sparse linear systems, based on [68, 69, 75, 76, 140]. For sparse systems, direct methods usually follow the four following steps:

1. Reordering the unknowns to reduce the phenomenon called fill-in that occurs during a factorization;
2. Computing a symbolic factorization that analyzes the structure of the matrix and predicts the number and the location of non-zeros in the factors;
3. Computing the numerical factorization that effectively calculates the values of the factors (LU, Cholesky, ...);
4. Computing forward/backward substitutions to finally find the solution of the initial system.

The first two steps are specific to sparse methods whereas the two others are also involved in dense methods. We consider a symmetric matrix A and therefore only the lower factor L is computed during its factorization. Nonetheless, if we want to apply a similar process for unsymmetric matrices, as, in our case, the values of L and U may be different, we may use a symmetric pattern for unsymmetric matrices. The matrix is symmetrized using the structure of $A + A^T$, i.e., zeros in A that correspond to non-zeros in $A + A^T$ are replaced by numerical zeros [23, 68]. For example, this approach is used in [23, 146]. This is the approach followed in this thesis. The reader interested in unsymmetric elimination structures can consult [81] and [97] and the references therein. All the following paragraphs may be applicable to all factorization procedures (§ 1.2.1.1) such as the Cholesky factorization, the LDL^T or the LU factorization, though the problem is considered symmetric and thus more adapted to the Cholesky or LDL^T factorizations.

We first cover the notion of fill-in and reorderings in § 1.3.1.1, followed by a brief introduction to symbolic factorizations in § 1.3.1.2. Finally, we present two classes of methods for the factorization of a sparse matrix, i.e., supernodal and multifrontal methods in § 1.3.1.3.1 and § 1.3.1.3.2, respectively.

1.3.1.1 Fill-in & Orderings

When computing the factorization of a sparse matrix, a phenomenon called fill-in occurs [91]. Fill-in is directly due to the computations involved in a factorization: in Algorithm 2, a_{ik} is divided by a_{kk}. This is the step of elimination of unknown k, related to the process of *Gaussian elimination*. The rest of the matrix then needs to be updated accordingly. This is done through the following operation:

$$a_{ij} \leftarrow a_{ij} - a_{ik}a_{kj}, \tag{1.11}$$

If a_{ij} is zero before this operation, and if entries a_{ik} and a_{kj} are not zero, k being ordered before i and j, a_{ij} will be transformed into a non-zero in the factorized matrix, i.e., *filled-in*. This update is often referred to as a *contribution*. Fill-in can be calculated for a specific entry via a simple function that provides a boolean decision, derived from the update of Eq. (1.11):

$$\texttt{Fill}(a_{ij}) = (a_{ij} \neq 0 \vee (\exists\ k < min(i,j)\ ,\ (\texttt{Fill}(a_{ik}) \wedge \texttt{Fill}(a_{kj})))). \tag{1.12}$$

If the value of Eq. (1.12) is true, the entry a_{ij} receives *contributions* from the other entries a_{ik} and a_{kj}. Therefore, if this entry was a zero and is transformed by the contribution

into a non-zero, it is *filled-in*. Besides, if a_{ik} or a_{kj} were zero at the beginning of the factorization and were filled-in at an earlier stage of the factorization, they are therefore not zero anymore at the time of the update of a_{ij}. If such is the case, the fill-in of a_{ik} or a_{kj} is *propagated* to a_{ij}. And so forth, this fill-in can induce other fill-in for larger indices. If `Fill`(a_{ij}) computes the fill-in for a scalar entry, it can also be generalized as `Fill`(A_{ij}) for block matrices, as well as for other scalar concepts developed here.

§ 1.3.1.1.1 details graph structures related to the process of elimination such as the elimination tree. In order to minimize this fill-in, we also introduce reordering techniques in § 1.3.1.1.2.

1.3.1.1.1 Elimination structures

To better apprehend the notion of fill-in and elimination, one may relate the matrix to its corresponding *adjacency graph* or *linear graph* G [161].

Theorem 1.2. *A matrix $A \in \mathbb{C}^{n\times n}$ with symmetric pattern can be associated with a graph $G = (V, E)$ with V a set of n vertices connected through edges in the set E representing the non-zero entries of matrix A. Each vertex i corresponds to an index, row or column, i of the matrix. There is an edge between i and j, noted (i, j) if, and only if, a_{ij} is not zero.*

For unsymmetric matrices, edges may be oriented and noted $i \to j$. Fig. 1.18 illustrates an example of an adjacency graph (Fig. 1.18a) and the corresponding matrix (Fig. 1.18b). Furthermore, we rely on the following definition of $\texttt{Adj}_G(i)$ to describe the list of nodes connected to vertex $i \in V$ of a graph $G = (V, E)$.

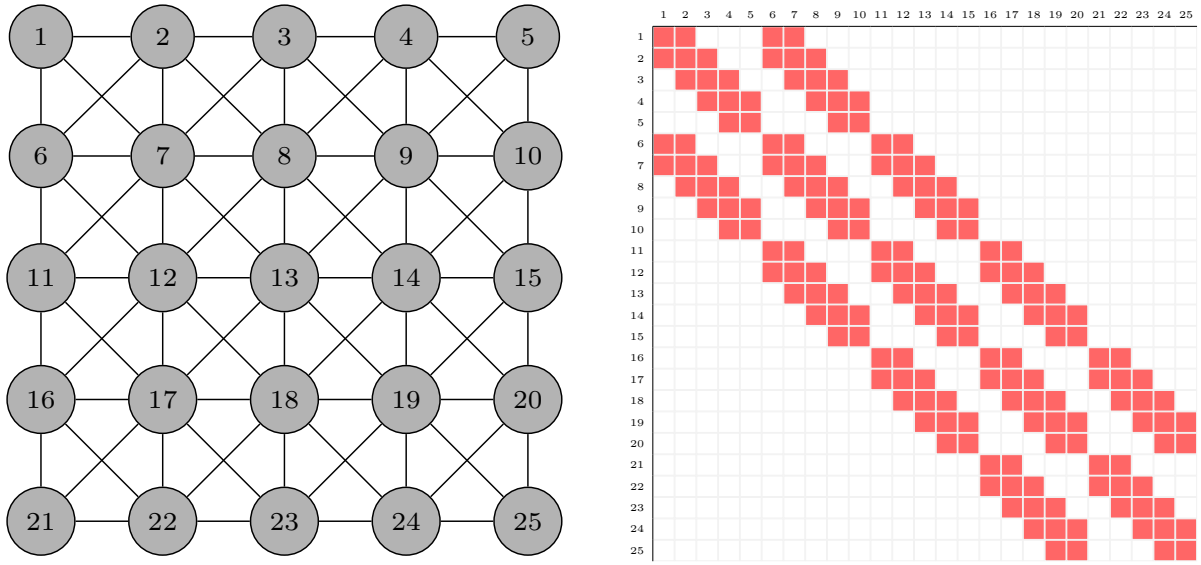
$$\texttt{Adj}_G(i) = \{j \mid \exists\ (i, j) \in E\}. \tag{1.13}$$

It should be noted that the structure of the graph G is closely related to the underlying PDE initial mesh; for instance mesh and graph connectivity are identical for the particular case of linear finite element discretizations we are interested in (see Fig. 1.19 and 1.20).

When we eliminate an unknown from the graph (at each step of the recursion in the LU factorization), its neighbor unknowns are connected together by a new edge. This is due to the update operation from Eq. (1.11). In terms of graph, the update translates as such: when we eliminate a vertex k, we check all its neighbors i and we must update or create the edge (i, j) if there is also an edge (k, j) in the graph, where k is ordered before i and j. This is illustrated by Fig. 1.21, where the new edges at each elimination represent fill-in. Here, G_i corresponds to the graph where the unknown i has been eliminated (G_0 being the original state of the graph).

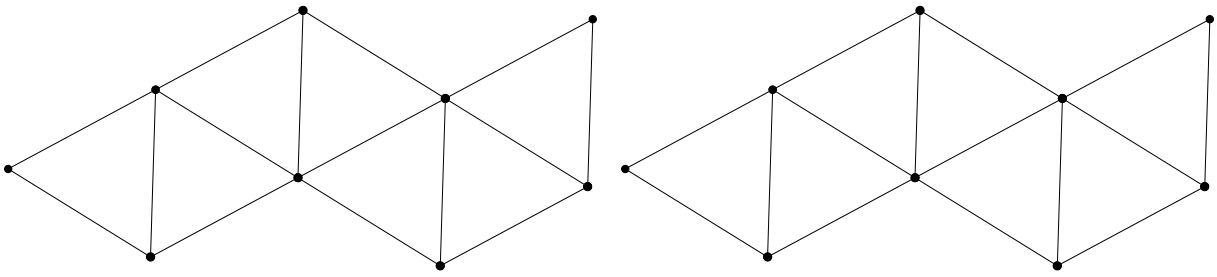
The graph associated with the factorized matrix LU is called the elimination graph [161] and noted G^*. The graph G^* has the same edges as G, with extra edges representing the fill-in. It is equivalent to the union of all graphs G_i, for $1 \leq i \leq n$. Therefore, the graph $G^* = (V, E^*)$ is associated with the factor L using Eq. (1.14).

$$\forall i, j \in V, (i, j) \in E^* \Leftrightarrow \ell_{ij} \neq 0. \tag{1.14}$$



(a) Adjacency graph (G). (b) Matrix (A).

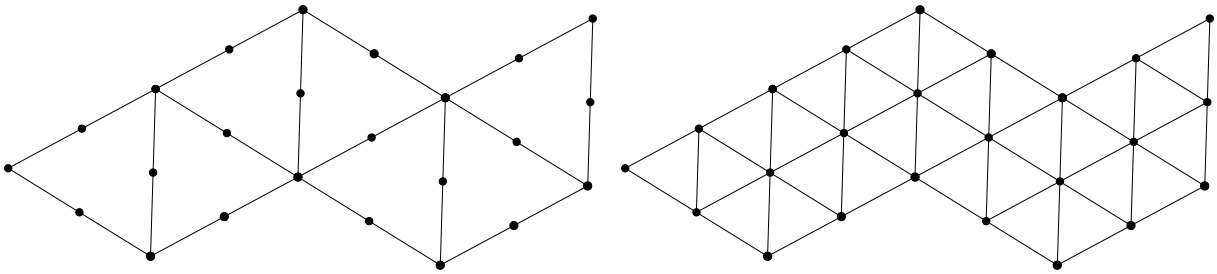
Figure 1.18: Lexical ordering of the adjacency graph (a) and the corresponding matrix pattern (b).



(a) Mesh. (b) Graph.

Figure 1.19: Example of a graph with the same structure as the mesh (e.g., P1 element in 2D).

Very simple examples of such graphs and their corresponding matrices are shown in Fig. 1.22 before and after factorization. The worst and best case scenarios illustrate how the ordering of the unknowns can impact the extent of fill-in. The factorization of the matrix and graph example in Fig. 1.18 results in the fill-in depicted in Fig. 1.23. We can see new (brown) edges in the graph (Fig. 1.23a) corresponding to the new (also brown) entries in the matrix (Fig. 1.23b). A larger case scenario, arising from the discretization of the cube example (Fig. 1.17), is illustrated in Fig. 1.24. On the left is shown the initial sparse matrix data. This is a blocked matrix. Each red block is therefore a matrix block, of which the shade is related to the percentage of non-zeros it contains (darker means more non-zeros). On the right can be noticed the fill-in that occurred in the matrix after



(a) Mesh. (b) Graph.

Figure 1.20: Example of a graph with a different structure than the mesh (e.g., P2 element in 2D).

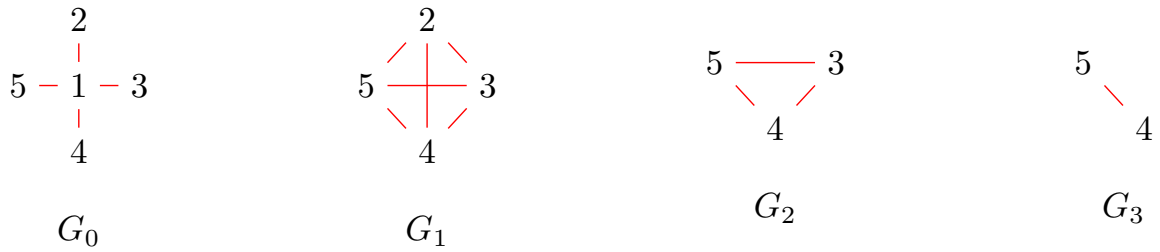


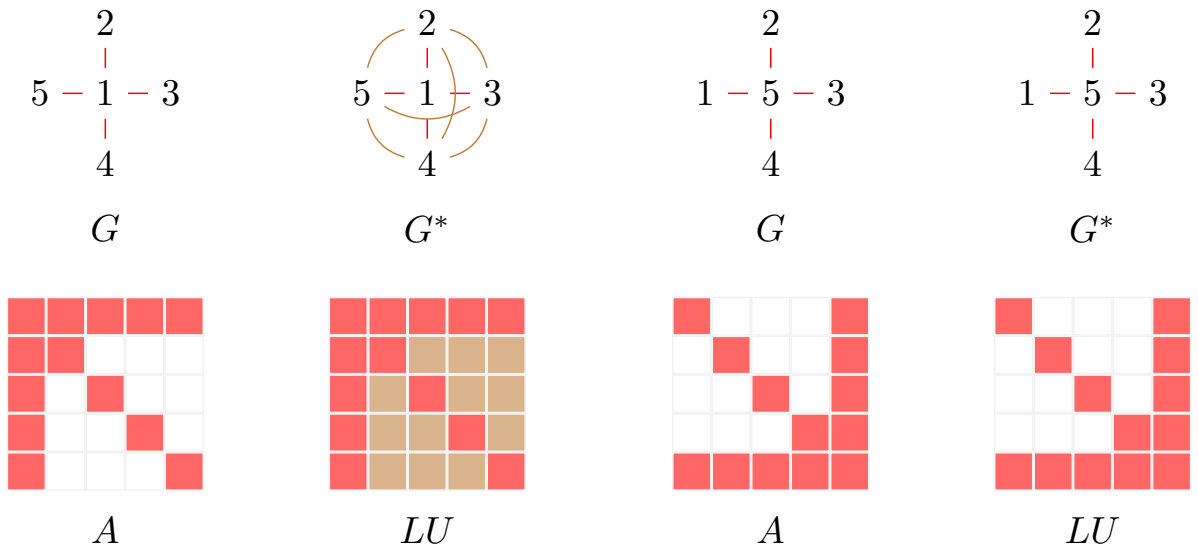
Figure 1.21: Example of eliminations of nodes of a graph and the corresponding fill-in induced, represented by the edges newly formed at each step. For example the elimination of 1 induces the fill-in between 2 and 3, 3 and 4, 4 and 5, 5 and 2, as well as 3 and 5, and 2 and 4.

factorization, which is illustrated by new red blocks or by the darker tone of existing blocks (meaning there are more non-zero entries in the block than initially).

From the elimination graph can be deduced the notion of *elimination tree*, detailed in [152, 177], by replacing the undirected edges (i,j) of the graph by directing edges $i \to j$ following the order of the elimination of each unknown and by omitting redundant edges (transitive reduction). For example, if an edge $1 \to 4$ and an edge $4 \to 5$ exist, the edge $1 \to 5$ is omitted. This tree can be represented by the structure [152] defined in Eq. (1.15). This structure essentially determines the parent of each node j of the original graph G in the elimination tree. It is noted here `EParent(`j`)` ("Elimination Parent") to differentiate it from the notion of parent used in the context of $\mathcal{H}$-Matrices.

$$\texttt{EParent}(j) = \min\{i \mid i > j \wedge \ell_{ij} \neq 0\}. \tag{1.15}$$

From the formulation of Eq. (1.15), we can directly deduce that this elimination tree describes the dependencies in the matrix computations. For example, if k is the parent of i and j in the elimination tree, then columns i and j must be computed before column k (due to $i \to k$ and $j \to k$ edges and the presence of k in both columns i and j) while columns i and j can be computed independently from each other (they belong to different subtrees).



(a) Worst case scenario: the first column, the first row and the diagonal are non-zeros (red blocks). The matrix is entirely filled-in (brown blocks in LU associated with the brown edges of G^*).

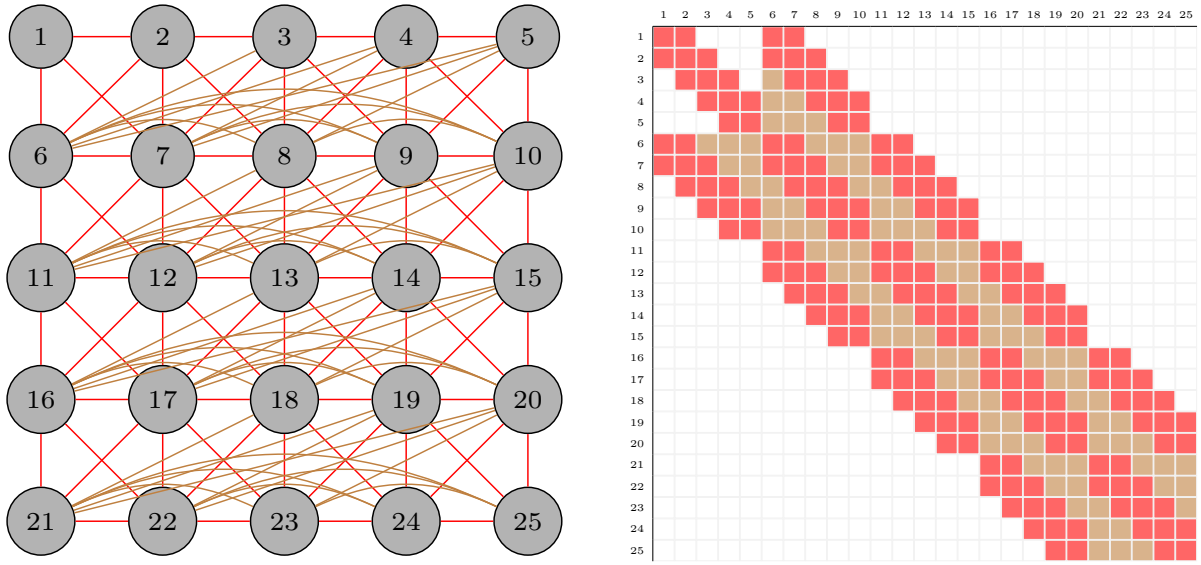
(b) Best case scenario: this ordering induces no fill-in at all.

Figure 1.22: Small example of ordering impact on fill-in. The initial matrix A is factorized into LU. G is the (adjacency) graph associated with A and G^* the (elimination) graph associated with LU.

The elimination trees corresponding to Fig. 1.22a and Fig. 1.22b are depicted in Fig. 1.25a and Fig. 1.25b, respectively. There is a relation between the height/broadness of a tree and the parallelization of the computations. The first elimination tree is the highest, each node depending on the preceding one. No parallelism is therefore possible. The other elimination tree is quite short and broad. Parallelism is exhibited for nodes 1 to 4.

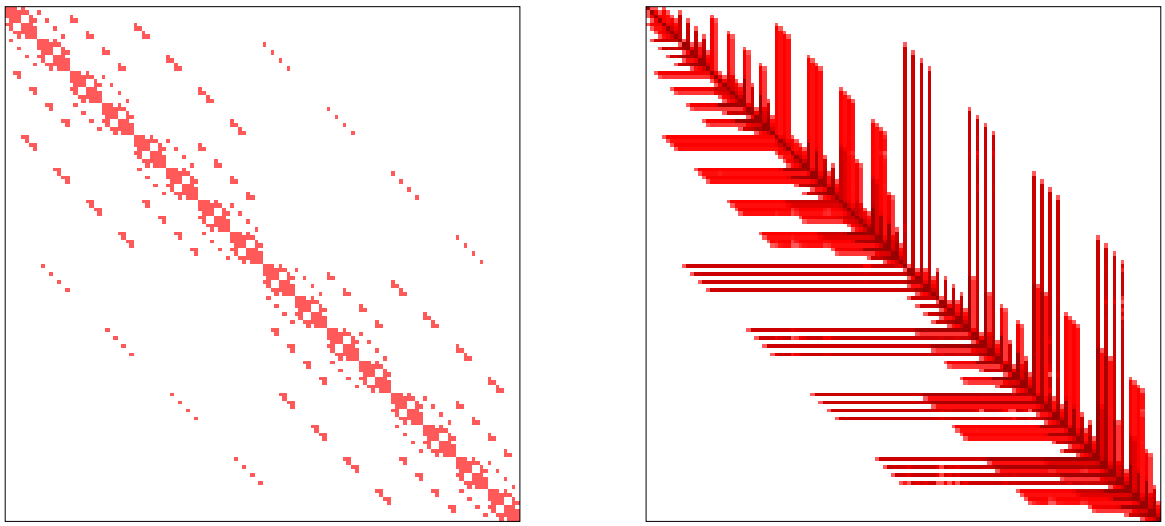
1.3.1.1.2 Ordering

The ordering of the unknowns is crucial both in terms of fill-in and parallelism (indicated by a short and broad elimination tree as stated earlier). We have seen that a specific ordering of the unknowns can change a system from the worst case scenario to Fig. 1.22a, in which the fill-in is maximized, to the best case scenario to Fig. 1.22b, where no fill-in occurs at all. The objective of reordering algorithms is to find an optimal ordering for any input matrix A that minimizes fill-in in its factorized form. This problem can be formulated as finding a permutation matrix P that induces less fill-in in the factorization of the system PAP^T than the factorization of the original input matrix A. For a matrix A, we may denote by $G^A = (V^A, E^A)$ the ordered graph following the ordering of the



(a) Elimination/filled graph (G^*). (b) Factorized matrix (LU).

Figure 1.23: Fill-in generated by the factorization of the initial example displayed in Fig. 1.18.



(a) Sparse matrix before factorization. (b) Matrix after factorization.

Figure 1.24: Block LU decomposition on a cube test case (Fig. 1.17) that results in fill-in: zeros were turned into non-zeros after factorization. The shade of red of a block indicates the percentage of non-zeros it contains. The darker blocks are the most filled with non-zero entries.

unknowns of the matrix A [93]. For simplification, we will simply refer to the reordered

(a) Elimination tree of Fig. 1.22a.

(b) Elimination tree of Fig. 1.22b.

Figure 1.25: Example of elimination trees corresponding to the matrix examples of Fig. 1.22.

graph as $G = (V, E)$. In the general case, minimizing fill-in is NP-complete [194], so heuristics are employed in practice. We introduce the two most broadly used classes of heuristics: local heuristics and global recursive approaches.

The general principle of local heuristics consist in selecting a vertex to be eliminated first and iterate on the resulting graph. The *minimum degree* algorithm is based on the choice of elimination of the unknown with the smallest degree (number of neighbors in the current adjacency graph at each step of the symbolic factorization) first. The minimum degree ordering algorithm has since then evolved [92] into computationally efficient variants, as the Approximate Minimum Degree (AMD) [20] or the Multiple Minimum Degree (MMD) [151]. The Minimum Fill algorithm [111, 159, 171] is another local approach based on the elimination first of the unknown leading to the smallest fill-in.

For large matrices, other orderings are usually preferred. Notably, *nested dissections* [90] usually provide better reorderings because they have a more global view of the graph. Nested dissection is a divide-and-conquer heuristic algorithm introduced by Alan George in 1973 in [90] based on recursive bipartitioning. We consider here the nested dissection method based on separators theorems [61, 98], of which we recall the fundamental definitions.

Definition 1.4. The class $\mathscr{S}$ of graphs satisfies the n^σ*-separator theorem* for constants $\alpha < 1$ and $\beta > 0$ if every n-vertex graph in $\mathscr{S}$ has a vertex partition $A \cup B \cup S$ such that:

- no edge has one endpoint in A and the other in B,
- $|A| \leqq \alpha n, |B| \leqq \alpha n$,
- $|S| \leqq \beta n^\sigma$.

The set S is called a *separator* and eliminated last.

Definition 1.5. Let $G = (V, E)$ be a graph in $\mathscr{S}$. G is said to admit a partition P of V in a separator tree $\mathscr{A}$ if the following property are satisfied: if G is partitioned into $A \cup B \cup S$,

- S is the root of $\mathscr{A}$,
- the two subtrees of $\mathscr{A}$ under S are separator trees corresponding to the partitions of the sets of vertices of the subgraphs A and B.

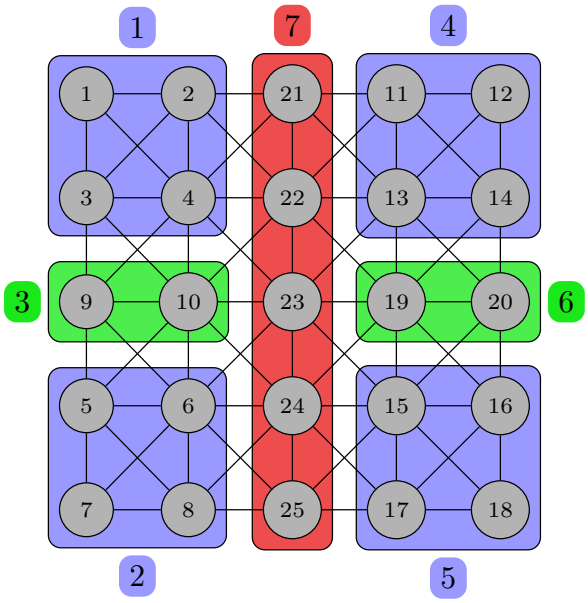
Definition 1.6. Let G be a graph in $\mathscr{S}$ partitioned into $A \cup B \cup S$ and $\mathscr{A}$ a separator tree of which the separator S is the root. An ordering following the nested dissection method based on the separator theorem must assign the larger indices to the vertices of S and the ordering of each subtrees of $\mathscr{A}$ must recursively follow the same principle.

The separators resulting from nested dissection (example in Fig. 1.26a) can be arranged into a separator tree $\mathscr{A}$ following the recursion of the algorithm, as defined by Fig. 1.26b. Fig. 1.26c shows the corresponding elimination graph and Fig. 1.26d the corresponding factorized matrix. The algorithm computes a vertex set S that separates the graph into two disconnected subsets (or subdomains) A and B, and S is ordered after A and B. Then the algorithm recursively proceeds in the same manner on both subdomains A and B until their sizes have reached a size smaller than a threshold value n_{leaf}. As the two subsets are ordered before the separator (see Definition 1.6) and are independent from each other, they are eliminated during the factorization independently from each other and the only contributions arising from their elimination are located on the separator. Therefore, an important property of nested dissection is the absence of fill-in between A and B. It should be noted that separators are preferred to be smaller in order for the size of A to be relatively close to that of B. This leads to a more balanced, broader and shorter elimination tree. The size of each leaf is smaller than n_{leaf}.

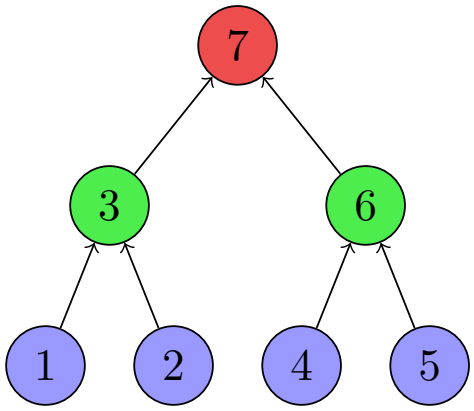
On the cube example of Fig. 1.17, the reordering computed through nested dissection, displayed in Fig. 1.27, leads to less fill-in than the lexical ordering in Fig. 1.24. One can readily see the much more located and reduced fill-in induced by the factorization in a LU form (Fig. 1.27b) of the reordered matrix of Fig. 1.27a, whereas the fill-in was much more significant in the factorized form in Fig. 1.24b of the initial matrix of Fig. 1.24a. Fig. 1.28 also shows the link between the separators found with nested dissection in the graph and the matrix. On the left, the computed separator at each step of the recursion of the algorithm is highlighted in the mesh using a new color that matches the color of the corresponding rows and columns in the matrix on the right. Only the first three levels of separators are shown. The first separator in Fig. 1.28a is colored in red and the corresponding rows and columns (Fig. 1.28b) are thus highlighted in red. The second level separators are colored in green (Fig. 1.28c) and the third level separators are shown in blue (Fig. 1.28e). The separators are on this example computed based on geometric considerations, and therefore the separators are optimal.

Finally, global orderings are often combined with local orderings [163], the coupling benefiting from the advantages of both techniques: a broader and shorter elimination tree computed with a nested dissection and a fast computation of local reordering using a minimum degree (or minimum fill) algorithm. A coupling of nested dissection and minimum fill is used for example in [119].

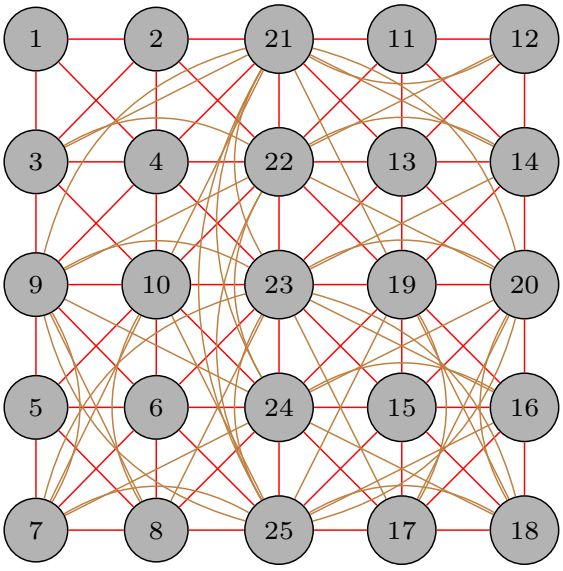
Applications do not necessarily have access to information on the geometric structure of the problem. Therefore, many topological tools have been developed to support these applications. SCOTCH [62] or METIS [132] are examples of such topological graph partitioning tools.



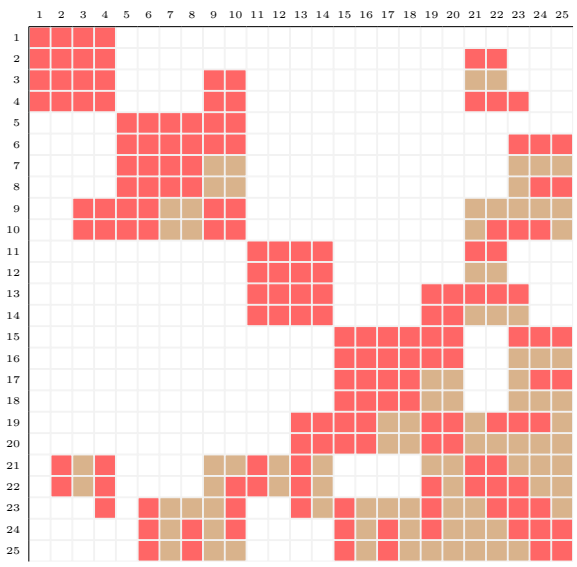
(a) Separators and new ordering computed by nested dissection. Each separator is given a number following the ordering of its unknowns.



(b) Separator tree $\mathscr{A}$.

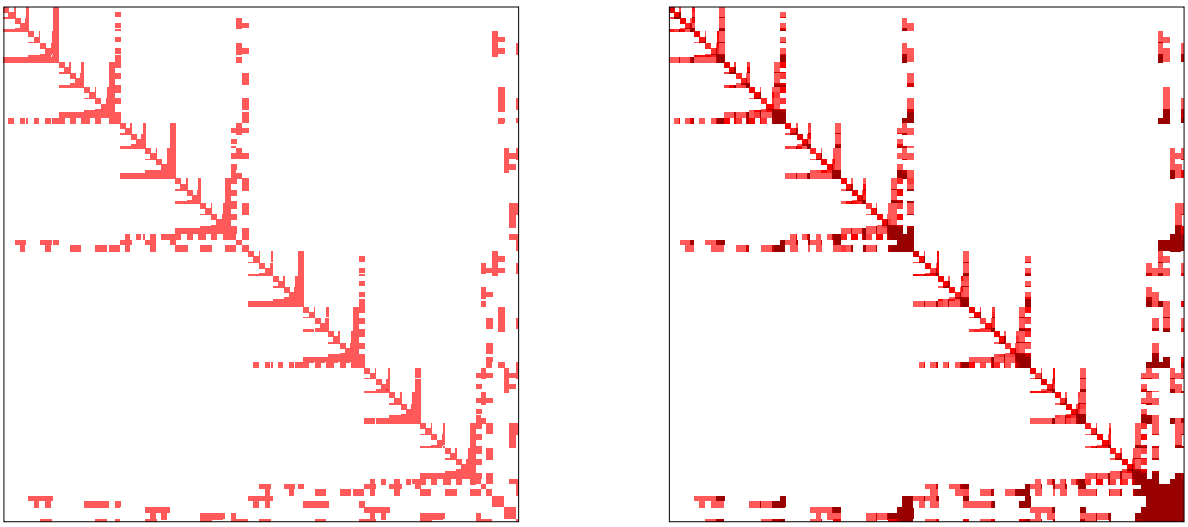


(c) Elimination graph (G^*).



(d) Factorized matrix (LU).

Figure 1.26: Nested dissection applied on the (2D) adjacency graph example from Fig. 1.18a.



(a) Sparse matrix before factorization, with nested dissection reordering.

(b) Sparse matrix after factorization, with nested dissection reordering.

Figure 1.27: Using nested dissection, fill-in induced by a LU factorization is reduced and located exclusively in the separators (either the rows or the columns must be indices inside a separator). This is to be compared to Fig. 1.24.

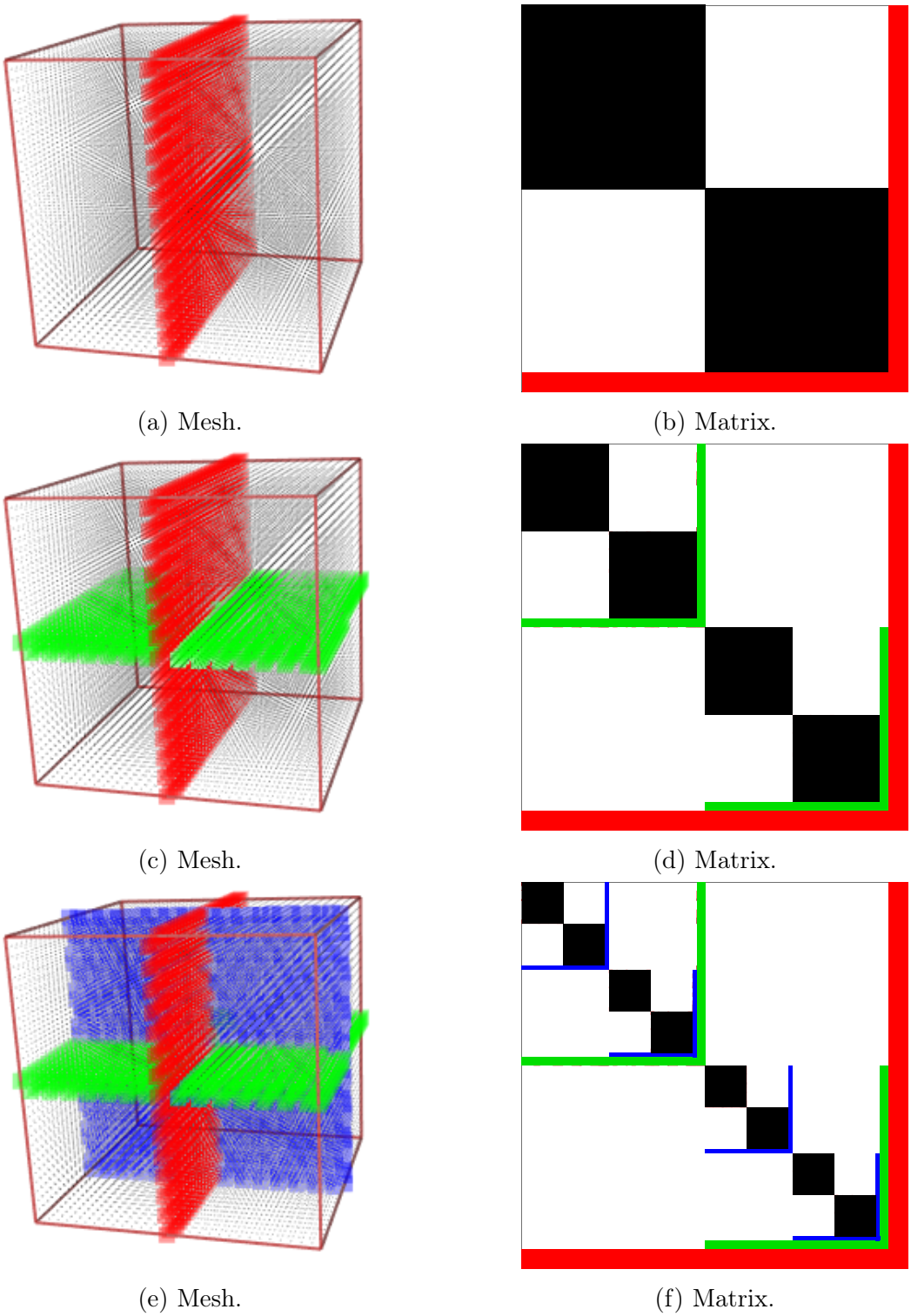
1.3.1.1.3 Supernodal Structures

For symmetric matrices, a supernode is usually defined as a range of columns of A with the same non-zeros structure [70] below the diagonal. All nodes from a supernode are eliminated together and therefore lead to a clique in the graph associated with the factors (they are all connected together). We may thus use dense submatrices to store the interactions of these supernodes. For example, if we focus on Fig. 1.23b, we can observe that columns 4 and 5 have the same pattern of non-zeros below the diagonal, as do columns 9 and 10, or columns 14 and 15, as well as columns 19 and 20. Each of these supernodes consist of two nodes. The other supernodes consist of only one node.

In the context of a reference sparse solver, SparseLU [70], supernodes are defined for unsymmetric matrices. We will use these unsymmetric supernodes to define symmetric supernodes. [70] introduces five types of supernodes.

- T1: Same row structure in U and same column structure in L;
- T2: Same column structure in L and full triangular diagonal block;
- T3: Same column structure in L and full diagonal block;
- T4: Same column structure in L and U;
- T5: Supernodes of A^TA.

T5 supernodes are sparse in the unsymmetric case. They are therefore not considered in [70]. Fig. 1.29 is an illustration of unsymmetric T1 to T4 supernodes. As we consider

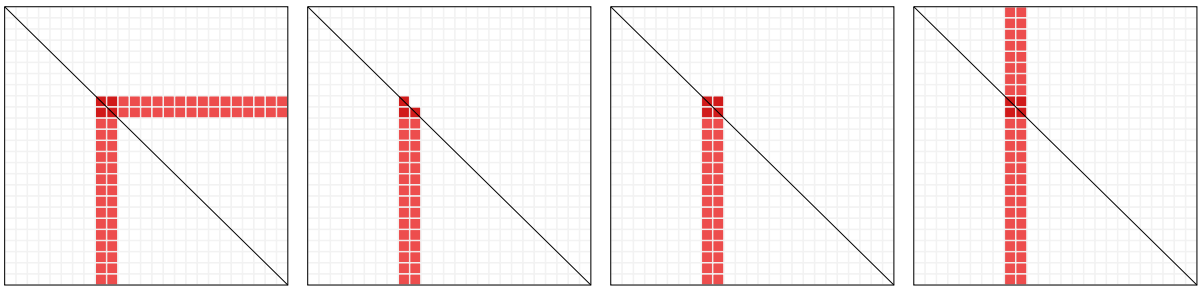


(a) Mesh. (b) Matrix.

(c) Mesh. (d) Matrix.

(e) Mesh. (f) Matrix.

Figure 1.28: 3D Laplacian equation (cube) example case showing the first three levels of separators of a geometric nested dissection algorithm.



(a) T1. (b) T2. (c) T3. (d) T4.

Figure 1.29: Possible types of unsymmetric supernodes.

matrices with a symmetric pattern, we may retain T2 and T4 supernodes, which are then re-defined as the following on the factor L:

- T2: Same column structure and full triangular diagonal block;
- T4: Same column and row structure and full triangular diagonal block.

One may indeed observe that T1 and T3 supernodes have the same structure as T2 supernodes in a symmetric paradigm. Following [70], we do not consider T5 supernodes either. The column structure in U for T4 supernodes corresponds to the row structure in L for a symmetric matrix and therefore T4 supernodes are redefined using the row structure in L. Fig. 1.30b provides a visual representation of such symmetric supernodes. In this thesis, we will primarily rely on T2 supernodes. We will consider T4 supernodes for

(a) T2. (b) T4.

Figure 1.30: Possible types of symmetric supernodes.

the concept they represent, i.e., rows with a similar pattern, as we will more extensively discuss in § 2.4.3.

Once these supernodes are defined, the matrix is partitioned in columns following the list of supernodes, noted P. Therefore, if we have N supernodes, the matrix has N block columns. The partition of the rows may be different according to the method involved. This is discussed in § 1.3.1.2 and in § 2.4.3.

The transformation of nodes into supernodes is however open to discussion. Other types of supernodes are listed in [70]. As the average size of supernodes such as defined previously is only two to three columns [70], we may compute larger artificial supernodes,

or *relaxed* supernodes [29, 70], to benefit from larger and more efficient matrix-matrix operations. However, this relaxation leads to extra memory usage, as zeros are thus introduced into the structure of each column block, and less parallelism. Examples of (relaxed) supernodes based on a nested dissection are shown in the adjacency graph in Fig. 1.26a and the corresponding matrix (partitioned in block columns) in Fig. 1.31c. The first separator is highlighted in red while the second levels separators are shown in green. Non-subdivided subdomains are colored blue.

The associated graph corresponding to the interactions between supernodes is called *quotient graph* [61], and noted G/P (notation used in this thesis) or, sometimes, $\mathcal{Q}(G,P)$. It is computed through the partition of supernodes P of the initial adjacency graph G. In other words, it is an adjacency graph applied on supernodes. Therefore, each supernode K is connected to other supernodes and its adjacency is defined as $\texttt{Adj}_{G/P}(K)$, using the quotient graph $G/P = (P, E/P)$.

$$\texttt{Adj}_{G/P}(K) = \{J \mid \exists\ (K,J) \in E/P\}. \tag{1.16}$$

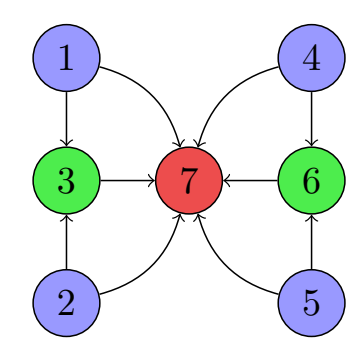
The quotient elimination graph (G^*/P) can be distinguished from the elimination quotient graph $(G/P)^*$. In the first case, we first compute the elimination of the unknowns, i.e., the operation *, and then compute the quotient graph, i.e., the operation $/P$. In the second case, we first compute the quotient graph using the list of supernodes, i.e., the operation $/P$, and then the elimination of the unknowns, i.e., the operation *. For a general matrix (and the associated graph G) and the set P of supernodes, there is no guarantee that these two processes (G^*/P) and $(G/P)^*$ lead to the same quotient graph. However, in the particular case of a nested dissection, under the connexion hypothesis formulated in Theorem 1.3 usually satisfied in practice, both processes do lead to the same result.

Theorem 1.3. *Let G be a reordered graph in $\mathscr{S}$. If for all $i = 1, N$ the separator S_i is contained into a single connected component of the subgraph G_i it separates, then $(G/P)^* = (G^*/P)$ [61].*

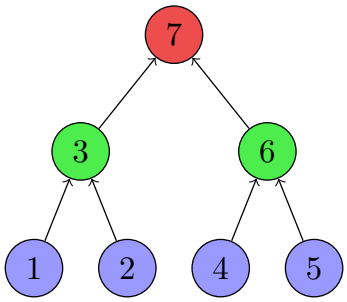
We refer to [61] for the proof. For cases satisfying Theorem 1.3 such as nested dissection, the graph $(G/P)^*$ is thus preferably computed [86, 140] in practice due to its far lower complexity. This is discussed more at length in § 1.3.1.2.3.

These notions may be defined differently in other contexts. For example, the quotient graph model presented in [93] consists in a list of quotient graphs after each elimination. Instead of deleting the eliminated node in the graph as depicted in Fig. 1.21, the *eliminated nodes* are grouped together. These are also called supernodes. But they must be distinguished from our previous definition of supernodes. In this representation, nodes may also be grouped if they are *uneliminated and indistinguishable*. Using this technique, the supernodes are therefore computed on-the-fly, instead of prior to the elimination.

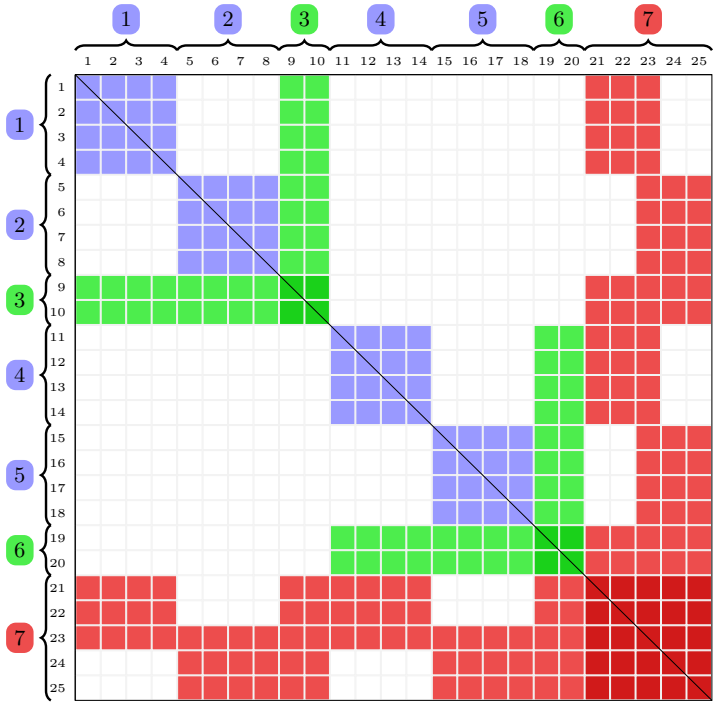
In $(G/P)^*$, the unknowns of a supernode are eliminated at the same time, so the elimination tree associated with A can be reduced to a simpler form. The transitive reduction of the elimination graph G^* leads to the elimination tree T. We can therefore



(a) Quotient graph ($(G/P)^*$).



(b) Elimination tree (T/P).



(c) Factorized matrix (LU).

Figure 1.31: Structures based on the adjacency graph and the separators (supernodes) computed in Fig. 1.26a. With these supernodes, the quotient elimination graph is computed, as illustrated in (a). The elimination (or assembly) tree (in (b)) is computed by transitive reduction. Finally, (c) shows the factorized matrix and the block column structure computed the symbolic factorization discussed in § 1.3.1.2.3. The supernodes are indicated by the braces over the lines and columns and displayed as colored numbers. Supernode 7 for example (here a separator) consists of nodes 21 to 25.

apply the same procedure on the elimination quotient graph (G^*/P) to obtain a new elimination tree (T/P). The tree that arises from this reduction is also referred to as an *assembly tree* [146], a *blocked elimination tree* [140], or a *quotient tree* [76]. It describes the dependencies between the solver's supernodes computations or contributions. This quotient tree is equal to the separator tree $\mathscr{A}$ (Fig. 1.26b) in the case of a nested dissection. An example of such a quotient tree is shown in Fig. 1.31b. However these two structures may differ when other reordering techniques are used.

1.3.1.2 Symbolic Factorization

Before computing the numerical factorization (the effective factorization), a symbolic analysis step is performed. Depending on the sparse direct method, the numerical objectives, the solver design, etc., this analysis step may have different objectives (cf. [68, § 4.6] for instance). In this manuscript, we focus on the computation of a symbolic factorization [94]. A symbolic factorization computes an explicit structure based on the

non-zero pattern of the factorized matrix. Note that a symbolic factorization has no effect on the extent of fill-in (which is governed by the ordering only) but locates where fill-in is produced. The objective of symbolic factorization, according to [146], is to *simulate the actual factorization in order to estimate the memory that has to be allocated for the factorization phase.*

To compute a symbolic factorization, we rely here on the notion of reachable sets [68, 69, 93].

Reachable sets

Consider the graph $G = (V, E)$. The structure $\texttt{Reach}(i, S)$ [68, 93] computes the set of nodes reachable from i through S, where $i \in V$ and $S \subset V$.

Definition 1.7. A node j is said to be reachable from i through S, i.e., $j \in \texttt{Reach}(i, S)$, if there exists a path $(i, s_1, s_2, ..., s_k, j)$ in G from i to j such that $s_\ell \in S$ for $1 \leq \ell \leq k$.

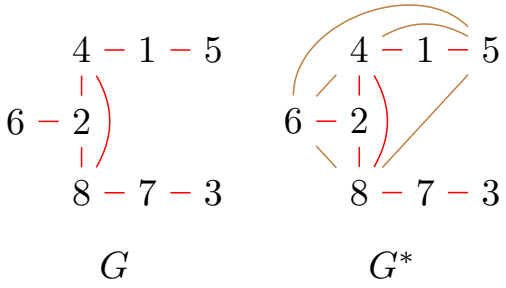
With the notation $S_k = \{1, 2, ..., k\} \subset V$, we define $\texttt{Reach}^+_G(i) = \texttt{Reach}(i, S_{i-1})$ the reachable set from i through all nodes ordered before i, following notations from [69]. It is also referred to as $Struct(L_{*i})$ in [97] or $Str(A(i : n, i))$ in [110]. From the graph point of view, at the step of elimination of i, the structure $\texttt{Reach}^+_G(i)$ represents all nodes reachable from i through the eliminated nodes in G. $\texttt{Reach}^+_G(i)$ also represents the edges of the elimination graph G^* starting from i and oriented following the ordering of the unknowns. From a matrix perspective, it represents all the row indices corresponding to non-zeros below the diagonal of the column i after factorization. From Eq. (1.14), we may now formulate Eq. (1.17).

$$i \in \texttt{Reach}^+_G(j) \Leftrightarrow \ell_{ij} \neq 0. \tag{1.17}$$

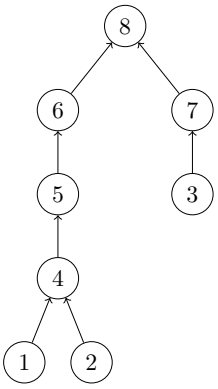
In Fig. 1.32 for example, $\texttt{Adj}_G(4) = [1, 2, 8]$ and $\texttt{Reach}^+_G(4) = [5, 6, 8]$. The state of the structure at the elimination of node 4 is depicted in Fig. 1.33.

1.3.1.2.1 Scalar Symbolic Factorization

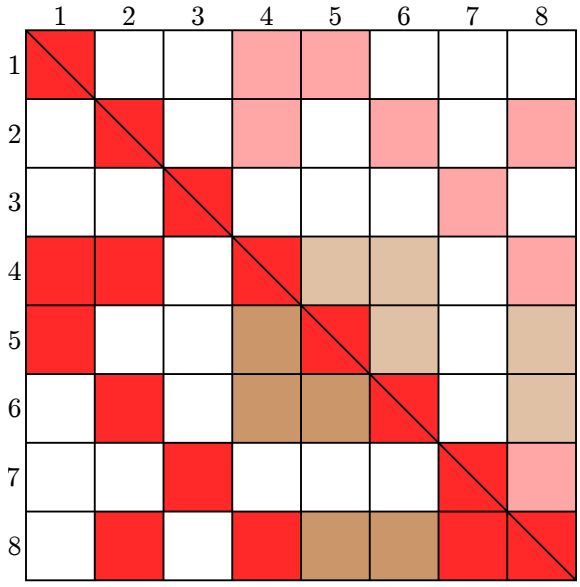
To simulate the behavior of a numerical factorization, one may rely on Algorithm 13, following the spirit of [93, p.167]. However, this algorithm has the same complexity as the numerical factorization. A possible way to improve this algorithm is to avoid some unnecessary computations that are redundant here. When an unknown k is eliminated, we can compute the fill-in induced for the unknown m_k, where m_k is defined as the minimum index in $\texttt{Reach}^+_G(k)$ (which is also the minimum row index of the non-zero blocks in the column k below the diagonal from a matrix point of view), instead of computing this for all row indices in $\texttt{Reach}^+_G(k)$. The proof is given in [93, p.168] or in the transition between Theorem 4.3 to 4.5 in [97]. As an example, using the matrix from Fig. 1.32c, Fig. 1.34a shows how the elimination of column $k = 1$ impacts column $m_k = 4$. Fig. 1.34b shows how the elimination of column $k = 2$ impacts column $m_k = 4$. 4 is the minimum row index corresponding to a non-zero in columns 1 and 2. The fill-in for larger indices in $\texttt{Reach}^+_G(1)$ can be computed later, at the stage of $k = 4$. Indeed, after the merging



(a) Adjacency graph G and elimination graph G^* with 8 nodes corresponding to Fig. 1.32c. Initial edges are colored red and fill-in edges colored brown.



(b) Elimination tree T corresponding to Fig. 1.32a.



(c) Symbolic factorization structure of a 8×8 matrix. Initial non-zeros are colored red and fill-in entries colored brown. Only the lower part of the matrix is considered in the future as the matrix is symmetric.

Figure 1.32: Example of the resulting structure of a symbolic factorization.

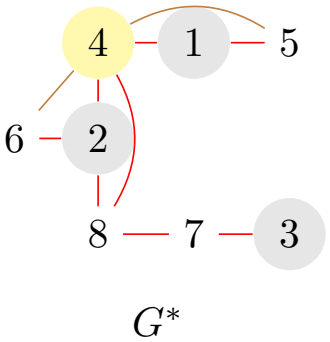
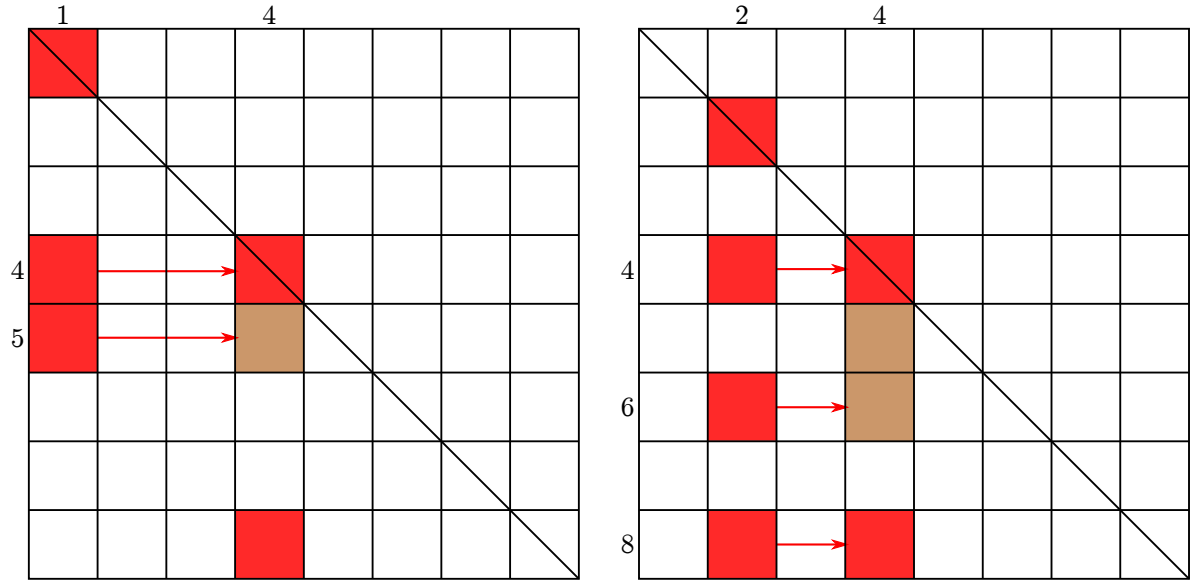


Figure 1.33: The nodes reachable from 4 through the eliminated nodes (ordered before 4 and colored gray) are 5, 6 and 8.



(a) Merging column $k = 1$ onto column $m_k = 4$.

(b) Merging column $k = 2$ onto column $m_k = 4$.

Figure 1.34: Right-looking contributions of column 1 and 2 on column 4 from example Fig. 1.32c.

of $\texttt{Reach}_G^+(1)$ and $\texttt{Reach}_G^+(4)$, all indices in $\texttt{Reach}_G^+(1)$ will also be in $\texttt{Reach}_G^+(4)$. The information of $\texttt{Reach}_G^+(1)$ is contained in $\texttt{Reach}_G^+(4)$ and can be transmitted to these larger indices. We must emphasize that we consider here a right-looking algorithm. A left-looking variant is discussed in § 1.3.1.2.4.

As mentioned earlier, this elimination can also be expressed by the elimination graph G^* (Fig. 1.32a) or the elimination tree introduced in § 1.3.1.1. When a node k (associated with the unknown k) is eliminated, it contributes to all its ascendants in the elimination tree. However, we may propagate this contribution only to its parent, i.e., the minimum adjacent node m_k ordered after k. Eq. (1.15) can also be written using the structure

Algorithm 13: Naive Scalar Symbolic Factorization of a $n \times n$ matrix A based on its adjacency graph G.

Function `ScalarSymbolicFactorization`(G, n)

1 **for** $k = 1$ **to** $n - 1$ **do**

2 $\quad \texttt{Reach}_G^+(k) \leftarrow \{j \mid j > k \wedge j \in \texttt{Adj}_G(k)\}$ ▷ Initial non-zeros

3 **for** $k = 1$ **to** $n - 1$ **do**

4 $\quad$ **for** $j \in \texttt{Reach}_G^+(k)$ **do**

5 $\quad\quad \texttt{Reach}_G^+(j) \leftarrow \texttt{Reach}_G^+(j) \cup (\texttt{Reach}_G^+(k) \backslash \{j\})$ ▷ Fill-in

$\texttt{Reach}^+_G(K)$ under the form of Eq. (1.18).

$$\texttt{EParent}(k) = \min(\texttt{Reach}^+_G(k)). \tag{1.18}$$

Therefore, the algorithm follows the edges of the elimination tree to propagate fill-in.

Algorithm 14 describes a scalar (it operates on the graph G but not on the quotient graph (G/P)) symbolic factorization that computes a sorted column structure $\texttt{Reach}^+_G(k)$ consisting of a list of non-zeros for that column and propagates fill-in on the following impacted columns. We begin by computing the non-zero pattern for each column of A below the diagonal, except for the last column as it has no entry below the diagonal. This corresponds to lines 1 & 2 in the algorithm and to the red blocks (non-zeros) in Fig. 1.34a and 1.34b. For example, the structure of $k = 1$ has two non-zeros. The diagonal is excluded as it is always non-zero in the considered problems. Fill-in is then computed by lines 3 to 5 in the algorithm for column m_k, where m_k is the index that corresponds to the first non-zero of column k, i.e., the minimum index in the structure $\texttt{Reach}^+_G(k)$. This computes the new column structure of m_k by merging the structures of m_k and k together. The column structure of $m_k = 4$ has one non-zero in the example figure and one (colored brown) entry will be filled by the contribution of 1. Later, other fill-in may appear from the contribution of other nodes. An example of a structure computed through a scalar symbolic factorization is depicted in Fig. 1.26 in the case of a nested dissection. Fig. 1.26d shows the initial non-zeros in red and the fill-in in brown.

Such a scalar symbolic factorization involves an arithmetic complexity as large as the fill-in produced. This complexity is equal to $\mathcal{O}(n \log n)$ for two-dimensional planar graphs with n vertices [150]. It is equal to $\mathcal{O}(n^{2\sigma})$ for $\sigma > \frac{1}{2}$, in the case of graphs verifying the n^σ-separator theorem (see Definition 1.4) [150]. For graphs associated with 3D meshes, as the size of the separator is

$$n_{sep} = n^{\frac{2}{3}}, \tag{1.19}$$

and $\sigma = \frac{2}{3} > \frac{1}{2}$, the number of non-zeros in the factors, as well as the arithmetic complexity, is equal to

$$nnz(LU) = \mathcal{O}(n^{2\sigma}) = \mathcal{O}(n^{\frac{4}{3}}). \tag{1.20}$$

As a consequence of this still consequent complexity, modern solvers [70, 119] instead compute a block (or block column) symbolic factorization.

Algorithm 14: Scalar Symbolic Factorization of a $n \times n$ matrix A based on its adjacency graph G.

Function `ScalarSymbolicFactorization`(G, n)

1 **for** $k = 1$ **to** $n - 1$ **do**
2 $\quad \texttt{Reach}^+_G(k) \leftarrow \{j \mid j > k \wedge j \in \texttt{Adj}_G(k)\}$ ▷ Initial non-zeros
3 **for** $k = 1$ **to** $n - 1$ **do**
4 $\quad m_k \leftarrow \texttt{EParent}(k)$ ▷ First non-zero below diagonal
5 $\quad \texttt{Reach}^+_G(m_k) \leftarrow \texttt{Reach}^+_G(m_k) \cup (\texttt{Reach}^+_G(k) \backslash \{m_k\})$ ▷ Fill-in

1.3.1.2.2 Block Symbolic Factorization

The block symbolic factorization computes a block sparse structure of the factorized matrix based on the initial matrix A (considered symmetric so that only the factor L is computed). In order to find this structure, we need to perform two tasks: (1) the elimination of the unknowns and (2) the partition of the unknowns.

We first recall some notations. We denote by G the adjacency graph of the matrix A, i.e., the corresponding non-zero pattern. The elimination of G is noted G^*. This elimination graph represents the non-zero pattern of the factorized matrix. G may be partitioned using a partition P of supernodes. The partition of G into a quotient graph is noted (G/P). We recall that these two tasks can be swapped only if $(G/P)^* = (G^*/P)$, which is true in the case of Theorem 1.3.

The objective of the block symbolic factorization is to compute the final block structure of L. Extra-diagonal blocks in L are related to the edges in the quotient elimination graph (G^*/P) [61]. Constructing (G^*/P) amounts to compute the quotient graph of the elimination graph G^* over P, meaning the eliminations have been calculated (through the computation of G^*) before the quotient operation. The scalar algorithm discussed in § 1.3.1.2.1 is used for the elimination of G into G^*. Then the quotient graph of G^* is computed, leading to the quotient elimination graph (G^*/P). However the complexity of this algorithm is greater than or equal to the complexity of the scalar symbolic factorization.

A faster way to compute the block symbolic factorization is to compute first the quotient operation over the supernodes (G/P) and then compute the eliminations over the quotient graph: $(G/P)^*$. This is far less costly, as we have less interactions due to the quotient property of the graph. As we have seen in § 1.3.1.1.3, for admissible cases such as the nested dissection, the graph $(G/P)^*$ is preferably computed [86, 140] due to its lower complexity.

The algorithm proceeds in the same manner as the scalar symbolic factorization, but searches through supernodes instead of nodes. The algorithm of this method is therefore identical to that of Algorithm 14, with the only difference that each element is not a node but a supernode, as illustrated by Algorithm 15. $\texttt{Reach}^+_{G/P}\texttt{(}K\texttt{)}$ lists the supernodes updated by supernode K. It is referred to as $BStruct(L_{*K})$ in [119]. Algorithm 15 details how the block symbolic factorization is computed, using this structure. Note that the parent of K in the quotient elimination tree T/P is therefore computed using the quotient structure (Eq. (1.21)).

$$\texttt{EParent}_{T/P}\texttt{(}K\texttt{)} = \min(\texttt{Reach}^+_{G/P}\texttt{(}K\texttt{)}). \tag{1.21}$$

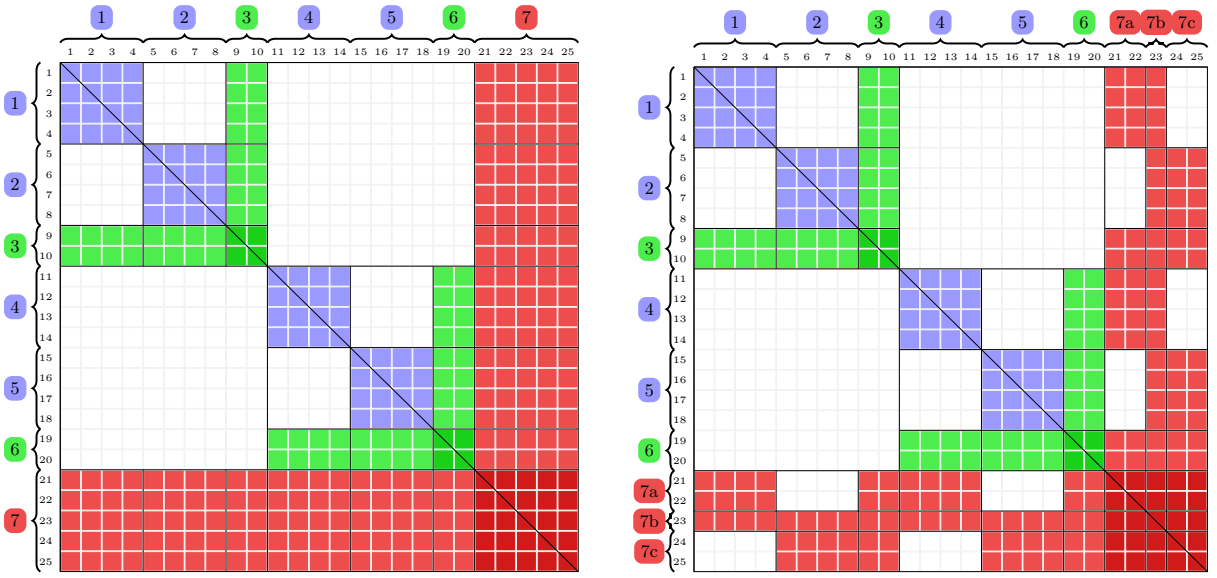
The union (line 5) now operates on lists of supernodes instead of lists of nodes. The graph G/P is used instead of the graph G so that the function $\texttt{Adj}_{G/P}\texttt{(}I\texttt{)}$ is used. However, note that with the type of supernodes chosen (see §§ 1.3.1.1.2 and 1.3.1.1.3), zeros may exist within each matrix block. Indeed, in a nested dissection-based symbolic factorization, each supernode may have the same lower column structure and yet have different interactions with supernodes ordered before. For example, if we take a look

Algorithm 15: Block Symbolic Factorization of a matrix A based on its adjacency graph G partitioned into a list P of N supernodes.

Function `BlockSymbolicFactorization`$(G,\ P,\ N)$

1 **for** $K = P[1]$ **to** $P[N-1]$ **do**
2 $\quad \texttt{Reach}^{+}_{G/P}(K) \leftarrow \{J \mid J > K \wedge J \in \texttt{Adj}_{G/P}(K)\}$
3 **for** $K = P[1]$ **to** $P[N-1]$ **do**
4 $\quad M_K \leftarrow \texttt{EParent}_{T/P}(K)$
5 $\quad \texttt{Reach}^{+}_{G/P}(M_K) \leftarrow \texttt{Reach}^{+}_{G/P}(M_K) \cup (\texttt{Reach}^{+}_{G/P}(K) \backslash \{M_K\})$

at Supernode 7 in Fig. 1.31c we can see that its interactions with previous supernodes does not match exactly the range of its rows. Some rows are filled with zeros for a specific supernode. This is due to the fact that supernodes are chosen as having the same lower column structure and not the same rows. By using these column supernodes, or T2 supernodes (Fig. 1.30a), we may not have sufficient information as depicted in Fig. 1.35a. This figure displays the structure computed by a block symbolic factorization using a partition of T2, column-based, supernodes. Using column and row information



(a) Partition of T2 supernodes.

(b) Partition of T4 supernodes.

Figure 1.35: Block Symbolic Factorization using different types of supernodes on a nested dissection-based partitioning. We can see here how relaxation introduces a few zeros in the final structure by comparing these structures to the scalar structure in Fig. 1.26d.

leads to T4 supernodes (Fig. 1.30b) and to the structure shown in Fig. 1.35b. However, choosing supernodes with this condition also leads to smaller supernodes. As discussed in [70], T2 supernodes, relying only on the lower structure of a column instead of their

entirety, are larger and are thus preferred. On the contrary, T4 supernodes, relying on rows and columns, are *too rare* and therefore rejected [70]. To fully take advantage of T2 supernodes, one must use a block column symbolic factorization instead of a block symbolic factorization.

1.3.1.2.3 Block Column Symbolic Factorization

The block column symbolic factorization is a method discussed in [61, 86, 140, 164] under the name of block symbolic factorization or blocked symbolic factorization. However we have preferred to name it here "block column symbolic factorization" to differentiate it from the "block symbolic factorization", which operates on block matrices instead of block columns in our context. Instead of relying on the interactions between supernodes, we can refine the definition of the adjacency of each supernode to capture more precisely the pattern of non-zeros associated with the original graph $G = (V, E)$:

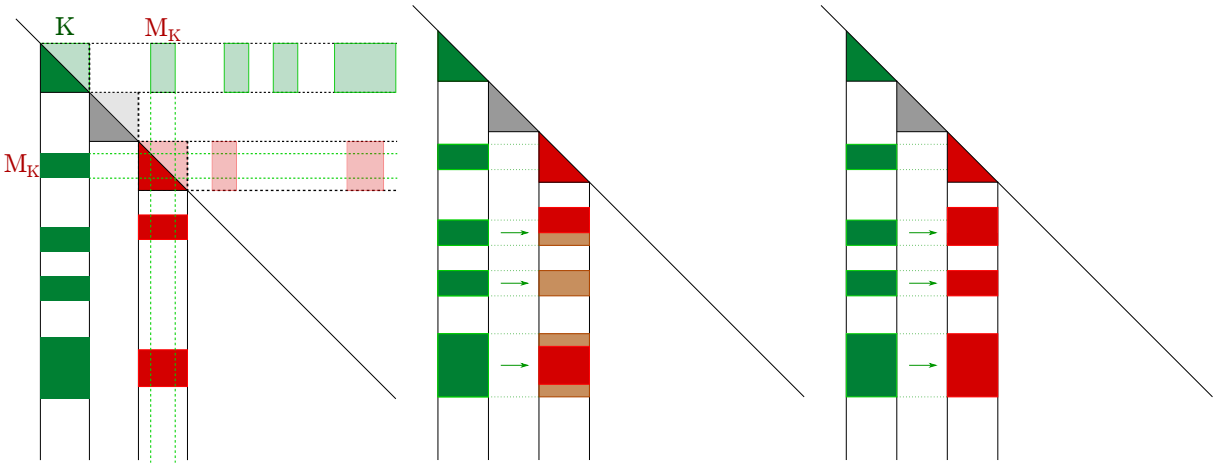
$$\texttt{Adj}_{(G/P)\to G}(I) = \{j \mid \exists i \in I \wedge (i,j) \in E\}. \tag{1.22}$$

This structure, introduced in [61], lists all the nodes which interact with at least one node of the supernode I. There is no conflict here between notations, $\texttt{Adj}_{(G/P)\to G}(I)$ is thus noted $\texttt{Adj}_G(I)$. We extend this notation to $\texttt{Reach}^+_{(G/P)\to G}(I)$, referring to the structure after factorization of each supernode. The set $\texttt{Reach}^+_{(G/P)\to G}(I)$ is consequently also noted $\texttt{Reach}^+_G(I)$ for the sake of compactness. They indeed consist in lists of nodes in G, no matter the input considered entry.

The block column symbolic factorization partitions the matrix into block-columns based on supernodes. It computes the rows that should be stored for each supernode due to fill-in. This is illustrated in Fig. 1.36. Fig. 1.36a indicates the initial non-zero rows by the green blocks for the supernode K or by the red blocks for the supernode M_K. The filled-in rows arising from the update operation, for example the brown blocks in Fig. 1.36b in the supernode M_K, are then merged with the existing red blocks into new larger blocks, as shown in Fig. 1.36c. M_K is the supernode corresponding to the first off-diagonal block of K. It does not mean it necessarily matches the same range of indices, as shown in Fig. 1.36, where the column block M_K is as large as the red blocks and is therefore larger than the first off-diagonal transposed (light green) block. Algorithm 16 details how the block symbolic factorization is computed. The set $\texttt{Reach}^+_G(K)$ lists all the nodes in G reachable from the supernode K. The structure is implemented as a list of intervals for the sake of efficiency. The parent of K in the elimination tree is consequently computed using Eq. (1.23).

$$\texttt{EParent}(K) = \{J \mid \min(\texttt{Reach}^+_G(K)) \in J\}. \tag{1.23}$$

Note that the structure $\texttt{EParent}(K)$ contains only one element (the parent supernode J). The first non-zero row in K corresponds to an index in J. Moreover, the union (line 5) operates on lists of intervals instead of lists of elements. Contrary to the block symbolic factorization previously defined (§ 1.3.1.2.2), the computed structures are not supernodes



(a) Supernodes K and M_K. (b) The elimination of K fills M_K. (c) Final storage.

Figure 1.36: Block Column Symbolic Factorization. The green blocks from supernode k impact the (red) supernode M_K due to the update operation during factorization. Therefore, for the supernode M_K, we must store all the rows corresponding to the rows of supernode k and M_K merged together.

but nodes. The initial non-zero pattern for all supernodes K is computed (lines 1,2); then it computes the new block column structure for the first supernode impacted by each K, denoted by M_K here, by merging the two structures of K and M_K (lines 3-5). This creates fill-in, as illustrated by the brown blocks in Fig. 1.36b. An example of such a block column symbolic factorization using nested dissection is depicted in Fig. 1.31.

The complexity in time and space of this algorithm [61] is related to the number of off-diagonal blocks in the final structure of the factorized matrix. This number is usually lower than the complexity of the factorization algorithm but depends on the ordering of A and the partition P. For cases satisfying Theorem 1.3, the complexity of the symbolic

Algorithm 16: Block Column Symbolic Factorization of a matrix A based on its adjacency graph G partitioned into a list P of N supernodes.

Function `BlockColumnSymbolicFactorization`(G, P, N)

1 **for** $K = P[1]$ **to** $P[N-1]$ **do**
2 $\quad \texttt{Reach}_G^+(K) \leftarrow \{j \mid j > K \wedge j \in \texttt{Adj}_G(K)\}$ ▷ List of intervals
3 **for** $K = P[1]$ **to** $P[N-1]$ **do**
4 $\quad M_K \leftarrow \texttt{EParent}(K)$
5 $\quad \texttt{Reach}_G^+(M_K) \leftarrow \texttt{Reach}_G^+(M_K) \cup (\texttt{Reach}_G^+(K) \backslash M_K)$ ▷ Exclusions of nodes in M_K

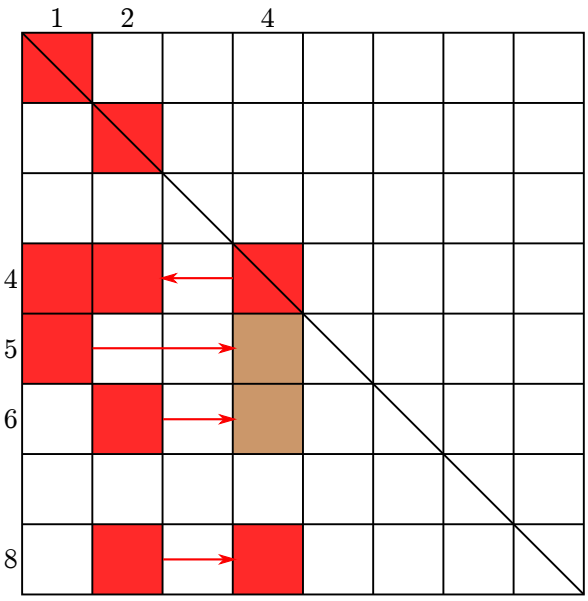


Figure 1.37: Left-looking symbolic factorization at the stage of iteration on column 4. 1 and 2 are children of 4 in the elimination tree (highlighted by the arrow pointing to the left on row 4).

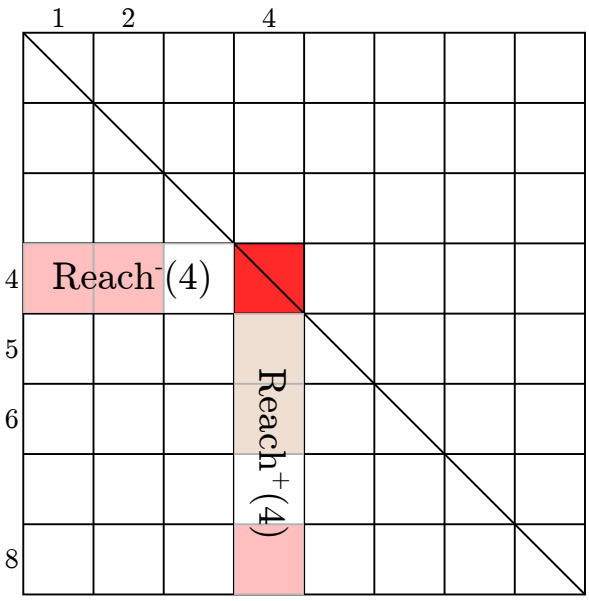


Figure 1.38: Left-looking and right-looking structures of 4. The left-looking structure contains 1 and 2. The right-looking structure contains 5,6 and 8.

factorization is linear with the number of off-diagonal blocks [61]. Furthermore, the number of off-diagonal blocks increases linearly with n [61].

1.3.1.2.4 Discussion on a Left-Looking Symbolic Factorization

Instead of applying the contribution to m_k, we could store the information that m_k receives contributions from k. Later on, the union of all the nodes contributing to m_k (of which m_k is the first non-zero row, or the parent in the elimination tree), can be computed. All these nodes are children of m_k in the elimination tree and are therefore stored in a structure named `EChildren(`m_k`)`. `EChildren(`k`)` is defined as:

$$\texttt{EChildren}(k) = \{j \mid \texttt{EParent}(j) = k\}. \tag{1.24}$$

This can be seen as a left-looking algorithm in the sense that contributions are applied from the *left*, i.e., the union is performed on the receiving node. This can be compared to the right and left-looking methods discussed in § 1.3.1.3.1. For the example from Fig. 1.32c, instead of applying the contribution $1 \rightarrow 4$ and $2 \rightarrow 4$ independently, as illustrated in Fig. 1.34a and 1.34b, we can apply both contributions together, as depicted in Fig. 1.37.

Algorithm 17, based on [93, 97, 110], computes this left-looking symbolic factorization. This algorithm can be generalized for the block and block column versions. However, we

Algorithm 17: Left-Looking Symbolic Factorization of a matrix A associated with a graph G with n nodes.

Function `LeftSymbolicFactorization`(G, n)

1 **for** $k = 1$ **to** $n - 1$ **do**
2 $\quad \texttt{Reach}_G^+(k) \leftarrow \{j \mid j > k \wedge j \in \texttt{Adj}_G(k)\}$
3 **for** $k = 1$ **to** $n - 1$ **do**
4 $\quad \texttt{Reach}_G^+(k) \leftarrow \texttt{Reach}_G^+(k) \cup \bigcup_{j \in \texttt{EChildren}(k)} \texttt{Reach}_G^+(j) \backslash \{k\}$
5 $\quad m_k \leftarrow \texttt{EParent}(k)$
6 $\quad \texttt{EChildren}(m_k) \leftarrow \texttt{EChildren}(m_k) \cup k$

will need a construction using the information of *all* the contributing columns in § 2.4. This information, which is the reverse from $\texttt{Reach}_G^+(k)$ can be noted $\texttt{Reach}_G^-(k)$. This structure corresponds to the interactions of row k before the diagonal. We can now distinguish the right-looking interactions $\texttt{Reach}_G^+(k)$ from the left-looking interactions $\texttt{Reach}_G^-(k)$, as depicted in Fig. 1.38. Also, $\texttt{EChildren}(k)$ must not be confused with $\texttt{Reach}_G^-(k)$. Indeed, $\texttt{Reach}_G^-(4) = \texttt{EChildren}(4) = [1, 2]$, but $\texttt{Reach}_G^-(6) = [2, 4, 5]$ in Fig. 1.32c, and $\texttt{EChildren}(6) = [5]$. A naive algorithm would consist of computing each $\texttt{Reach}_G^-(k)$ using the previous structure $\texttt{Reach}_G^-(i), i < k$, as detailed in Algorithm 18. However, this algorithm has a complexity higher than the usual symbolic factorization, as another 'for' loop is inserted at line 4. Therefore we introduce Algorithm 19

Algorithm 18: Left-Looking Symbolic Factorization of a matrix A associated with a graph G with n nodes and computing a left-looking interactions structure.

Function `LeftSymbolicFactorization`(G, n)

1 **for** $k = 1$ **to** $n - 1$ **do**
2 $\quad \texttt{Reach}_G^-(k) \leftarrow \{j \mid j < k \wedge j \in \texttt{Adj}_G(k)\}$ ▷ Left-looking
3 **for** $k = 1$ **to** $n - 1$ **do**
4 $\quad$ **for** $j = 1$ **to** $k - 1$ **do**
5 $\quad\quad$ **if** $\texttt{Reach}_G^-(j) \cap \texttt{Reach}_G^-(k) \neq \emptyset$ **then**
6 $\quad\quad\quad \texttt{Reach}_G^-(k) \leftarrow \texttt{Reach}_G^-(k) \cup j$ ▷ fill-in between j and k

which computes such a left-looking structure with an arithmetic complexity close to Algorithm 17, traded for a slightly higher memory usage. We must emphasize that the algorithm is categorized as left-looking due to the fact that the updates are applied on the receiving node, even though it relies on right and left-looking structures.

Algorithm 19: Efficient Left-Looking Symbolic Factorization of a matrix A associated with a graph G with n nodes and computing both the right and left-looking interactions structures.

Function `LeftSymbolicFactorization`$(G,\ n)$

1 **for** $k = 1$ **to** $n-1$ **do**
2 $\quad \texttt{Reach}^{+}_{G}(k) \leftarrow \{j \mid j > k \wedge j \in \texttt{Adj}_{G}(k)\}$ ▷ Right-looking
3 $\quad \texttt{Reach}^{-}_{G}(k) \leftarrow \{j \mid j < k \wedge j \in \texttt{Adj}_{G}(k)\}$ ▷ Left-looking
4 **for** $k = 1$ **to** $n-1$ **do**
5 $\quad \texttt{Reach}^{+}_{G}(k) \leftarrow \texttt{Reach}^{+}_{G}(k) \cup \bigcup_{j \in \texttt{EChildren}(k)} \texttt{Reach}^{+}_{G}(j) \backslash \{k\}$
6 $\quad m_k \leftarrow \texttt{EParent}(k)$
7 $\quad \texttt{EChildren}(m_k) \leftarrow \texttt{EChildren}(m_k) \cup k$
8 $\quad$ **for** $j = \texttt{Reach}^{+}_{G}(k)$ **do**
9 $\quad\quad \texttt{Reach}^{-}_{G}(j) \leftarrow \texttt{Reach}^{-}_{G}(j) \cup k$

1.3.1.2.5 Symbolic Factorization with Different Partitions

In the case of matrices with different row and column partitions, we may re-write all right-looking algorithms 14 to 16 into the same general form, as shown in Algorithm 20. To avoid confusion, we focus here on right-looking algorithms but this discussion may be translated for left-looking algorithms.

Algorithm 20: Generalized Symbolic Factorization of a matrix A associated with a graph G with n nodes following a column partition P_{col} over a row partition P_{row}.

Function `SymbolicFactorization`(G, P_{row}, P_{col}, N)

1 **for** $K = P_{col}[1]$ **to** $P_{col}[N-1]$ **do**
2 $\quad \texttt{Reach}(K) \leftarrow \{J \mid J > K \wedge J \in \texttt{Adj}_{(G/P_{col})\rightarrow(G/P_{row})}(K)\}$
3 **for** $K = P_{col}[1]$ **to** $P_{col}[N-1]$ **do**
4 $\quad M_K \leftarrow \texttt{EParent}(K)$
5 $\quad \texttt{Reach}(M_K) \leftarrow \texttt{Reach}(M_K) \cup (\texttt{Reach}(K) \backslash \{M_K\})$

The row partition P_{row} may be different from the column partition P_{col} by its size as well as by the unknowns it covers. The two partitions may therefore be (1) equal, (2) overlapping, (3) disjoint. The first case corresponds to most cases, including the sparse linear system arising from the FEM considered in this thesis. The second case may correspond to a symbolic factorization applied on the sparse submatrices of the overall FEM/BEM coupling, just as the subsystem in Eq. (1.25).

$$\begin{bmatrix} A_{vv} \\ A_{sv} \end{bmatrix}. \tag{1.25}$$

Here, P_{col} partitions the unknowns from the FEM exclusively, while P_{row} partitions the whole set of unknowns of the overall problem. The notion of symbolic factorization is therefore extended to non-square matrices $m \times n$, with $m \geq n$, and may be adapted to QR factorization. An example of the third case could simply be the computation of the symbolic factorization of the matrix A_{sv} alone. In this last case, without any extra information from A_{vv}, as the information of the whole column is not available, the symbolic factorization may not be entirely computed, as fill-in may not be propagated properly.

This is further discussed in the context of the FEM/BEM coupling in § 2.5.

Eventually, the symbolic factorization computes the final pattern of non-zeros in the factors. The symbolic factorization must trade-off between efficiency of the computations and memory usage. This step is necessary to avoid efficiency issues due to on-the-fly fill-in management during numerical factorization, introduced in the following section.

1.3.1.3 Numerical Factorization

We now discuss two different classes of methods used to factorize the matrix A: multifrontal methods and supernodal methods. In the LU factorization, the contributions (or updates), i.e., the operations $A_{ij} \leftarrow A_{ij} - A_{ik}A_{kj}$, can be computed in different ways. The multifrontal and supernodal methods differ in the way these contributions are applied.

1.3.1.3.1 Supernodal Methods

Supernodal methods apply contributions directly on the targeted supernode as soon as they appear. Within the class of supernodal methods, there are two classical variants: the *left-looking* and the *right-looking* algorithms. The difference lies in the order the operations are carried out: in the right-looking approach, a supernode is factorized and then applies all the contributions *on the right*, whereas in the left-looking approach, all contributions to a panel are accumulated *from the left* until the algorithm arrives at the factorization of the destination panel, meaning each supernode is read once when factorized and later once again to apply the contributions. The right-looking algorithm

Algorithm 21: Simplified left-looking algorithm factorization.

```
1 for j = 1 to n do
2 |   forall i contributing to j do
3 |   |   Update(A_{*i}, A_{*j})
4 |   Factorize(A_{*j})
```

needs less read operations but writes several times on the destination supernode while the left-looking algorithm reads multiple times the (contributions) data but writes only once on the destination supernode. The left-looking and right-looking algorithms are

Algorithm 22: Simplified right-looking algorithm factorization.

```
1 for i = 1 to n do
2     Factorize(A_{*i})
3     forall j depending on i do
4         Update(A_{*i}, A_{*j})
```

respectively described in Algorithm 21 and Algorithm 22. It should be noted that the right-looking algorithm has already been introduced in § 1.2.1.1 for dense matrices. The cost of the update matrix-vector operations may be quite high. To find a compromise between performance and memory consumption, a variant has been developed, which groups updates by supernodes and scatters the results of the updates onto the destination panels. This method is referred to as gather/scatter [145].

The right-looking supernodal solver PaStiX [119] can be used in Actipole to handle sparse operations (*LU* factorization for example). Other supernodal solvers include SuperLU [70], PARDISO [176] or symPACK [127]. We refer the interested reader to [69, 73, 76] for more informations.

1.3.1.3.2 Multifrontal Methods

Multifrontal methods [77, 78, 146, 153, 177] apply contributions through dense submatrices, called *frontal matrices*, that hold all the contributions from one subtree of the elimination (or assembly) tree to its ascendants. The first frontal matrix (associated with supernode 1) of the example matrix of Fig. 1.31c is shown in Fig. 1.39. All elimination operations are performed through these frontal matrices. A frontal matrix can be formulated as four matrix blocks, arranged as in Eq. (1.26):

$$F = \begin{bmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{bmatrix}. \tag{1.26}$$

The first block column and row are called *fully-summed*, because all possible updates have been computed for the corresponding supernodes. The second block column and row are partly summed. The Schur complement matrix, equal to $F_{22} - F_{21}F_{11}^{-1}F_{12}$, is also referred to as the *contribution block* as it will be used to update rows and columns from ancestor frontal matrices (upper in the assembly tree). The operations and frontal matrices involved in the example matrix of Fig. 1.31c are shown in Fig. 1.40. This assembly tree, based on the elimination tree in Fig. 1.31b, shows the dependencies between computations involving frontal matrices. The memory consumption of a multifrontal method varies over time as each frontal matrix is assembled and then processed and contribution blocks are stacked, waiting to be consumed by another front. This *active memory* increases and decreases each time a front or contribution block is assembled and freed, respectively, while the memory consumed by the factors L and U, the fully-summed variables, constantly increases during the factorization. Multifrontal methods therefore operate with a larger

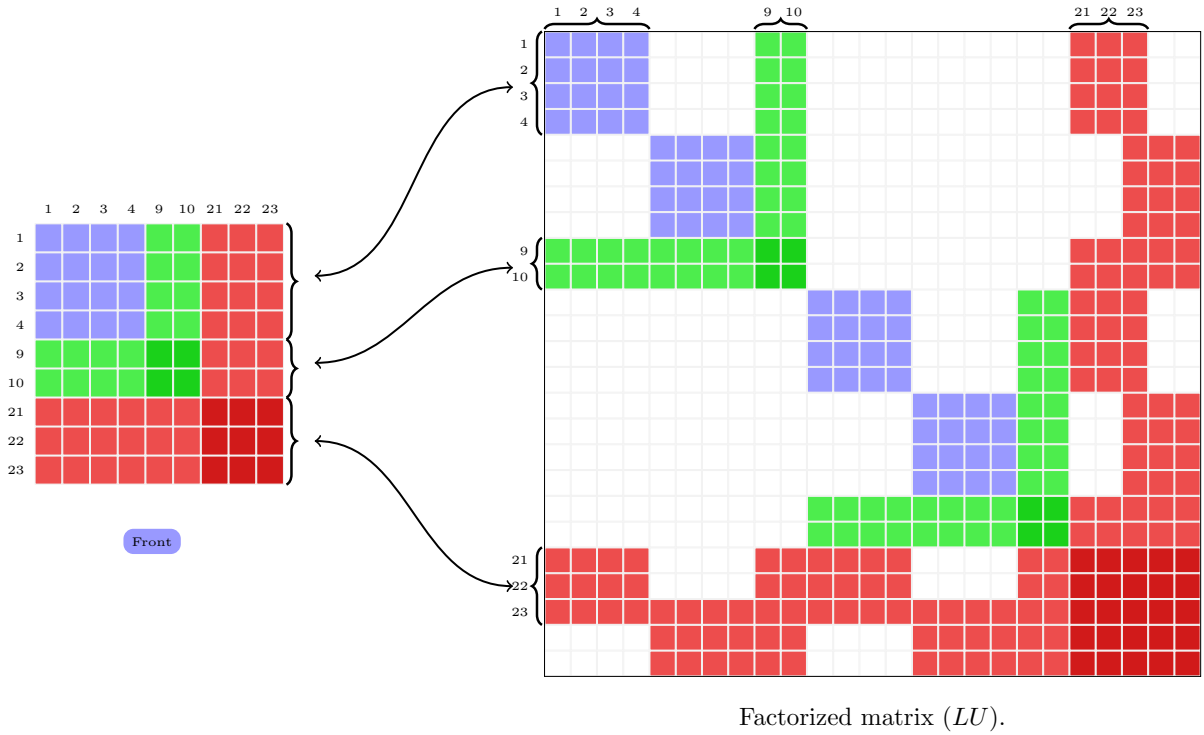


Figure 1.39: Construction of a frontal matrix for the first supernode.

granularity than supernodal methods (which may be used to design efficient blocked algorithms) but consume extra memory for storing the contribution blocks.

The multifrontal solver MUMPS [23, 24] is available in Actipole for performing various sparse operations, including LU factorization or computing a Schur complement. Other examples of multifrontal implementations may include UMFPACK [67] or the Harwell Subroutine Library (HSL) code MA41[5] [21] . TAUCS [186] gathers multiple implementations, including a multifrontal factorization and a left-looking factorization.

1.3.2 Sparse Iterative Methods

The main principles of iterative methods are the same, regardless of the characteristics of the matrices (sparse or dense). We present here some of the elements that show in what respect they differ. We have seen in § 1.1.3.2 that iterative methods compute a matrix-vector product at each iteration of the process. This matrix-vector is usually computed in $\mathcal{O}(n^2)$ for dense problems (or $\mathcal{O}(n \log n)$ when using the FMM). Yet, when applied on a sparse matrix, the matrix-vector product can be computed in only $\mathcal{O}(nnz)$. Moreover, there are specific preconditioners that exploit the sparsity of the structure, as for example the ILU method [156], which computes an Incomplete LU factorization, ignoring some of the fill-in that occurs. The SPAI [13, 55, 54] is also a sparse (approximate

[5] http://www.hsl.rl.ac.uk/catalogue/ma41.html

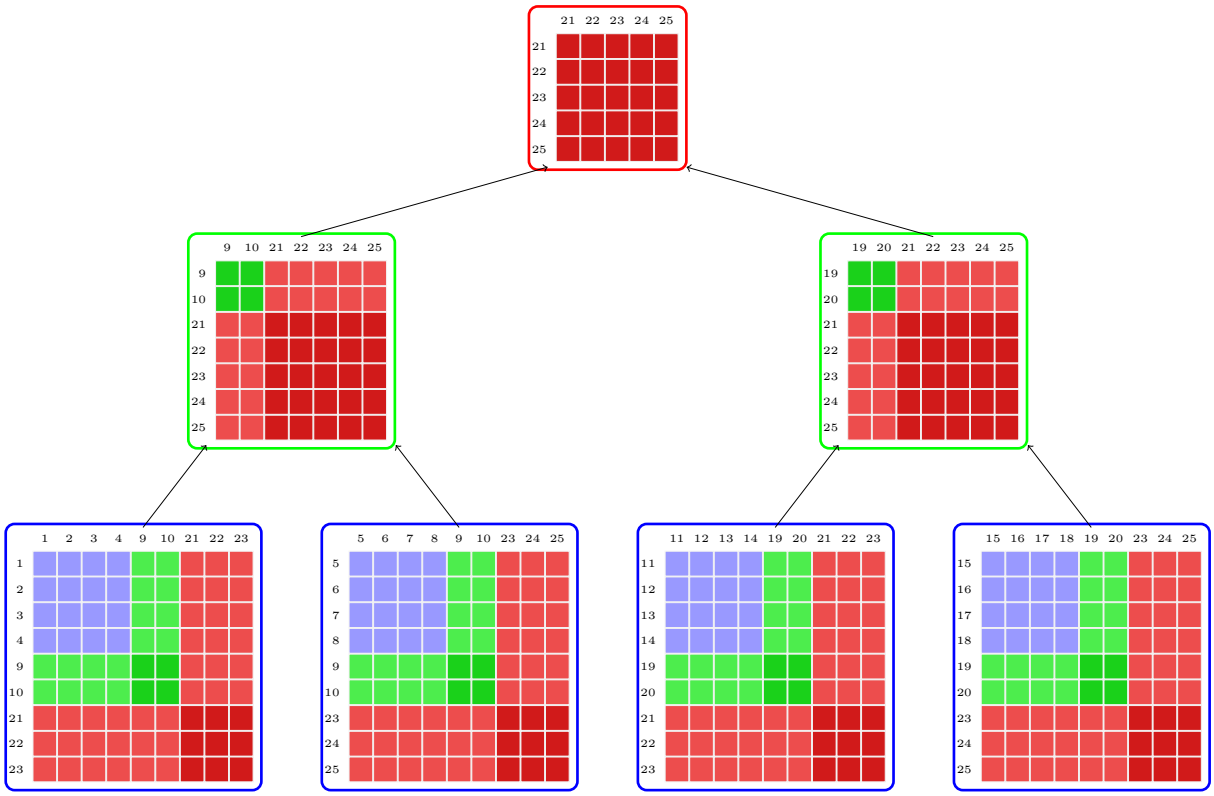


Figure 1.40: Assembly tree with the frontal matrices involved.

inverse) preconditioner, used for example in the Airbus framework [184]. We refer to [172] for an overview of modern iterative methods.

1.4 Solution of a FEM/BEM system

We have presented the linear system we are interested in solving in Eq. (1.2). We have then discussed the difference between dense solutions in § 1.2 and sparse solutions in § 1.3. We now detail how the solution of the coupled system is computed in the industrial framework of this thesis. Let us remind the reader we are interested in solving a linear system $Ax = b$, where $A \in \mathbb{C}^{n \times n}$ is decomposed into a 2×2 block matrix with the following form:

$$A = \begin{bmatrix} A_{vv} & A_{vs} \\ A_{sv} & A_{ss} \end{bmatrix}. \tag{1.27}$$

where A_{vv}, A_{vs}, A_{sv} are sparse matrices and A_{ss} is dense. We first discuss architecture and efficiency considerations for the coupling in § 1.4.1, then give references to related works on FEM/BEM couplings in the literature in § 1.4.2 and then introduce the methods considered for the solution of the coupling in § 1.4.3.

1.4.1 Efficiency and Architecture Considered

In the context of this thesis, we heavily rely on parallel linear solvers, necessary for the study of wave propagation. Indeed, when a mesh is computed, the size of each element needs to be consistent with the physical problem considered. For an acoustic problem, the range of frequencies studied correspond to the human hearing, i.e., 20 to 20,000 Hz, which corresponds to wavelengths of 17 meters to 1.7 cm. In order for the calculations to be sufficiently precise, the mesh element size needs to match those wavelengths. Typically, the mesh element size would be around $\lambda/10$ or $\lambda/15$. But we can calculate that a mesh of $1m^3$ discretized using elements of $1mm^3$ would already mean 1 billion elements for a mesh computed using the FEM. Consequently, the increasing number of elements required for such simulations if the mesh covers objects as large as aircraft engines leads to very large systems of equations, therefore implying the need for efficient computations in both time and memory. Even though the number of elements to store and the number of calculations are reduced due to the sparsity of the problem arising from FEM, they are still quite large. Thus, the use of HPC platforms is often mandatory to reduce the time to solution of such computations. The same is true for problems arising from the BEM: the two-dimensional characteristics of the meshes computed using this method lead to problems of smaller size. However, they also lead to dense linear systems, meaning more intensive calculations and more memory requirements than for a mesh computed with the FEM that would have the same number of elements, thus necessitating the use of HPC.

HPC is a large domain whose main goal is to process calculations faster and/or being able to use more memory. It can be seen as a means to reach the ability to solve larger problems or to solve faster problems. There can be two ways to achieve this goal: either optimize the algorithm or improve the performance of the hardware. HPC relies heavily on parallel computing. In this work, we do not investigate this field but rather try to design novel algorithms, first studying their sequential behavior and then their parallel efficiency. We consider in this thesis mainly multicore parallelism with shared memory, though distributed parallelism remains a long-term objective. A parallel algorithm should be aware of the memory and cache characteristics to be efficient [147] . To avoid a difficult management of parallelism, a runtime system such as StarPU [30, 31] may be used. In the context of the $\mathcal{H}$-Matrix solver, a runtime system has also been implemented as an alternative, named toyRT, of which a version may be found at [7].

1.4.2 Related Work

The coupling between BEM and FEM have been discussed in numerous studies since at least the 1970s [158, 200, 130]. Also, more recently, [196] introduced a coupling between a stochastic method using decomposition to exploit the block structure of a matrix for FEM-FEM simulations and an $\mathcal{H}$-Matrix to compress the integral domain from the BEM. In [128], the authors describe the solution of a FEM-BEM system using a Domain Decomposition Method (DDM) relying on an $\mathcal{H}$-Matrix multiplicative Schwarz

preconditioner. The authors of [120] introduce a solution of a FEM/BEM system using $\mathcal{H}^2$-Matrices.

1.4.3 Airbus Solver

Two methods can be used in the solver considered [12] to solve the global linear system. Either the volume unknowns are reduced on the boundary using a Schur complement, or the system is solved as is. Depending on this choice, we must also choose if direct methods or iterative methods will be used. We first detail how the system is solved when using a Schur complement, and later explain how it is solved without.

With a Schur complement

The Schur complement method [198] consists in reducing the whole problem on the boundary. To solve the system, we can first state x_v based on the first equation in the 2×2 system (1.2), which can be formulated as in Eq. (1.28).

$$x_v = A_{vv}^{-1}(b_v - A_{vs}x_s). \tag{1.28}$$

When inserting this term in the second equation of the system, we obtain Eq. (1.29), in which we can identify a new matrix, called the Schur complement and denoted $\mathcal{S}$ here, which is more precisely expressed by the formula of Eq. (1.30).

$$(A_{ss} - A_{sv}A_{vv}^{-1}A_{vs})x_s = b_s - A_{sv}A_{vv}^{-1}b_v, \tag{1.29}$$

$$\mathcal{S} = A_{ss} - A_{sv}A_{vv}^{-1}A_{vs}. \tag{1.30}$$

The system (1.29) has only one vector unknown left to be calculated (x_s). When using a direct solver, the Schur complement $\mathcal{S}$ is explicitly computed and factorized in order to solve the system, whereas for iterative solvers the matrix is not necessarily computed and successive implicit matrix-vector products are used on each term of the formula, i.e., $A_{ss} - A_{sv}A_{vv}^{-1}A_{vs}$ [12]. In the case of an iterative solution, the system (1.30) can be preconditioned using a SPAI, which approximates the inverse of A_{ss}.

Without a Schur complement

In the case of a solution without a Schur complement, the only option available in the solver is to use an iterative solver. In this case, a block diagonal preconditioner is used to improve convergence, as formulated in Eq. (1.31).

$$\begin{bmatrix} A_{vv} & A_{vs} \\ A_{sv} & A_{ss} \end{bmatrix} \times \begin{bmatrix} P_{vv} & 0 \\ 0 & P_{ss} \end{bmatrix} \times \begin{bmatrix} x_v \\ x_s \end{bmatrix} = \begin{bmatrix} b_v \\ b_s \end{bmatrix}. \tag{1.31}$$

We may observe that the matrix-vector products involve here a vector of size $n_{FEM}+n_{BEM}$ which may prove to be quite large. In this solver, P_{vv} is either a SPAI, or A_{vv}^{-1} (which the ideal local preconditioner), or the identity and P_{ss} can be either the identity or a SPAI.

Available options

To solve the global system (1.2), several options are available in the context of the considered Airbus' solver. For a direct solution of the system, we may rely on a sparse direct solver (MUMPS or PaStiX) in combination with a dense direct solver (SPIDO, an in-house dense direct solver) or the $\mathcal{H}$-Matrices for all operations. For an iterative solution of the system, Block GCR or GMRES [173] may be used coupled with the FMM to speed up calculations and a sparse solver (MUMPS, PaStiX) to handle sparse operations that may arise, as in the computation of the Schur complement (Eq. (1.28) and (1.30)).

1.4.4 Considered Methods

We propose here to compare two approaches for the direct solution of Eq. (1.2):

1. a method based on MUMPS as a sparse direct solver for the elimination of the FEM unknowns and the in-house dense direct solver SPIDO for the factorization of the BEM-BEM subsystem, discussed in § 1.4.4.1;
2. a method based on $\mathcal{H}$-Matrices, described in § 1.4.4.2.

1.4.4.1 Solution Combining MUMPS and SPIDO

This approach relies on the computation of a Schur complement via MUMPS and the dense factorization being handled by SPIDO (see § 1.4.4.1.1). Due to the nature of the dense matrix A_{ss}, two algorithms may compute the Schur complement of the FEM unknowns over the BEM unknowns. The original Multi-Solve variant is explained in § 1.4.4.1.2 while its modified algorithm, the Multi-Factorization, is detailed in § 1.4.4.1.3.

1.4.4.1.1 SPIDO

SPIDO is a direct solver from Airbus that may factorize a matrix either using the LU factorization or the LDL^T decomposition. This parallel and out-of-core solver was implemented in the 1990s, since then updated in order to remain robust and reliable [154]. When physical memory is not sufficient for the required computations, it is possible to use an out-of-core approach to avoid running *out-of-memory*. Out-of-core blocks will be stored on disk when not needed, thus saving physical memory. Disks can logically be seen as a way to store data as much as physical memory and memory can therefore be compared to a cache for the disks. However, using the disks as a means to store data usually leads to a strong decline in performance, as disks have a greater latency and lower bandwidth when accessing and moving data. In order to process very large and dense systems, SPIDO must rely on the out-of-core technique and parallelism to efficiently reduce the computation timespan and the memory allocated at a specific time. Therefore, SPIDO works with two levels of matrix division:

- Out-of-core blocks;
- Processor blocks.

Definition 1.8 lists the different dimensions used in this section, following the notations used in Fig. 1.5.

Definition 1.8. For a matrix $A_{ss} = \mathbb{C}^{n_{BEM} \times n_{BEM}}$, n_{blocks} is defined as the number of out-of-core blocks in one column of the division, i.e., the number of rows. It is equal to the number of out-of-core blocks in one row of the division, i.e., the number of columns. n_b is the row or column size of each out-of-core block. We have $n_b \times n_{blocks} = n_{BEM}$. Also, $n_{ooc} = n_{blocks} \times n_{blocks}$ is the total number of out-of-core blocks in the matrix.

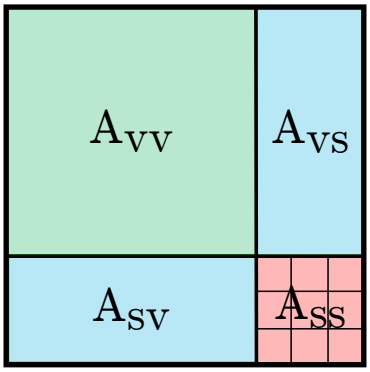


Figure 1.41: Subdivision of A_{ss} into nine out-of-core blocks.

Fig. 1.41 illustrates the division of A_{ss} into out-of-core blocks. When an out-of-core block is in use, each of its underlying processor blocks are distributed on the nodes selected for the experiment so that we can use distributed parallelism to speed up calculations. Naturally, multi-threading computations can also be used here in order to further improve performance within each node.

We now discuss two techniques used for the solution of the system (1.2) by the computation of a Schur complement, i.e., Eq. (1.30). To that end, these two methods rely on different techniques: the first one, the Multi-Solve method, computes many solves involving A_{vv} while the second, the Multi-Factorization method, will directly compute parts of the Schur complement on the destination submatrix (an out-of-core block) of the matrix A_{ss}.

1.4.4.1.2 Multi-Solve

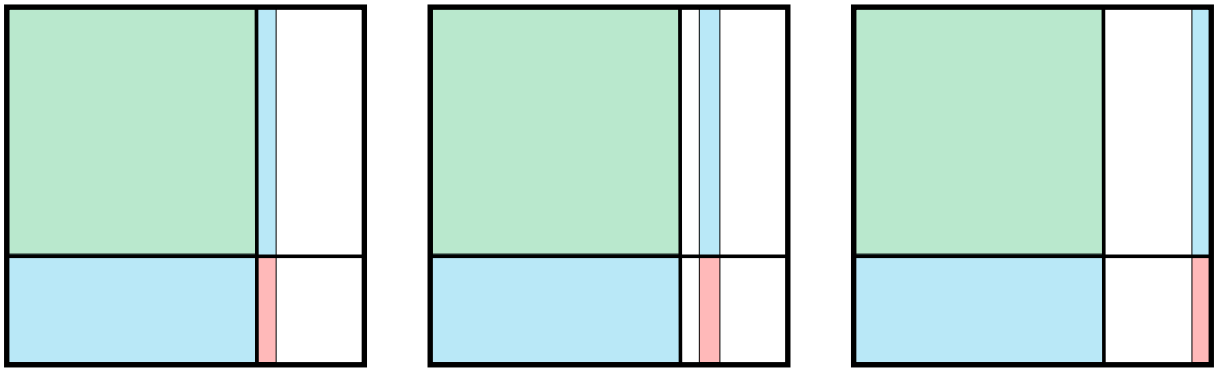
The solution of the coupling system (1.2) through the Multi-Solve technique relies on the computation of the Schur complement by block columns. This is detailed in Algorithm 23. The algorithm loops on block columns of size 32 and therefore uses a subset of A_{vs} such as shown in Fig. 1.42. At line

4 in the algorithm, the matrix $A_{vs}[i]$ is a subset of A_{vs} with 32 columns.

Algorithm 23: Solution of $Ax = b$ using Multi-Solve.

Function `MultiSolve`(A, b)

1 $A = \begin{bmatrix} A_{vv} & A_{vs} \\ A_{sv} & A_{ss} \end{bmatrix}$
2 `Factorization`(A_{vv})
3 **for** $i = 1$ **to** $n_{BEM}/32$ **do**
4 $Y \leftarrow A_{vv}^{-1} A_{vs}[i]$ ▷ Solve using i-th column of A_{vs}
5 $Z \leftarrow A_{sv} Y$ ▷ GEMM
6 $A_{ss} \leftarrow A_{ss} - Z$ ▷ AXPY
7 `Solve`(A_{vv}, b_v) ▷ In-place solution
8 `Factorization`(A_{ss}) ▷ SPIDO
9 `Solve`$(A_{ss}, b_s - A_{sv} b_v)$



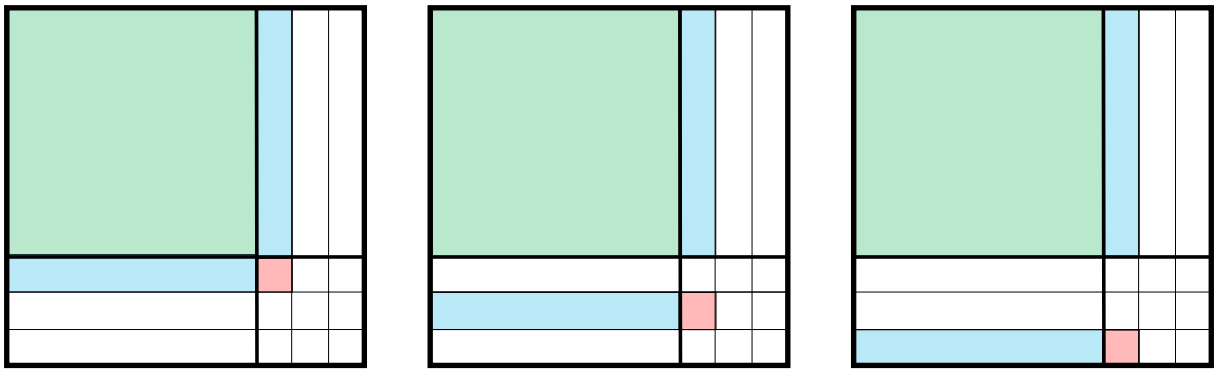
(a) First iteration. (b) Second iteration. (c) Last iteration.

Figure 1.42: Iterations of the Multi-Solve algorithm.

Operation	Iterations	Flop	Storage
Total	n_{BEM}	$nnz(A_{vv}) + nnz(A_{vs}) + n_{BEM}$	$\mathcal{O}(n_{FEM})^{\frac{4}{3}} + \lvert A_{sv}\rvert + \lvert A_{vs}\rvert + n_{BEM}^2$
Solve	n_{BEM}	$nnz(A_{vv})$	
GEMM	n_{BEM}	$nnz(A_{vs})$	
AXPY	n_{BEM}	n_{BEM}	

Table 1.3: Multi-Solve Schur complement detailed arithmetic and memory complexities.

Using Definition 1.1, the theoretical complexity of the computation of the Schur complement (the main loop) is therefore given in Table 1.3. The complexity is given for each column instead of 32 block columns for the sake of simplicity.



(a) First iteration. (b) Second iteration. (c) Third iteration.

Figure 1.43: Three iterations of the Multi-Factorization loop on the first column.

1.4.4.1.3 Multi-Factorization

To find the solution of the problem (1.2), the Multi-Factorization method iterates over the out-of-core blocks of A_{ss} (an example of three steps for the first column are shown in Fig. 1.43). It constructs a new matrix Y composed of the submatrix A_{vv} and the submatrices of A_{sv}, A_{vs} and an empty block corresponding to the current block of A_{ss} (Fig. 1.44). The Schur complement of Y is computed and added to A_{ss}. This Schur complement includes the factorization of the matrix Y, hence the name of the method. This is formally expressed in Algorithm 24.

Algorithm 24: Solution of $Ax = b$ using Multi-Factorization.

Function `MultiFactorization`(A, b)

1 $A = \begin{bmatrix} A_{vv} & A_{vs} \\ A_{sv} & A_{ss} \end{bmatrix}$
2 **for** $i = 1$ **to** n_{blocks} **do**
3 **for** $j = 1$ **to** n_{blocks} **do**
4 $Y \leftarrow \begin{bmatrix} A_{vv} & A_{vs}[j] \\ A_{sv}[i] & 0 \end{bmatrix}$
5 $A_{ss}[i][j] \leftarrow A_{ss}[i][j] +$ `SchurComplement`(Y)
6 `Factorization`(A_{vv})
7 `Solve`(A_{vv}, b_v) ▷ In-place solution
8 `Factorization`(A_{ss}) ▷ SPIDO
9 `Solve`$(A_{ss}, b_s - A_{sv}b_v)$

This algorithm allows the solver to handle less calls to the external MUMPS library, but performs more factorizations instead of solves. In theory, this algorithm should therefore lead to a higher complexity (see Table 1.4) that the Multi-Solve technique. Indeed, if we subtract the total arithmetic complexity of Table 1.3 to that of Table 1.4, multiplied by the number of iterations of the loop, and replace n_{ooc} using Definition 1.8,

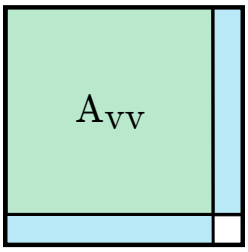


Figure 1.44: Structure of the assembled matrix Y used to compute each Schur complement in the Multi-Factorization routine.

Iterations	Flop	Storage
n_{ooc}	$\mathcal{O}(n_{FEM}+n_b)^2$	$\mathcal{O}(n_{FEM}+n_b)^{\frac{4}{3}}$

Table 1.4: Multi-Factorization Schur complement detailed arithmetic and memory complexities.

we obtain:

$$n_{ooc} \times (n_{FEM}+n_b)^2 - n_{BEM} \times (nnz(A_{vv}) + nnz(A_{vs}) + n_{BEM}) = \frac{n_{BEM}^2}{n_b^2} \times (n_{FEM}+n_b)^2 - n_{BEM} \times (nnz(A_{vv}) + nnz(A_{vs}) + n_{BEM}). \quad (1.32)$$

By dividing the equation by n_{BEM}, and simplifying the terms, we obtain:

$$\frac{n_{BEM}}{n_b^2} \times (n_{FEM}+n_b)^2 - nnz(A_{vv}) - nnz(A_{vs}) - n_{BEM} = \frac{n_{BEM} \times n_{FEM}^2}{n_b^2} + 2\frac{n_{BEM} \times n_{FEM} \times n_b}{n_b^2} + \frac{n_{BEM} \times n_b^2}{n_b^2} - nnz(A_{vv}) - nnz(A_{vs}) - n_{BEM} = \frac{n_{BEM} \times n_{FEM}^2}{n_b^2} + 2\frac{n_{BEM} \times n_{FEM}}{n_b} - nnz(A_{vv}) - nnz(A_{vs}). \quad (1.33)$$

We know that $n_b < n_{BEM}$ and Eq. (1.33) decreases when n_b increases. To study the best theoretical performance of the Multi-Factorization, we may thus replace $n_b = n_{BEM}$ in Eq. (1.33). We obtain:

$$\frac{n_{FEM}^2}{n_{BEM}} + 2 \times n_{FEM} - nnz(A_{vv}) - nnz(A_{vs}). \quad (1.34)$$

If we consider that n_{FEM} increases with N^3 and n_{BEM} with N^2 due the volume and surface characteristics of the discretizations, and that $nnz(A_{vv}) = \mathcal{O}(n_{FEM}^{\frac{4}{3}})$ following Eq. (1.20), we may perform a change of variable in Eq. (1.34) and write:

$$\frac{N^6}{N^2} + 2 \times N^3 - N^4 - nnz(A_{vs}) = 2 \times N^3 - nnz(A_{vs}). \quad (1.35)$$

Also, by definition of sparse matrices, $nnz(A_{vs}) \ll n_{FEM} \times n_{BEM} = \mathcal{O}(N^5)$. Therefore, Eq. (1.35) may be positive or negative following the number of non-zeros of A_{vs}, implying that the difference between the theoretical complexities of the Multi-Factorization and the Multi-Solve methods depends on the structure of this matrix. However, for smaller values of n_b, the complexity of the Multi-Factorization should be larger than the complexity of the Multi-Solve technique. The Multi-Factorization may nonetheless have an advantage over the Multi-Solve technique for considerations of BLAS-3 efficiency and management of external libraries.

1.4.4.2 Solution Using $\mathcal{H}$-Matrices

To find a solution of the global linear system (1.2) using $\mathcal{H}$-Matrix techniques, we have two options. An $\mathcal{H}$-Matrix may be constructed over the combined surface and volume mesh such as displayed in Fig. 1.45, however we do not benefit from the sparsity of the volume mesh in this case. Alternatively, the $\mathcal{H}$-Matrix may be divided into four submatrices (three in the symmetric case) corresponding to the 2×2 matrix given in Eq. (1.27). An example of the matrices involved in the solution of a symmetric linear system using $\mathcal{H}$-Matrices based on recursive bisection is shown in Fig. 1.46. A large part of the off-diagonal submatrices in Fig. 1.46a are in fact low-rank matrices with a null rank (light green matrices without numbers) and represent as such effective zero blocks.

The main operation benefiting from the use of $\mathcal{H}$-Matrices is the dense factorization of the matrix block A_{ss} arising from the interaction of the unknowns located in the mesh discretized using BEM with themselves. However, the factorization of the sparse system A_{vv} does not benefit from the sparse techniques introduced in § 1.3. Chapter 2 therefore discusses the introduction of such techniques for the solution of a sparse system (involving for example A_{vv}) through a sparse hierarchical factorization, involved in the overall process of the overall hierarchical factorization. In particular, the incorporation of these techniques for the FEM/BEM coupling is discussed in § 2.5.

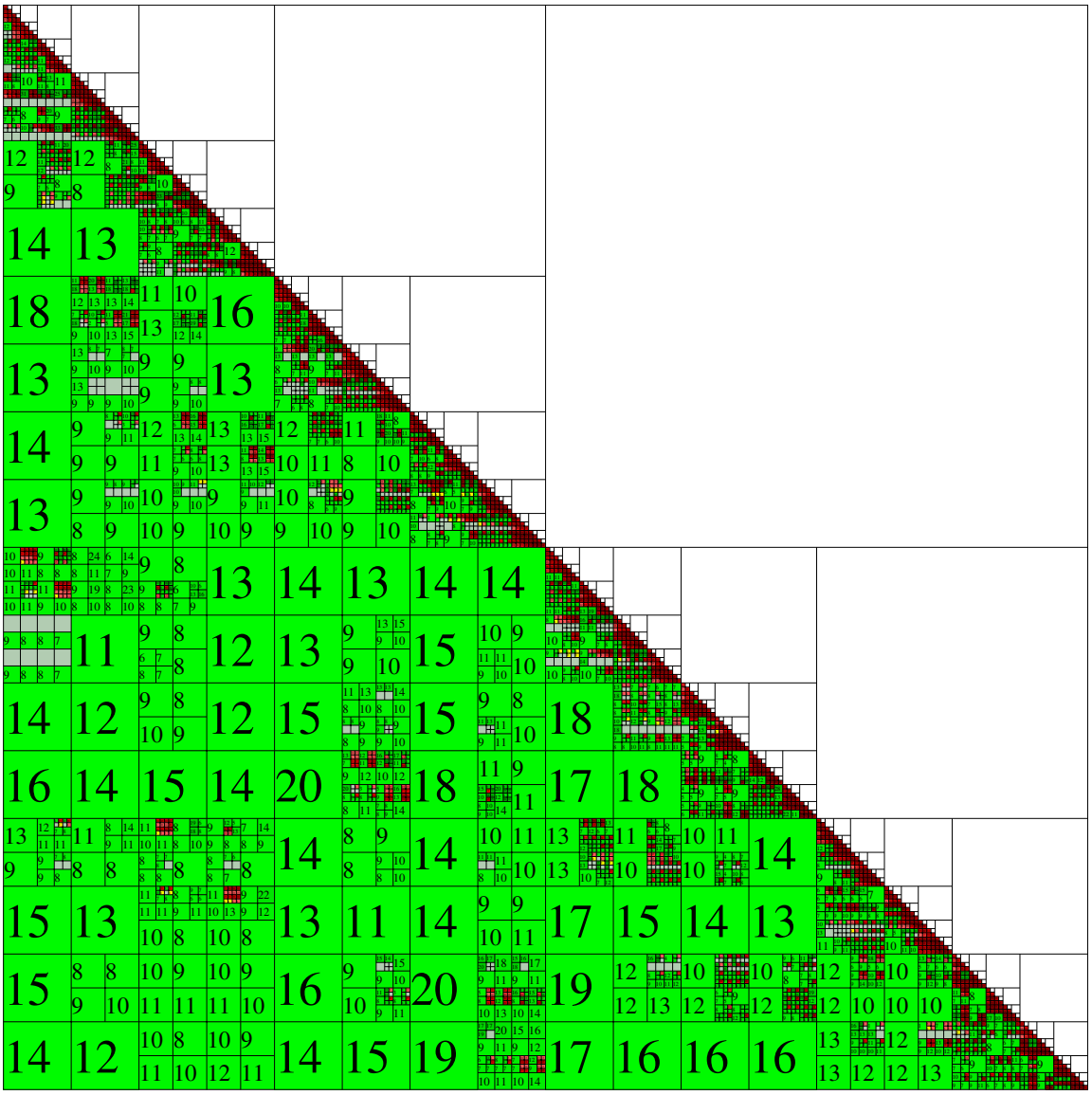
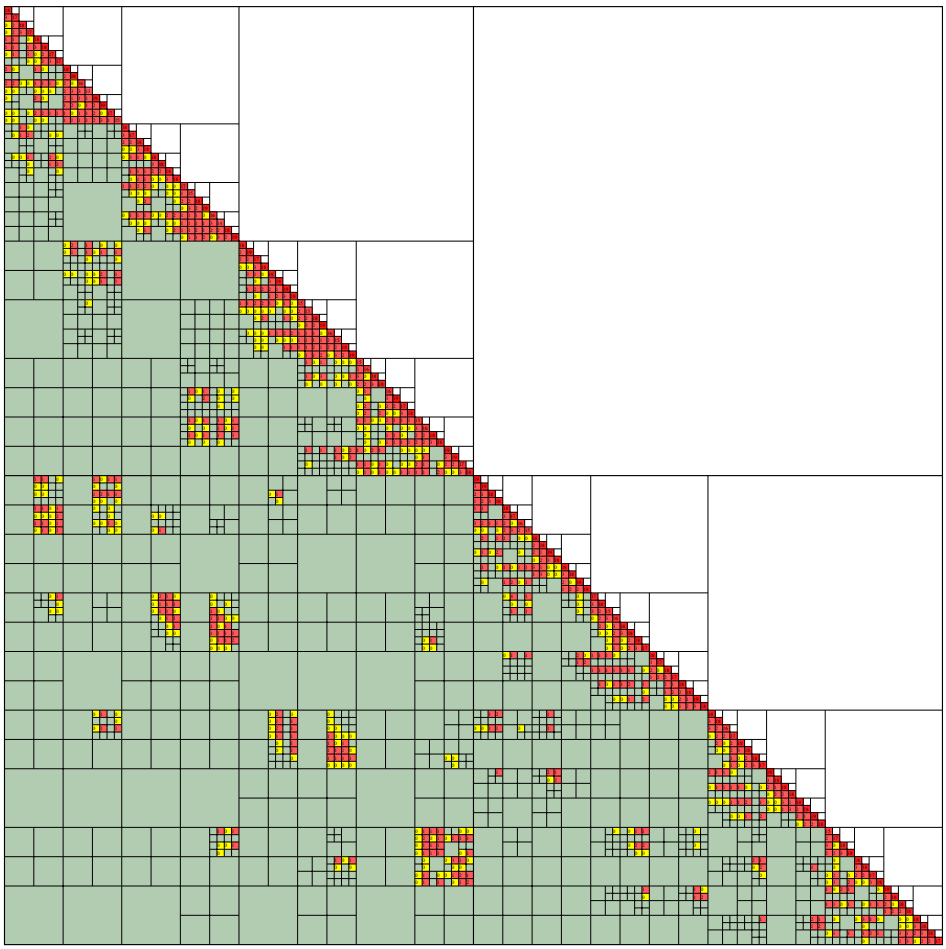
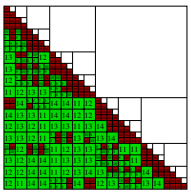


Figure 1.45: One symmetric $\mathcal{H}$-Matrix using recursive bisection constructed on the overall FEM/BEM coupling. Red blocks are Full-Matrices whereas green blocks are $\mathcal{R}k$-Matrices, in which the number indicates the rank.



(a) A_{vv}.

(b) A_{sv}.



(c) A_{ss}.

Figure 1.46: Coupling system split into three $\mathcal{H}$-Matrices (four in asymmetric case) using recursive bisection. Red blocks are Full-Matrices whereas green blocks are $\mathcal{R}k$-Matrices, in which the number indicates the rank.

Chapter 2

Low-Rank Compression in Sparse Linear Systems

In § 1.4, we have presented multiple numerical methods to compute the solution of the linear systems introduced in § 1.1.2. In particular, § 1.3 introduced the specificities and optimization techniques used for sparse systems arising from FEM discretizations. The sparse system is therefore characterized in this chapter with the equation

$$Ax = b, \tag{2.1}$$

where $A \in \mathbb{C}^{n \times n}$ is sparse and contains $nnz(A)$ non-zero entries before factorization. We denote by $nnz(LU)$ the number of non-zeros after the factorization. Sparse methods usually aim for a storage in $\mathcal{O}(nnz(LU))$. On the other hand, hierarchical matrices ($\mathcal{H}$-Matrices) were introduced to the reader in § 1.2.3 as a direct method for the solution of dense linear systems using compression to reduce the arithmetic and memory complexities of the solver. Let us remind the reader that a dense problem is usually stored in $\mathcal{O}(n^2)$ and factorized in $\mathcal{O}(n^3)$ operations. Hierarchical matrices reduce these complexities to $\mathcal{O}(n \log n)$ for valid classes of matrices, including for example matrices arising from standard discretizations such as the BEM [114]. From these considerations, we might consider $\mathcal{H}$-Matrices capable of reducing the storage of sparse methods below $nnz(LU)$. This chapter hence focuses on the efficient solution of sparse systems using hierarchical methods. The hierarchical framework of this chapter relies on the $\mathcal{H}$-Matrices defined by Hackbusch [114] and introduced to the reader in § 1.2.3.

Over recent years, there have been several attempts to use (hierarchical) compression techniques for the factorization of sparse matrices. In this thesis, we distinguish two main communities aiming at this common objective, combining strategies arising from both communities.

On the one hand, the $\mathcal{H}$-Matrix community, which developed the hierarchical tools discussed in § 1.2.3, has introduced sparse techniques such as the nested dissection in hierarchical solvers, as we will see in § 2.1. For $\mathcal{H}$-Matrices to reach the promised near-linear complexity, the considered linear systems must have suited numerical characteristics leading to a data-sparse representation. Problems arising from BEM

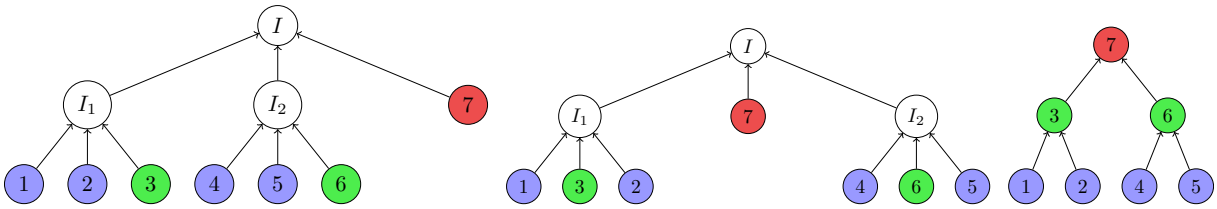
and FEM lead to such matrices (of which some submatrices have low-rank properties). Consequently, $\mathcal{H}$-Matrices are applicable to such linear systems. Yet, for sparse matrices arising from FEM, we have seen that fill-in can be reduced through various algorithms, of which we have introduced heuristics such as the nested dissection, AMF or AMD in § 1.3.1.1. Historically, $\mathcal{H}$-Matrices have been created in the context of discrete integral operators [112] and therefore did not initially rely on such techniques. Nonetheless, it has later been suggested to apply nested dissection for the clustering of $\mathcal{H}$-Matrices in [143]. Since then, many articles have shown a significant gain due to the inclusion of the nested dissection method in the construction of $\mathcal{H}$-Matrices [102, 105, 107, 126, 138, 193].

On the other hand, the Sparse-Direct community, from which originate many solvers and techniques optimized for sparse linear systems as discussed in § 1.3, have recently introduced compression in sparse solvers, as we will detail in § 2.2. The methods differ in their usage of hierarchical formats, relying for example on BLR, HSS or HODLR formats [17, 18, 25, 59, 95, 157, 164, 165, 188, 190, 191].

Fundamental Components

We first discuss concepts involved in the creation of an $\mathcal{H}$-Matrix that can be adjusted to fit more adequately the solution of sparse linear systems. They also can be compared to concepts used by the Sparse-Direct community. As an example, the division techniques used to create a cluster tree (§ 1.2.3.2) can be related to some reordering techniques (§ 1.3.1.1) such as the nested dissection and the corresponding separator tree. This tree is also related to the notion of quotient elimination tree. Nested dissection leads to a separation between subsets and this can effectively be represented as a tree. Using the $\mathcal{H}$-Matrix community vocabulary, the construction of a cluster tree using nested dissection leads to a ternary tree, of which each two first children are independent from one another. Using the Sparse-Direct community vocabulary, a separator tree based exclusively on nested dissection is binary and exhibits the non-dependence between computations of sibling nodes. In fact, the separator tree is also related to the separation tree structure presented in [170, Fig. 2.1]. The separation tree and the cluster tree associated with a nested dissection are indeed equivalent and represent the same structure, with the minor difference of children being ordered in a different manner. This is illustrated by Fig. 2.1. While the nested dissection leads to a partition noted (I_1, I_2, S) usually in graphs and examples in this thesis, the corresponding clusters are noted (τ_1, τ_2, τ_S) to highlight their hierarchical characteristics.

The **ordering** of unknowns is critical to limit the fill-in of sparse direct methods. Sparse direct solvers therefore perform a **reordering** step to reduce the fill-in. Multiple strategies may be employed to do so. They usually consist of a **graph partitioning** technique, such as the nested dissection, possibly refined with a local heuristic such as AMF or AMF. In the $\mathcal{H}$-Matrix community, the partitioning of the unknowns is referred to as the **clustering**. Different algorithmic procedures can be considered to perform this partitioning. The domain division techniques presented in § 1.2.3.2.2 as bisections can be related to the ordering or partitioning techniques such as nested dissection presented



(a) Cluster tree. (b) Separation tree [170, Fig. 2.1]. (c) Separator tree.

Figure 2.1: Classical tree structures used in the $\mathcal{H}$-Matrix community (a) and the Sparse-Direct community (c), respectively. The intermediate (b) is an alternative tree to (c), as discussed in [170, Fig. 2.1].

in § 1.3.1.1. **Bisections** compute an edge separator to bisect a graph in two subsets of unknowns. **Nested dissections** compute a vertex separator in order for the two subgraphs to be independent from each other and thus avoid fill-in between them. Other algorithms such as AMF or AMD may also be used to reduce fill-in; they are however preferred as local heuristics for local subdomains, i.e., sets too small so that they are not divided using nested dissection. Therefore, we may use nested dissection as a global heuristic to find a good separator (or cluster) tree, i.e., sufficiently broad and short. Then a local heuristic such as AMF or AMD can be used to efficiently compute a local ordering. Bisection also leads to a good matrix partition in terms of compression for dense matrices. It is consequently used on the separators, as the interactions of a separator with another (or itself) generally lead to dense submatrices in the factors. The overall ordering should eventually consist in a list of **supernodes** or **clusters** which partition the whole mesh/graph.

The **matrix** can then be assembled following this ordering or cluster tree. Sparse matrices usually rely on a **symbolic factorization** to create matrix blocks fitting the sparse pattern inherent to the problem. Based on the supernodes previously mentioned, the symbolic factorization must provide a way to locate permanent zeros (that will remain so until the end of the computations) so that we may avoid their storage, or, from another perspective, it must locate the fill-in generated by factorization. **Compression** can greatly influence the efficiency of the solver through the representation of large matrix blocks into a product of matrices in a low-rank format such as the $\mathcal{R}k$-Matrices.

In the case of $\mathcal{H}$-Matrices, the ordering is based on a **hierarchy**, in a structure named cluster tree, to be able to benefit from large compressions as well as from a fine granularity when suited. The matrix is therefore constructed following the hierarchy of a row cluster tree and column cluster tree. An **admissibility condition** decides then the format of a submatrix, which can be either an $\mathcal{H}$-Matrix, a $\mathcal{R}k$-Matrix or a Full-Matrix.

All these components may have an influence on the efficiency of the solution of Eq. (2.1). In this chapter, we first review the contributions of the $\mathcal{H}$-Matrix community in § 2.1 and the Sparse-Direct community in § 2.2 to design efficient sparse solvers enhanced with low-rank compression. We then investigate how to find a sparser structure adequate

for the storage of non-zeros in § 2.3. The use of a symbolic factorization is then discussed in § 2.4. Finally we open the discussion to the FEM/BEM coupling in § 2.5.

2.1 Hierarchical Low-Rank Algorithms Extended to Sparse Matrices

$\mathcal{H}$-Matrices can reduce the memory consumption and computation time by means of compression if the physical problem has low rank properties. More precisely, it can reduce memory storage from $n \times n$ to $\mathcal{O}(n \log n)$ for dense linear systems of n unknowns [114]. However it may prove inefficient on sparse linear systems. Indeed, an efficient sparse solver has a low memory consumption due to the usage of symbolic factorization: $\mathcal{O}(nnz(LU))$, where $nnz(LU) \ll n^2$ is the number of non-zeros in the factorized matrix (reduced by the usage of nested dissection).

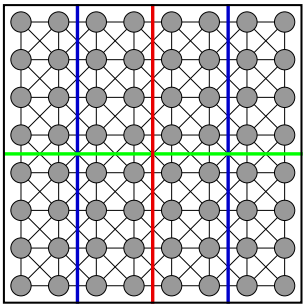
Literature on this subject shows that $\mathcal{H}$-Matrices can also take advantage of the characteristics of a sparse problem. [149] first exploited the sparsity of a problem in the context of $\mathcal{H}$-Matrices. Later, $\mathcal{H}$-Matrices combined with nested dissection showed a significant advantage over usual bisection-based $\mathcal{H}$-Matrices [102]. We review here the characteristics of the methods developed by the $\mathcal{H}$-Matrix community that may be used on sparse linear systems.

2.1.1 $\mathcal{H}$-Matrices Based on Bisection

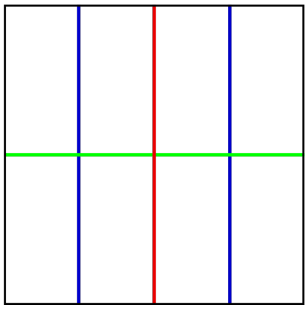
$\mathcal{H}$-Matrices have originally been designed to be used on dense linear systems, as mentioned in § 1.2.3. Therefore their construction is based on a pure bisection approach, following the principles of divide and conquer algorithms. At each step of the domain division method used to create a cluster tree (on which will be based an $\mathcal{H}$-Matrix), the two newly found subsets are separated by an edge separator. This is illustrated in Fig. 2.2 on a square mesh. The cluster tree that arises from the bisections in Fig. 2.2b and 2.2c is shown in Fig. 2.2d. This cluster tree leads to the block cluster tree depicted in Fig. 2.2e. An example of an $\mathcal{H}$-Matrix constructed with bisection applied on the Laplacian cube from Fig. 1.17 is shown in Fig. 2.3. The initial $\mathcal{H}$-Matrix, before factorization, is shown in Fig. 2.3a, and the factorized $\mathcal{H}$-Matrix is shown in Fig. 2.3b. We can see in this last figure that fill-in is largely compressed (compressed blocks are colored green). Moreover, the vast majority of the matrix is still filled with zeros indicated by the $\mathcal{R}k$-Matrices of rank 0 (colored light green and without numbers), representing for example nearly 80% of the 8000-unknowns Laplacian cube problem. Therefore, we must wonder how this compression compares with reordering techniques such as the nested dissection.

2.1.2 $\mathcal{H}$-Matrices Combined with Nested Dissection

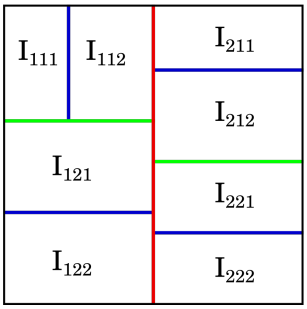
We have seen in § 1.3.1.1.2 a method named nested dissection that recursively partitions a graph in two independent clusters using a vertex separator, with the aim of minimizing



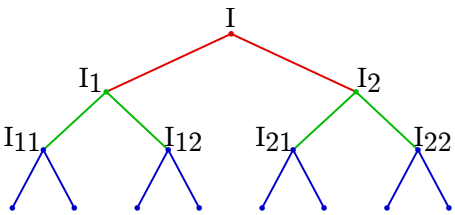
(a) Perfect bisection of an adjacency graph associated with a 8×8 mesh.



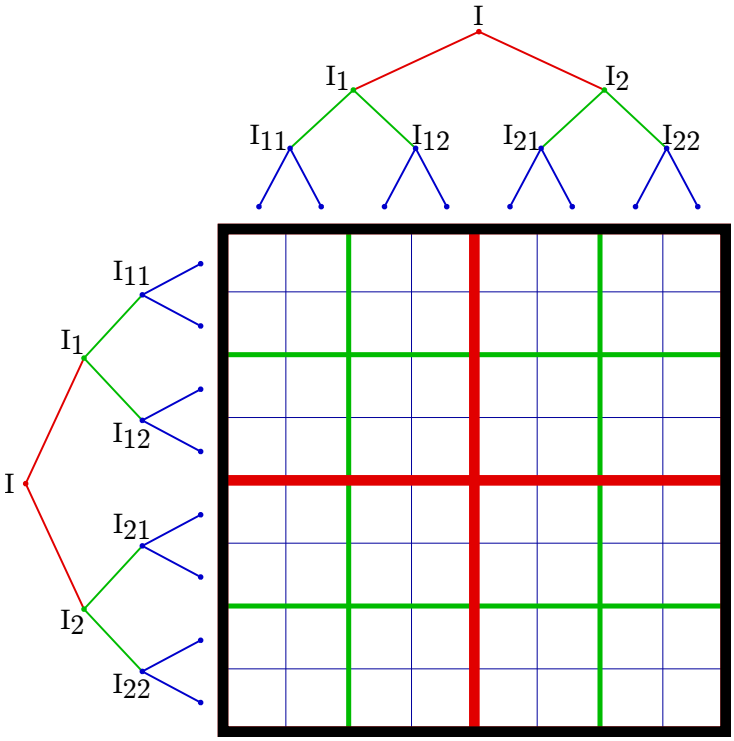
(b) Perfect Bisection without the graph.



(c) Bisection resulting in an unbalanced partitioning.

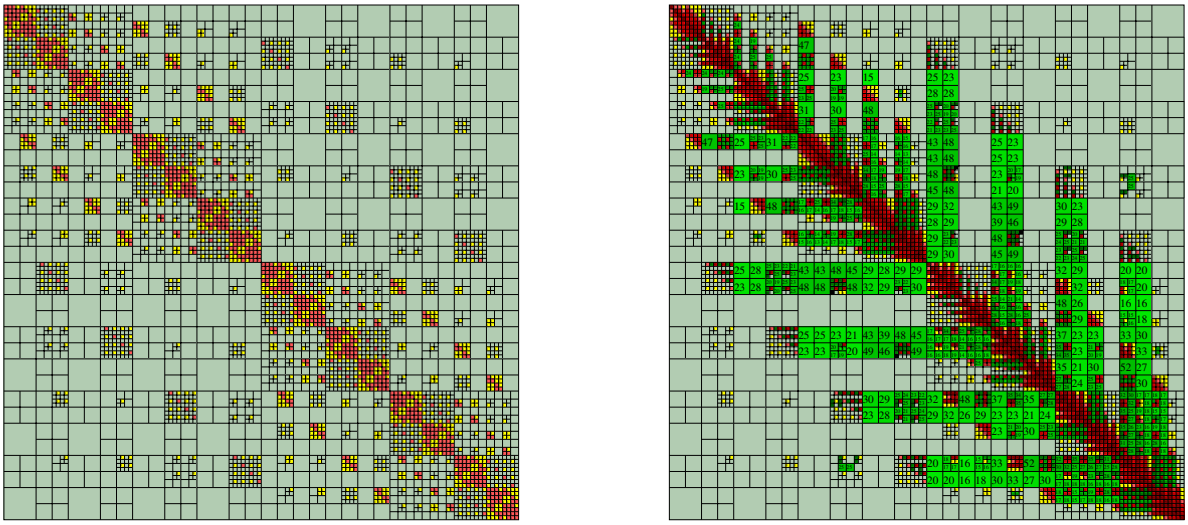


(d) Binary cluster tree based on the hierarchical bisection of (c).



(e) Two binary cluster trees lead to a block cluster quadtree.

Figure 2.2: Bisection would lead to a perfect partitioning on a cube, as illustrated in (a), and, for visual reasons without the mesh, to the edge separators shown in (b). However, in the general case, for irregular shapes such as those used in an industrial environment, bisection could lead to an irregular partitioning as illustrated in (c). The first separator is colored red, the two separators of the following level are colored green, and those of the third level are colored blue. In following figures representing meshes, the mesh will usually be hidden.



(a) Sparse matrix before factorization, with a bisection clustering.

(b) Matrix after factorization, with a bisection clustering.

Figure 2.3: We can see the fill-in induced by factorization is compressed (green blocks). See legend in Fig. 2.4 for more details.

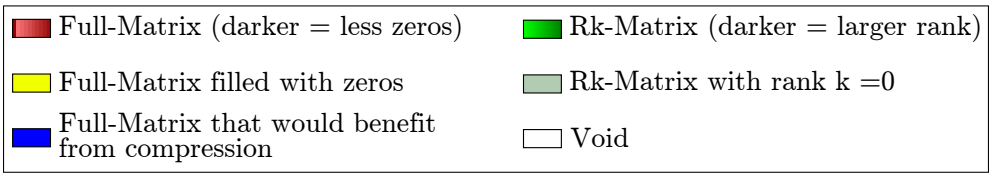
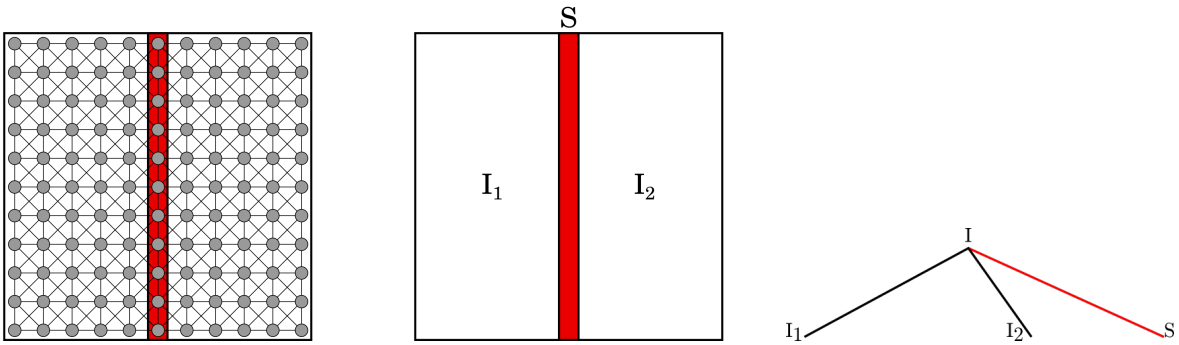


Figure 2.4: Legend indicating the signification of colors for submatrices in all following $\mathcal{H}$-Matrices. Each submatrix may be stored either as a Full-Matrix (dense format), as a $\mathcal{R}k$-Matrix (compressed format) or not be stored at all (void). The blue color is used only in Fig. 2.20.

the fill-in generated by the factorization. Thus, instead of the classical recursive bisection, nested dissection can be applied in order to find a ternary cluster tree and causes large blocks of zeros to appear inside the resulting $\mathcal{H}$-Matrix as well as in its factors. The $\mathcal{H}$-Matrix community used this approach under the denomination of Domain Decomposition or Nested Dissection. To give an overview of the progresses made in this area of research, we base ourselves on the last survey of Hackbusch first published in 2015 [115, § 5.8] listing various studies carried out in this field [102, 126, 138, 143]. In [107], the authors propose a hierarchical solver relying on a black-box partitioning for methods unaware of the geometry of the problem, which is able to compute a nested dissection. Comparisons of $\mathcal{H}$-Matrix solvers using this technique are performed in [105, 193] using reference sparse solvers such as PARDISO or MUMPS.

The clustering of separators is discussed in the literature (see § 2.1.3); we will discuss this clustering more extensively in § 2.4.3. However, in the literature, the most frequently used method is the recursive bisection [138, § 3.3].

Let us give an example of such a construction. We start from a configuration similar to Fig. 1.26, i.e., a mesh divided using nested dissection. The first step of the division of the mesh I is shown in Fig. 2.5, leading to the red separator S between I_1 and I_2



(a) First level of a recursive domain division using nested dissection over a 11×11 mesh.

(b) Domains (separator S and subdomains I_1 and I_2) found using nested dissection, without the mesh.

(c) Ternary cluster tree based on a nested dissection.

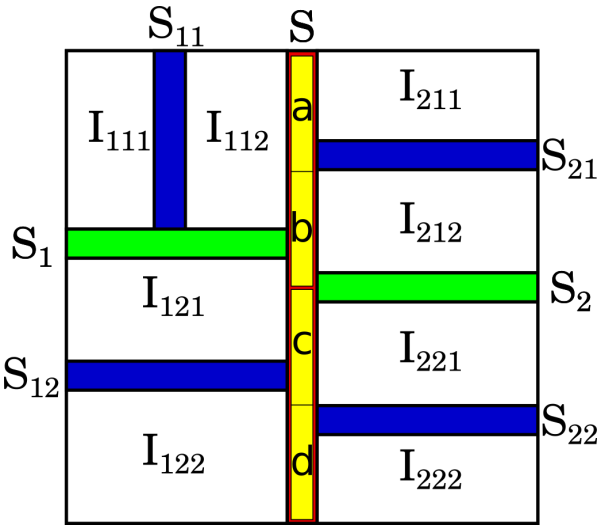
Figure 2.5: One step of nested dissection.

(Fig. 2.5a shows a 11×11 mesh, but it can be generalized to larger meshes, and the mesh is thus disregarded in Fig. 2.5b). This leads us to a ternary-based cluster tree as shown in Fig. 2.5c. After this first division, I_1 and I_2 are recursively divided using nested dissection, while the separator S is divided into $\{a, b\}$ and $\{c, d\}$, and then $\{a, b\}$ divided into $\{a\}$ and $\{b\}$ as illustrated in the graph (Fig. 2.6a) and the cluster tree (Fig. 2.6b), leading to the block cluster tree in Fig. 2.6c. The interaction between I_1 and I_2 is null. Eventually, this leads to a sparse $\mathcal{H}$-Matrix such as depicted in Fig. 2.6d.

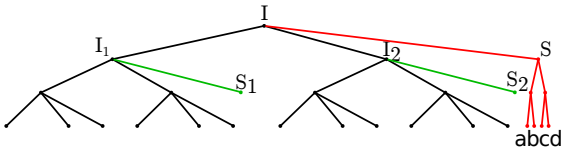
More generally, we start from the initial cluster $\tau = I$. Then τ is divided by nested dissection into subdomains τ_1 and τ_2, and a separator τ_S, via the function `NestedDissection(`τ`)`. Each subdomain cluster is then divided using nested dissection whereas all separators are divided using bisection via the method `Bisection(`τ`)` until we reach the size limit N_{ND}.

Definition 2.1. N_{ND} is the minimum size of a cluster on which the nested dissection is performed. Below that threshold, other strategies may be applied.

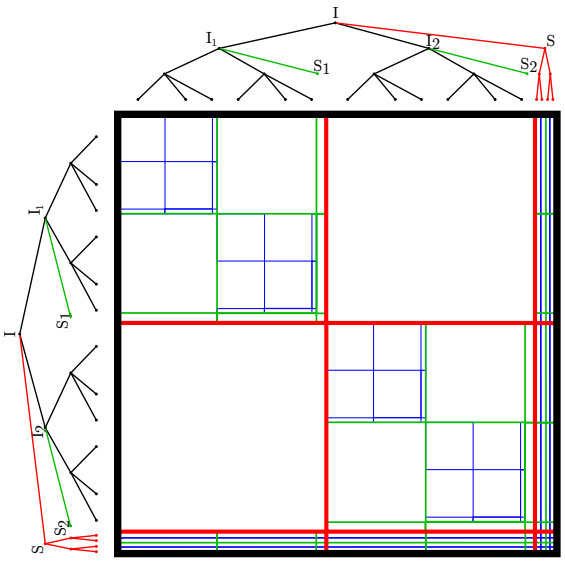
Furthermore, τ_1 and τ_2 are arranged in arbitrary order but τ_S must be ordered after them. The corresponding recursive algorithm is shown in Algorithm 25, where the function `IsSparse(`τ`)` is defined to return $True$ if a cluster is not a separator (or one of its descendant). The size N_{leaf} of the leaf clusters is not a priori equal to the size N_{ND} determining when to stop the nested dissection algorithm. Below the threshold N_{ND},



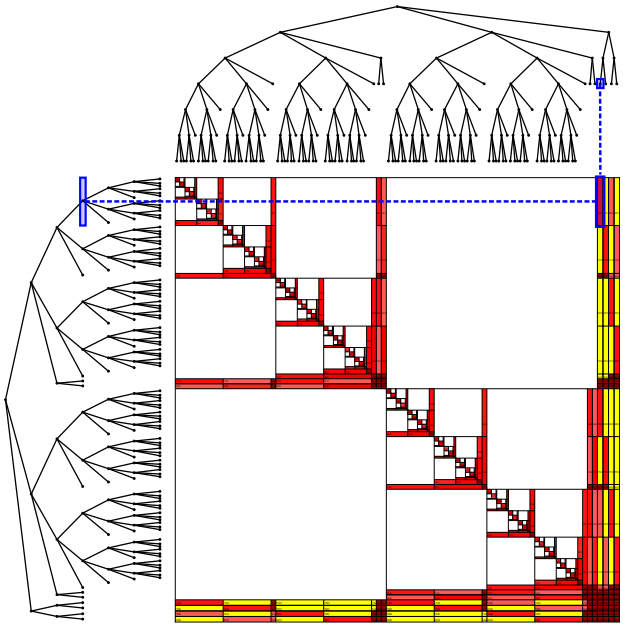
(a) Adjacency graph.



(b) Cluster tree.



(c) Block cluster tree. The maximum arity of the tree is 9.



(d) $\mathcal{H}$-Matrix applied on the Laplacian cube (Fig. 1.17). One tall & skinny block is highlighted in blue. See legend in Fig. 2.4.

Figure 2.6: Construction of an $\mathcal{H}$-Matrix using nested dissection with bisection applied on separators.

Algorithm 25: Construction of a cluster tree based on nested dissection, coupled with bisection applied on separators.

```
Function CreateClusterTree(τ)
1   if |τ| ≤ N_leaf then
2       Children(τ) = ∅                          ▷ The cluster τ will be a leaf
3   else
4       if IsSparse(τ) ∧ |τ| ≥ N_ND then
5           (τ_1, τ_2, τ_S) ← NestedDissection(τ)
6           CreateClusterTree(τ_1)
7           CreateClusterTree(τ_2)
8           CreateClusterTree(τ_S)
9           Children(τ) = {τ_1, τ_2, τ_S}
10      else
11          (τ_1, τ_2) ← Bisection(τ)
12          CreateClusterTree(τ_1)
13          CreateClusterTree(τ_2)
14          Children(τ) = {τ_1, τ_2}
```

reordering algorithms may be used, such as the recursive bisection or a local heuristic such as AMF or AMD, until a smaller size N_{leaf} is reached. However, in the literature, N_{ND} is equal to N_{leaf} as far as we know, due to the fact that no local heuristic is used. The functions `NestedDissection(τ)` and `Bisection(τ)` may be either geometrical or topological, as we will see in § 2.3.1.

2.1.3 Separator Clustering in the $\mathcal{H}$-Matrix Literature

As we have mentioned earlier, in order to benefit from an efficient compression, recursive bisection is usually preferred for the clustering of separators in the literature [138, § 3.3], possibly modified following the rules detailed in § 2.1.5.

However, in [53], the authors investigate a way to avoid the computation of weak interactions between a separator and its neighbors. They subdivide the separators in order to be able to better separate these interactions, in a way similar to the algorithms discussed in § 2.4.3.3.

2.1.4 Using Symbolic Information in Combination with a $\mathcal{H}$-Matrix Structure

In [126], the authors study the computation of an approximate Cholesky decomposition of a sparse matrix using $\mathcal{H}$-Matrices. They propose to rely on the elimination graph to compute a sparse block structure that will be used in the construction of an $\mathcal{H}$-Matrix

used for the decomposition. The computation of this sparse structure is similar to what is done by the Sparse-Direct community and presented in § 1.3.1.2. A block column symbolic factorization is used, under the name of "cluster-wise elimination" instead of a "vertex-wise elimination" (a scalar symbolic factorization), resulting in a block column structure. The "sparsity pattern" discussed in that paper relies on the "complete connection graph" associated with the problem. This structure is closely related to the notion of quotient elimination graph previously mentioned, or symbolic information. Based on this sparsity pattern, it can then be decided which blocks should be stored. In the paper, the authors propose to modify the construction of the block cluster tree in the following way. A block (σ, τ) is admissible and stored as a "Null"-type (zero) matrix if no edge connects σ and τ in the connection graph. The diagonal blocks are stored as Full-Matrices. The remaining non-zero off-diagonal blocks in the hierarchy of the $\mathcal{H}$-Matrix are stored in a low-rank format based on a QR decomposition. Also, the $\mathcal{H}$-Matrix seems to be initialized with Full-Matrices and low-rank matrices are only used during the Cholesky decomposition. Yet, this method is strongly related to the algorithms we discuss in § 2.4.2.

Not satisfied with the number of re-compression performed by this algorithm, another proposition of this paper is to use a method similar to multifrontal methods (§ 1.3.1.3.2). If we consider the dense frontal matrix represented as the 2×2 matrix F in Eq. (1.26), the authors then propose to use a low-rank representation to compress the matrix F_{21}.

2.1.5 Geometric Prevention of the Occurrence of Tall & Skinny Blocks

Let us consider a division of a cluster $\tau \in I$ using nested dissection, as detailed in Algorithm 25. When applying bisection to create a separator subtree, another problem arises from the difference between the sizes of each subdomain τ_1 and τ_2 and the separator τ_S. τ_1 and τ_2 are usually far larger than τ_S, as, for three-dimensional linear systems, they are also three-dimensional domains whereas τ_S is two-dimensional. For two-dimensional linear systems, the same applies, with I_1 and I_2 being two-dimensional and τ_S one-dimensional. In later divisions, τ_1 and τ_2 are subdivided nearly in two, and τ_S is divided in exactly two subsets at each recursion, thus propagating the difference of size between the descendants of τ_S and their counterparts of the same level. This gives rise to "tall & skinny" blocks in the resulting matrix, as illustrated in Fig. 2.6d. A large number of off-diagonal blocks are very flat (bottom-left) or tall (top-right) due to this difference in size. The unknowns located under the separator τ_S are very few in comparison to the number of their siblings τ_1 and τ_2.

[114, §9.2.4] details a different construction of the cluster tree to avoid this distortion by dividing separators subtrees in a new way, based on the geometrical dimensions of these clusters. The notation $T_d(I)$ is used to refer to cluster trees in d dimensions, i.e., non-separator subdomains. $T_{d-1}(I)$ refers to cluster trees in $d-1$ dimension, i.e., separators. Then, following this notation, a cluster $\tau \in T_d(I)$ is divided using nested dissection into $\tau_1 \in T_d(I)$ $\tau_2 \in T_d(I)$ and $\tau_S \in T_{d-1}(I)$. For $d = 2$, a cluster $\tau \in T_{d-1}(I)$ is

then divided using bisection only one every two levels, following the value of the function

$$\varkappa(\tau) = \min\{\texttt{level}(\tau) - \texttt{level}(\tau') \ / \ \tau' \in T_d(I) \ \texttt{ancestor of} \ \tau\}.$$

If $\varkappa(\tau)$ is odd, τ remains unchanged, else it is divided using bisection. Indeed, bisection divides the diameter of clusters by about $\sqrt{2}$ each level for a dimension $d = 2$ (consequently, by 2 every two levels) and by 2 each level for $d = 1$, so using $\varkappa$ to decide to divide or not clusters ensures the same division ratio applies on both clusters from $T_d(I)$ and $T_{d-1}(I)$, i.e., a division by 2 every two levels. For three-dimensional linear systems, this formula can then be changed accordingly. The diameter of a cluster $\tau \in T_3(I)$ is divided by (approximately) 2 every three levels whereas the diameter of a cluster $\tau' \in T_2(I)$ is divided by 2 every two levels. Therefore one should arrange the clusters division of $T_2(I)$ to take place two out of three times using this technique. The first levels of this method for two-dimensional linear systems are shown in Fig. 2.7. This can be

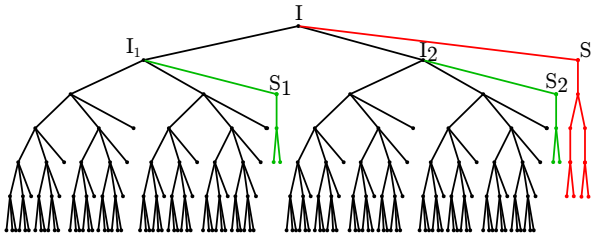


Figure 2.7: Cluster tree skipping the separator S division every one level out of two.

seen as a **geometry**-preserving method in that the matrix blocks constituting the final $\mathcal{H}$-Matrix will therefore have row and column clusters with closer geometric dimensions.

2.2 Sparse Direct Solvers Using Compression Techniques

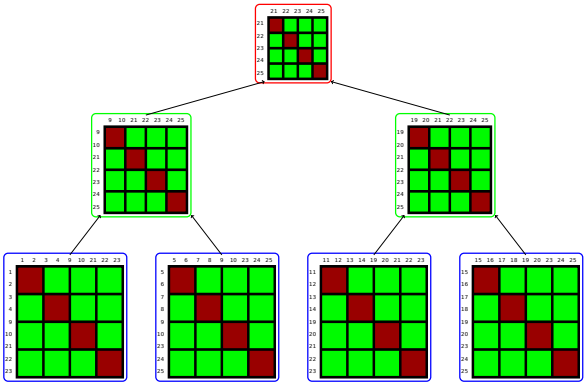
In the same manner the $\mathcal{H}$-Matrix community has introduced sparse-specific techniques into the $\mathcal{H}$-Matrix arithmetic for the solution of sparse linear systems, the Sparse-Direct community has been conducting research on the introduction of low-rank compression and hierarchical techniques to lower the computational requirements of sparse direct solvers.

2.2.1 Compression Formats in Sparse Solvers

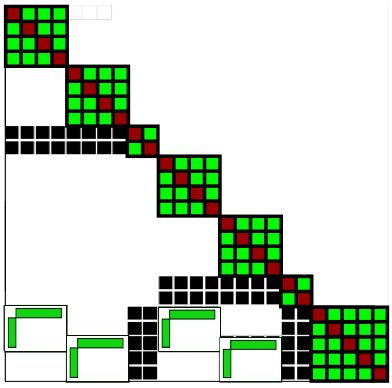
Low-rank compression techniques have successfully been introduced in multifrontal and supernodal solvers. The compression is applied differently depending on the factorization method employed.

Multifrontal methods focus on the compression of frontal matrices and Schur complements involved in the updates of the assembly tree. Multiple hierarchical formats (§ 1.2.3.4) have been studied for such a compression. For example, HSS-Matrices have

been incorporated inside multifrontal methods [190, 191] for the compression of frontal matrices. They have also been introduced in the STRUMPACK software package [95]. In [25], the HODLR format is used instead of the HSS format. Likewise, researchers from the MUMPS [17, 18, 157, 188] solver have studied the impact of single-level hierarchical matrices (referred to as BLR) to compress frontal matrices. We will refer to this solver as BLR-MUMPS in this thesis. A simplified example illustrates the use of BLR for the compression of frontal matrices in Fig. 2.8a.



(a) Simplified representation of frontal matrices compressed using BLR format with a 4×4 partitioning. Every off-diagonal block is here stored in a low-rank representation (in green).



(b) Simplified representation of a supernodal storage of which some submatrices are stored into a low-rank format and diagonal blocks are compressed using BLR.

Figure 2.8: Example of low-rank matrices usage in sparse direct methods.

Supernodal methods instead focus on the compression of separator diagonal blocks or off-diagonal blocks that are sufficiently large to benefit from compression. The authors of [59] introduce the use of off-diagonal compression (HODLR format) on diagonal blocks and use low-rank compression on submatrices arising from the interactions of two separators. The use of single-level hierarchical matrices (BLR) has also been introduced in the supernodal context [164, 165] for the PaStiX solver, where separator diagonal blocks, as well as off-diagonal blocks considered large enough, are compressed using this format.

Once submatrices are stored in a low-rank format, one may also discuss the possibility of relying on low-rank arithmetic. This is the subject of [164, § 3.2] or the LUAR variant developed in BLR-MUMPS [157].

Furthermore, modifications have been proposed regarding the order of the operations involving compressed submatrices. For example, in the context of multifrontal methods, the authors in [17] introduced a variant called FSCU, for Factor, Solve, Compress and Update, which performs a right-looking factorization where each submatrix impacted by the contribution of the elimination of a supernode may be compressed before applying the updates (see Eq. (1.11) in § 1.3.1.1 based on Algorithm 2). In [157], a variant called UFSC is discussed, the corresponding left-looking variant of FSCU, as well as variants UFCS, UCFS and CUFS, which, as their names suggest, compress at different times of the algorithm. In the context of supernodal solvers, [164] studies the impact of compressing

all matrix blocks at the start of the factorization (the Minimal Memory strategy) or compressing blocks after all contributions have been applied to it (the Just-In-Time strategy).

With respect to these methods, the compression scheme employed in the hierarchical solver considered in this thesis relies on the admissibility condition described earlier in § 1.2.3.2.3 and therefore compresses each submatrix satisfying the admissibility condition before factorization. Each block remains compressed until the end of the factorization or if its storage surpasses that of the dense format.

Other elements of comparison may include the hierarchical format involved. Indeed, we have seen that HSS or BLR or $\mathcal{H}$-Matrices are used in sparse solvers for example. While BLR offers an easier implementation (by tile), the lower complexity of $\mathcal{H}$-Matrices is more interesting for larger matrices. However, linear systems with more than two million unknowns may not be large enough for the low exponent of $\mathcal{H}$-Matrices complexity to compensate its larger constant, as shown by [188, § 2.3.1]. For a problem with n unknowns, the authors write the $\mathcal{H}$-Matrix arithmetic complexity under the form $\alpha_H n^{\beta_H}$ and the BLR complexity under the form $\alpha_B n^{\beta_B}$ and show examples where $\alpha_H > \alpha_B$ while $\beta_H < \beta_B$. Therefore, $\mathcal{H}$-Matrices may necessitate even larger linear systems for their near-linear complexity to counterbalance this constant. That study also shows examples where HSS-Matrices are more interesting than $\mathcal{H}$-Matrices for low precision requirements, while the reverse is true for higher precisions.

2.2.2 Separator Clustering in the Sparse-Direct Literature

As we have mentioned earlier, in order to benefit from an efficient compression, the $\mathcal{H}$-Matrix community rely on a separator clustering based on recursive bisection [138, § 3.3]).

In the Sparse-Direct community, the ordering of separators is often discussed with the goal of minimizing communications or the number of off-diagonal blocks. For example, the authors of [180, chapter 7] introduced strategies reducing the communications in the MUMPS solver. In [166], the authors discuss methods minimizing the number of blocks based on TSP (Traveling Salesman Problem) heuristics. SCOTCH [162] proposes strategies of which we give example in § 2.4.4.2 that will later be used in our experiments. In [188], the block clustering for the BLR-MUMPS solver is discussed. Two strategies are proposed. The *explicit clustering*, which clusters each front independently, leads to good blocking sizes but each variable is assigned to a different reordering in each front. The *inherited clustering*, which assigns the same clustering for each front based on the previously computed separator clustering, leads to irregular blocking sizes. This inherited clustering has then been developed using the algebraic concepts from the black box partitioning introduced in [107].

The Sparse-Direct community has also investigated separator orderings based on the rest of the nested dissection in an effort to produce blocks satisfying low-rank characteristics. In [164, chapter 5] for example, a clustering based on the interactions with separators from close levels in the separator tree is investigated. If we focus on the

first separator for example, the idea is to first select the vertices corresponding to the traces of the second and third separator, i.e., the vertices connected to these separators. If some vertices are then isolated between these traces, they are aggregated together within the traces until a specified size threshold has been reached. Once this pre-selection has been made, the rest of the vertices is clustered using a K-way partitioning, i.e., an equivalent to recursive bisection. This method therefore uses the same approach as described in § 2.4.3.3 in the top-down approach (see Fig. 2.43, a method introduced first in [11] and also presented in [84]) for the first levels of recursion and then switches to recursive bisection for the rest of the recursion.

2.3 Investigation of a Sparser Structure for $\mathcal{H}$-Matrices

We now discuss how to exploit sparse techniques within a hierarchical solver, starting from nested dissection. Sparse solvers indeed exploit the sparsity of sparse linear systems to reduce their memory usage and reach a complexity of $\mathcal{O}(nnz(LU))$, as mentioned previously in § 2.1, where $nnz(LU)$ is the number of non-zeros in the factors. We thus intend to give an overview of the problems and possible solutions for the adaptation of hierarchical methods to sparse problems.

2.3.1 Global Clustering Based on Nested Dissection

We distinguish three types of clusterings resulting from nested dissection methods. The

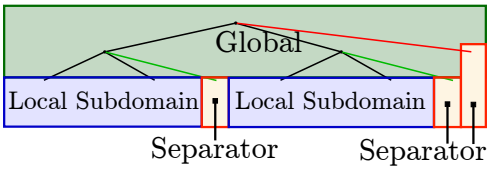


Figure 2.9: Categories of clusterings used on a cluster τ following the position of τ in the cluster tree. The global clustering corresponds to the computation of the nested dissection here, while a local subdomain strategy may be used to cluster subdomain leaves and a local separator strategy to cluster separators.

global clustering is the computation of the nested dissection in itself. The **local subdomain clustering** is the possible clustering used on the subdomain leaves of the nested dissection (leaves in Fig. 1.26b). With the notations used in § 2.1.2, it corresponds to applying a new clustering strategy on sparse-labeled clusters if their size is under the threshold N_{ND} (Definition 2.1). The **local separator clustering** is the clustering used in the separators. These categories are shown in Fig. 2.9 with respect to their location in the cluster tree arising from nested dissection. We here discuss global and local subdomain clusterings. The local separator clustering is discussed in § 2.4.3. The global clustering (computing a nested dissection) may be performed either in a "geometrical" fashion, based on the space coordinate of the unknowns in the physical mesh of the problem, or in a

"topological" fashion, using graph partitioning tools such as SCOTCH [62] applied to the adjacency graph of the sparse matrix.

2.3.1.1 Geometric Nested Dissection (BBox ND)

As the geometry of the problem is known, we can perform the domain division into separators and subdomains using the same procedures as the bisection methods (§ 1.2.3.2.2) slightly adapted to compute not an edge separator but a vertex separator. These bisections rely on bounding boxes and therefore the space coordinates of the unknowns. The bisection method may be median, geometric or hybrid, though we have focused on the median bisection in this thesis. The algorithm used in our application to compute such a geometry-based separation is detailed in Algorithm 26. A bounding box bisection is first applied to decompose τ into two subdomains τ_1 and

Algorithm 26: Division of a cluster into three subdomains (two non-separators and one separator) based on a prior bisection.

Function `GeometricSeparation(`τ`)`

1 $(\tau_1, \tau_2) \leftarrow$ `Bisection(`τ`)` ▷ Bounding-box based
2 $\tau_S \leftarrow \emptyset$
3 traversingEdges $\leftarrow \emptyset$
4 **for** $(i \in \tau_1) \wedge (j \in \texttt{Adj}_G(i)) \wedge (j \in \tau_2)$ **do**
5 traversingEdges $\leftarrow$ traversingEdges $\cup (i, j)$
6 **for** $(i, j) \in$ traversingEdges **do**
7 **if** `pickFromLeft()` **then**
8 $\tau_S \leftarrow \tau_S \cup i$
9 $\tau_1 \leftarrow \tau_1 \setminus i$
10 traversingEdges $\leftarrow$traversingEdges $\setminus \{(k, \ell) \mid k = i\}$
11 **else**
12 $\tau_S \leftarrow \tau_S \cup j$
13 $\tau_2 \leftarrow \tau_2 \setminus j$
14 traversingEdges $\leftarrow$traversingEdges $\setminus \{(k, \ell) \mid \ell = j\}$
15 **return** (τ_1, τ_2, τ_S)

τ_2, which can also be formulated as the computation of an edge separator between the two subdomains. Once the edges traversing between τ_1 and τ_2 are computed (Algorithm 26 lines 4-5), we may pick endpoints from this edge separator and thus create a vertex separator. There are multiple ways to decide which endpoint to pick from each edge, which is the purpose of the function `pickFromLeft()` here.

Among the considered implementations of this function, we may decide to keep:

1. approximately an equivalent number of unknowns from the two subdomains τ_1 and τ_2 by alternatively picking one vertex from one cluster and the following from the other cluster;
2. the first half vertices from τ_1 and the second half from τ_2;
3. only the vertices from τ_1, thus creating an imbalance between the left and right subdomains but leading to a perfectly planar separator (if the edge separator is planar);
4. both endpoints for each edge, leading to a double separator, in the sense that two layers of vertices separate the left subdomain from the right one.

An example of geometric nested dissection using the first method is given in Fig. 2.10a. For the experiments, we retain the last two methods (3 and 4). The third method is simply referred to as **BBox ND** while the last method is referred to as **BBox ND 2**.

2.3.1.2 Topological Nested Dissection (Scotch ND)

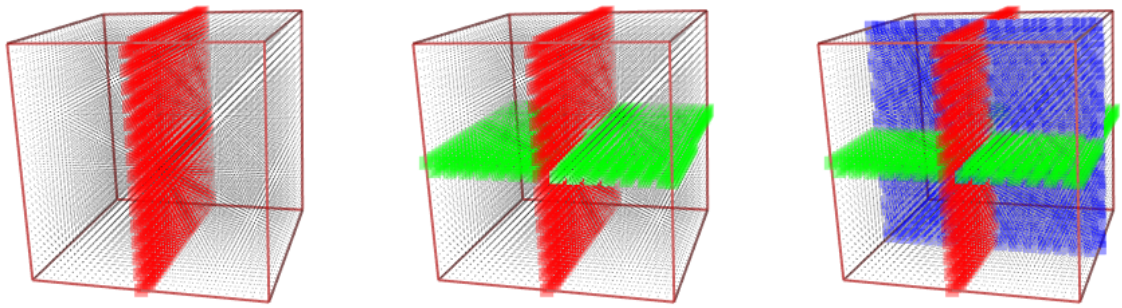
Topological tools such as SCOTCH [62] or METIS [132] are often used for graph partitioning and a fortiori for nested dissection. This usually applies to solvers that do not have the geometric information necessary to compute a geometric-based nested dissection such as discussed in the previous paragraph. However, due to the quality of performance in time computation as well as in the resulting separator tree, we will also use these tools as a control over the quality of the computed geometric nested dissection. In this thesis, the strategy given to SCOTCH to compute a nested dissection is given in Algorithm A.1 (see appendix A). An example of topological nested dissection using this SCOTCH strategy is given in Fig. 2.10b.

2.3.1.3 Hybrid Clustering Combining Nested Dissection and Minimum Fill (Scotch ND+)

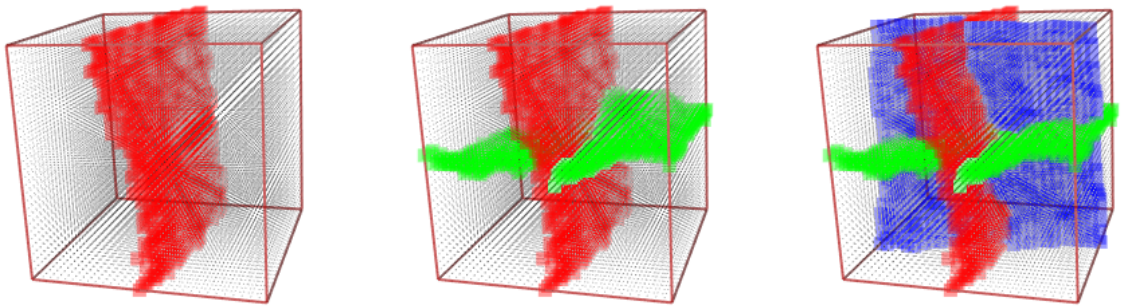
Using nested dissection as a global clustering to partition a domain into smaller independent subdomains is only one of the many existing clusterings used in sparse solvers which we have introduced in § 1.3.1.1.2. To further reduce fill-in in the local subdomains of the separator tree resulting from nested dissection, one may rely on the use of local orderings, preferred for smaller matrices. Therefore, we consider in this thesis the coupling between a nested dissection as a global clustering and the Minimum Fill method as a local clustering on local subdomains. To compute the global and local clustering, we rely on SCOTCH [62]. The strategy given to SCOTCH to compute this coupling is given in Algorithm A.2 (see appendix A).

2.3.2 Preventing the Occurrence of Tall & Skinny Blocks

Following the discussion of § 2.1.5 on the literature's approach to minimize tall & skinny blocks, we introduce here some alternative ways to prevent tall & skinny blocks. The usual approach relies on skipping divisions in the separator subtree to maintain a **geometric**



(a) Nested dissection using Bounding-Boxes (BBox ND).



(b) Nested dissection using SCOTCH (Scotch ND).

Figure 2.10: Three levels of nested dissection using Bounding-Boxes and using SCOTCH.

balance between clusters from the separator and the ones associated with subdomains (non-separator clusters). We discuss here a similar approach aiming at preserving an **algebraic** balance between the size of each cluster, i.e., their cardinality, instead of their geometric diameter. To this end, let us consider the situation of Fig. 2.5b and 2.5c. The separator S is of a lesser dimension than I_1 and I_2 (one-dimensional if the problem is two-dimensional) and its cardinality is therefore smaller. We will use the notation $\text{Desc}_\ell(\tau)$ to indicate the descendants of τ at a specific level ℓ. We will also use the notation $\text{SparseDesc}_k(\tau_1)$ indicating the subdomain (non-separator) descendants of τ_1. The problem is formulated as follows:

Problem 1. Let (τ_1, τ_2, τ_S) be a partition computed through nested dissection with $|\tau_S| \ll \min(|\tau_1|, |\tau_2|)$. Find k such that skipping k divisions in the separator τ_S subtree leads to its cardinality and that of its descendants matching approximately the cardinalities of the descendants of τ_1 and τ_2. This may be formulated as:
$\forall \sigma_S \in \text{Desc}_\ell(\tau_S), \forall \sigma_1 \in \text{Desc}_\ell(\tau_1), \forall \sigma_2 \in \text{Desc}_\ell(\tau_2), |\sigma_S| \sim |\sigma_1| \sim |\sigma_2|$.

The dimensions of the problem are shown in Fig. 2.11.

We proceed in two steps.

The **first step** consists in establishing a relation between the mean cardinality of the descendants of the separator τ_S and the mean cardinality of non-separator clusters descendants of τ_1 and τ_2, i.e., the other separators are put aside for now. We suppose first

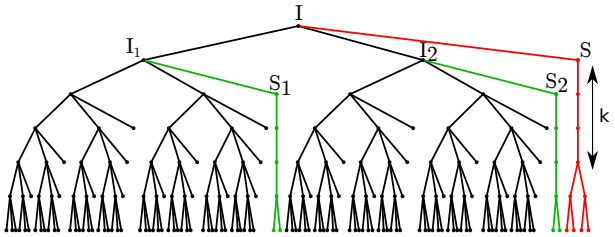


Figure 2.11: Cluster tree skipping the division of separator S for k levels in order for the cardinality of S to be equivalent to those of the descendants of I_1 or I_2.

that

$$|\tau_1| \sim |\tau_2|. \tag{2.2}$$

The cardinality of the clusters in branches of the cluster tree are approximated to be divided by two at each level. Indeed, a two-dimensional cluster of cardinality $n \times n$ is approximately divided into two clusters of cardinality $\frac{n(n-1)}{2}$ and a separator of cardinality n. A three-dimensional cluster of cardinality n^3 leads to two clusters of cardinality $\frac{n^2(n-1)}{2}$ and a separator of cardinality n^2. Keeping only the highest term leads to the aforementioned approximation. The descendants of τ_1 (or τ_2) have a cardinality approximately equal to $\frac{|\tau_1|}{2^k}$ after k divisions (except for the separators and their descendants):

$$\forall \sigma_1 \in \text{SparseDesc}_k(\tau_1), |\sigma_1| \sim \frac{|\tau_1|}{2^k}. \tag{2.3}$$

The mean cardinality of the descendants of the separator τ_S should be equal to that term. Thus, by choosing

$$k = log_2\left(\frac{|\tau_1|}{|\tau_S|}\right) \tag{2.4}$$

the number of levels to skip before dividing the separator leads to its cardinality being equivalent to the mean cardinality of the subdomain clusters of the same level,

$$\forall \sigma_S \in \text{Desc}_k(\tau_S), \forall \sigma_1 \in \text{SparseDesc}_k(\tau_1), |\sigma_S| \sim |\sigma_1|. \tag{2.5}$$

The next divisions are then also equivalent if bisection is used on the descendants of separators: both nested dissection and bisection leads to a division by two of the cardinality of clusters at each level using Eq. (2.3). Note that in both two-dimensional and three-dimensional contexts, we obtain approximately $k = log_2\left(\frac{n-1}{2}\right)$.

We have studied this equivalence for only one separator and the rest of the tree, excluding other separators. Let us now produce an equivalence between separators. This is the **second step** of our reasoning. The application of formula (2.4) to skip the division of other separators divisions induces this equivalence. Indeed, if a second separator τ_R is also divided using this technique, then the mean cardinality of its descendants is equivalent to the mean cardinality of subdomain clusters of the same level, i.e.,

$$\forall \sigma_R \in \text{Desc}_k(\tau_R), \forall \sigma_1 \in \text{SparseDesc}_k(\tau_1), |\sigma_R| \sim |\sigma_1|. \tag{2.6}$$

From (2.5) and (2.6), by transitive relation between the equivalence $\sim$ of the descendants of τ_S and of τ_R with the rest of the subdomain clusters of the same level, Problem 1 has been solved.

This policy leads to a more balanced cluster tree: the leaves of separator subtrees (with approximately the same cardinality as the other leaves by definition) are also located at the same depth as the other leaves. An $\mathcal{H}$-Matrix constructed with this technique is illustrated in Fig. 2.12. We must however point out that even if this technique is used on all separators, skipping levels when possible, another problem remains. If a separator τ is coincidentally a leaf (or too low in the cluster tree) and its cardinality is too small compared to the other leaves of the global cluster tree, meaning for example that the other branches are shorter than that of τ, it will not benefit from this technique. Skipping levels of division will not be useful as other branches will not be further divided. An example of remaining tall & skinny blocks is shown in Fig. 2.12 in orange.

Another issue is the inexactitude of the computations. We rely on approximations made before the actual divisions of the subsequent clustering of lower levels. If Eq. (2.2) or Eq. (2.3) are not satisfied, then the solution does not answer Problem 1.

Algorithm 27: Construction of a block cluster tree, preventing tall & skinny blocks with an abstract size-preserving approach.

```
Function CreateBlockClusterTree(σ × τ)
1   if IsLeaf(σ) ∧ IsLeaf(τ) ∨ Admissible(σ, τ) then
2       Children(σ × τ) ← ∅                                   ▷ Cluster Leaf
3   else
4       if IsLarger(σ, τ, R) then
5           for σ′ ∈ Children(σ) do
6               Children(σ × τ) ← Children(σ × τ) ∪ (σ′, τ)
7               CreateBlockClusterTree(σ′, τ)                 ▷ τ is not divided
8       if IsLarger(τ, σ, R) then
9           for τ′ ∈ Children(τ) do
10              Children(σ × τ) ← Children(σ × τ) ∪ (σ, τ′)
11              CreateBlockClusterTree(σ, τ′)                 ▷ σ is not divided
12      else
13          for (σ′, τ′) | σ′ ∈ Children(σ), τ′ ∈ Children(τ) do
14              Children(σ × τ) ← Children(σ × τ) ∪ (σ′, τ′)
15              CreateBlockClusterTree(σ′, τ′)                ▷ Division of σ and τ
```

These issues led us to another method to tackle this problem: a modified algorithm for the creation of block cluster trees, detailed in Algorithm 27, relying on the division of the largest dimension of the current block. With this modified algorithm, clusters do not need to have equivalent cardinalities or geometric diameters among the same

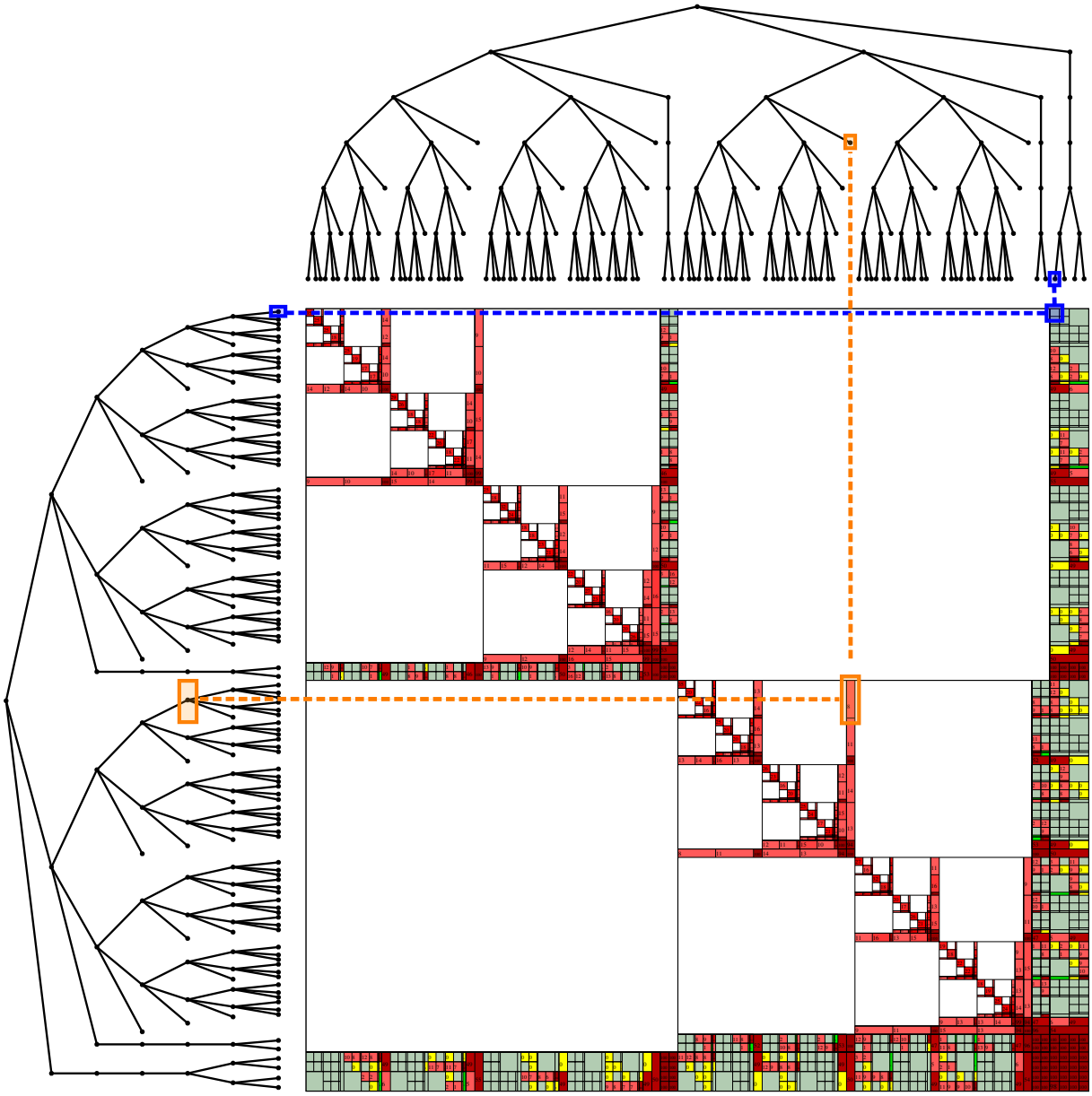


Figure 2.12: Tall & skinny blocks are avoided (example in blue) by skipping levels in the cluster tree. We can still see tall & skinny blocks (example in orange) when separators are leaves and are therefore not subdivided. See legend in Fig. 2.4.

level of clustering. The function `IsLarger`(σ, τ, R) used in the algorithm is either an algebraic comparison of the cardinality of the clusters or a geometric comparison of their diameter. The algebraic version, comparing the cardinalities of σ and τ, is defined in Algorithm 28. To preserve a geometric balance between the cluster diameters, the function

Algorithm 28: Algebraic comparison of the cardinality of σ and τ using a ratio R.

Function `IsAlgebraicLarger`(σ, τ, R)
1 **return** $|\sigma| > R \cdot |\tau|$

`IsGeometricLarger`(σ, τ, R) is defined in Algorithm 29.

Algorithm 29: Geometric comparison of the diameters of σ and τ using a ratio R.

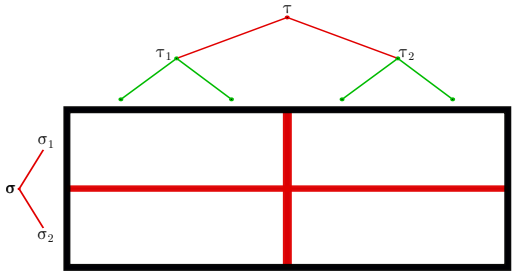
Function `IsGeometricLarger`(σ, τ, R)
1 **return** `Diameter`$(\sigma) > R \cdot$ `Diameter`(τ)

Using Algorithm 27, when a block $M_{\sigma\times\tau}$ is created, the dimensions of τ and σ are used to determine if the algorithm should recurse on their respective children to create the children of $M_{\sigma\times\tau}$. If τ is too large compared to σ by a chosen ratio R, we create a block using σ and the children of τ. This is depicted in Fig. 2.13. As depicted in Fig. 2.13b, the algorithm may either use the comparison `IsLarger`(τ, σ) on all clusters matrix blocks $M_{\sigma\times\tau}$ or it can be performed only if one of the two clusters is a leaf. The latter has been implemented due to its easier implementation. An example of a matrix constructed using this technique relying on algebraic comparison is shown in Fig. 2.14. One can see the shape of the blocks are more square than blocks resulting from previous algorithms. For this example, and in the rest of this document, the ratio has been set to

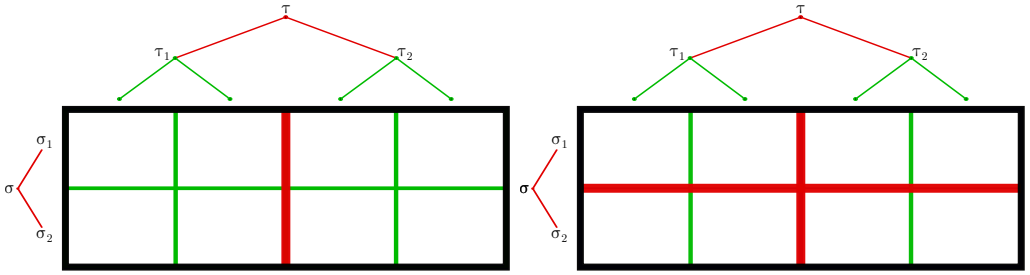
$$R = \sqrt{2}. \tag{2.7}$$

Matrix blocks tend towards a shape closer to a square for a ratio close to $\sqrt{2}$. Choosing a ratio between σ and τ lower than $\sqrt{2}$ would be meaningless, as the children of a cluster τ are usually around two times smaller than τ and the ratio between the children blocks and σ would then be lower than $\frac{\sqrt{2}}{2} = \frac{1}{\sqrt{2}}$, meaning the reversed ratio (τ compared to σ instead of σ compared to τ) would only be greater.

Table 2.1 summarizes the methods introduced in this discussion (of which the primary goal is to minimize the distortion of tall & skinny blocks) based on either a geometric or an algebraic balance, and a modification of either the cluster tree or the block cluster tree. We refer the interested reader to [114, § 5.5.3] for a discussion on alternative block cluster tree construction leading to a binary block cluster tree instead of a quadtree.



(a) Example of a rectangular hierarchical block such as constructed initially. The recursion stops after the first (red) step because σ_1 and σ_2 are leaves.



(b) The size of τ is too large so we do not recurse on the children of σ and create blocks using σ and the children of τ. This is done until the size of the rows and the columns are sufficiently close. The left example shows a division using this difference in size at the creation of $M_{\sigma\times\tau}$. The right example depicts a division using this difference on leaves only (see Fig. 2.13a). The latter is used in practice due to implementation consideration.

Figure 2.13: Tall & skinny example of a matrix block $M_{\sigma\times\tau}$. The first level of hierarchy is colored red while the second level is colored green.

2.3.3 Example of a Sparse Format on the $\mathcal{H}$-Matrix Leaves

To further exploit the sparsity pattern of a matrix, another (sparser) format could be used on the leaves of the $\mathcal{H}$-Matrix, along with the preexisting leaf formats, i.e., $\mathcal{R}k$-Matrices and Full-Matrices. We present here a format discarding the empty rows and columns of a submatrix and grouping the remaining non-empty rows and columns together via a permutation into a single, smaller storage. The format may be applied on either $\mathcal{R}k$-Matrices or Full-Matrices. This format matches more precisely the pattern of non-zeros of the matrix but the local permutation means an extra step of verification of the ordering at each performed operation. Fig. 2.15 shows an example of a $m \times n$ matrix block with nnz non-zeros. For such a block, we consider:

- the Admissibility format, leading to the Full-Matrix in Fig. 2.16 and the $\mathcal{R}k$-Matrix in Fig. 2.17;
- the Fit format, leading to the restricted Full-Matrix in Fig. 2.18 and the $\mathcal{R}k$-Matrix in Fig. 2.19;

However this also implies a reordering local to each submatrix. We will see in § 2.4.3.4 another way to benefit from this sparsity without requiring such a constraint.

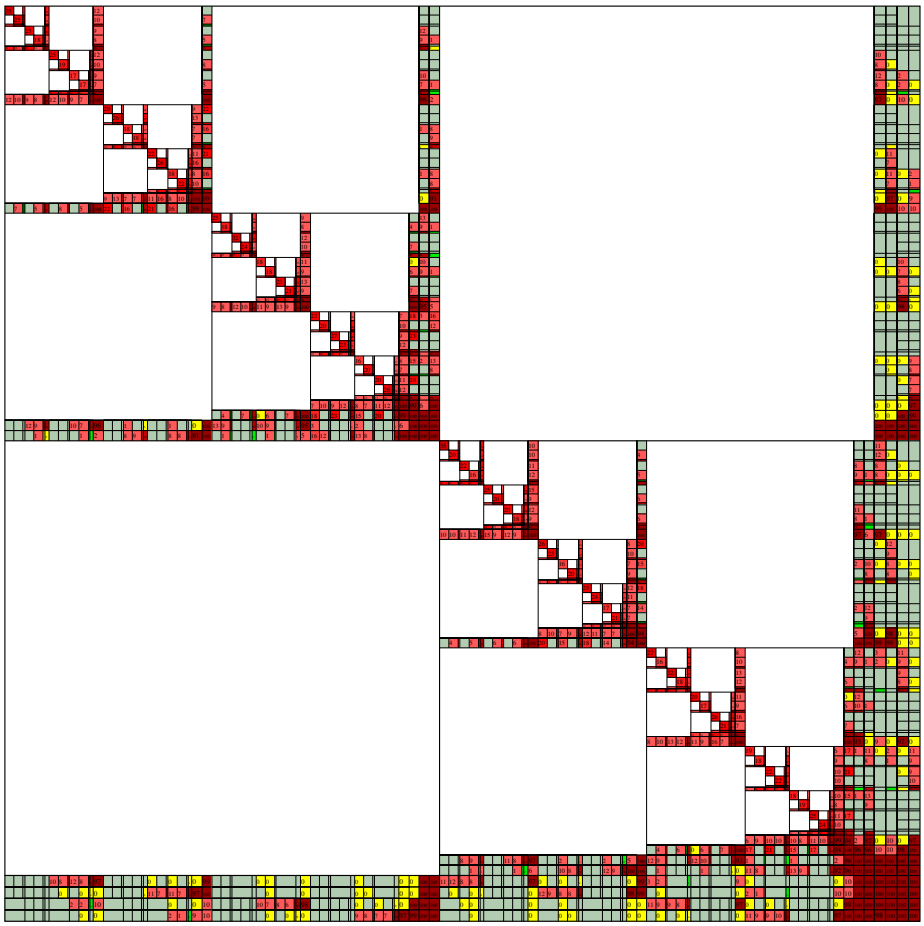


Figure 2.14: Tall & skinny blocks are avoided by using an algebraic balance when constructing a block cluster tree (Algorithm 27). Here, the ratio R has been set to $\sqrt{2}$. The tall & skinny blocks in orange in Fig. 2.12 are not present anymore. See legend in Fig. 2.4.

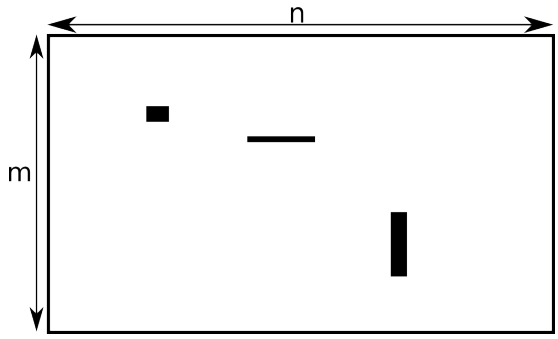


Figure 2.15: Non-zero pattern (black rectangles) of a $m \times n$ matrix block. There are nnz non-zeros.

	Cluster tree	Block cluster tree	Balance
[114, §9.2.4], Fig. 2.7	Modified	Canonical	Geometric
Fig. 2.11 and 2.12	Modified	Canonical	Algebraic
algorithms 27 and 29	Canonical	Modified	Geometric
algorithms 27 and 28 and Fig. 2.14	Canonical	Modified	Algebraic

Table 2.1: Methods to prevent tall & skinny blocks, preserving a geometric or an algebraic balance, and based on the modification of the cluster tree or the block cluster tree.

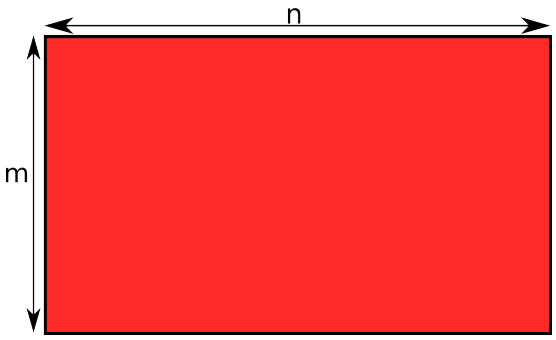


Figure 2.16: Admissibility Full format.

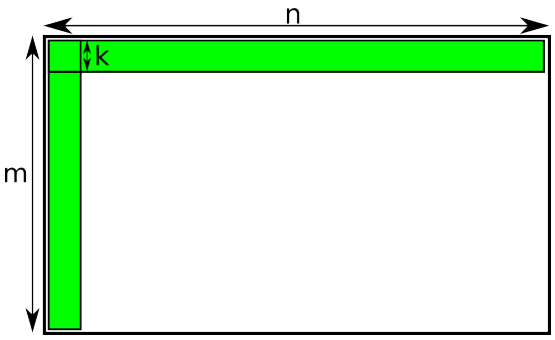


Figure 2.17: Admissibility $\mathcal{R}k$ format.

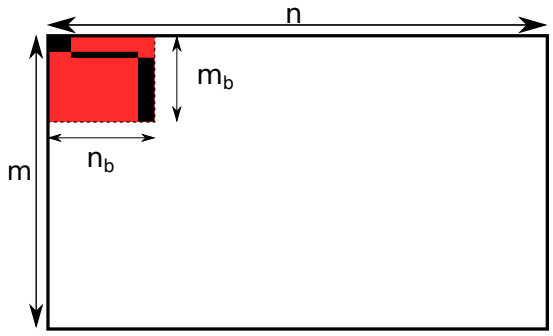


Figure 2.18: Fit Full format: a permutation that leads to a better locality of non-zeros.

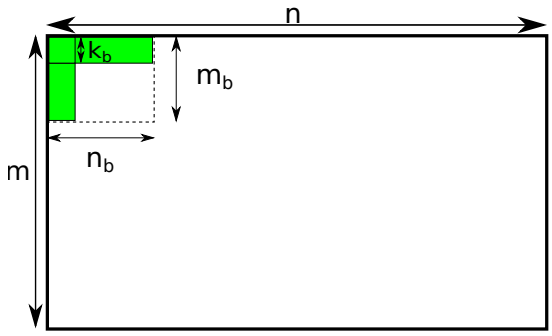


Figure 2.19: Fit $\mathcal{R}k$ format: $\mathcal{R}k$ compression on a Fit Full block.

2.3.4 Towards a Sparse Admissibility Condition

Another approach to further reduce the memory consumption of $\mathcal{H}$-Matrices is to elaborate a new admissibility condition to compress more submatrices. Indeed, Fig. 2.20 shows that a lot of blocks are mistakingly stored as Full-Matrices while they would benefit from compression. The original admissibility condition described in Eq. (1.8) for dense linear systems seems to fail categorizing some blocks as admissible even though they should have been. In order to avoid confusion, we refer to this original admissibility condition as *strong*. We may now wonder what a sparse-specific admissibility condition should rely on. In fact, if we take a closer look at Fig. 2.20, we may observe that almost every block that constitutes an interaction between a separator and a local subdomain is colored blue, i.e.,

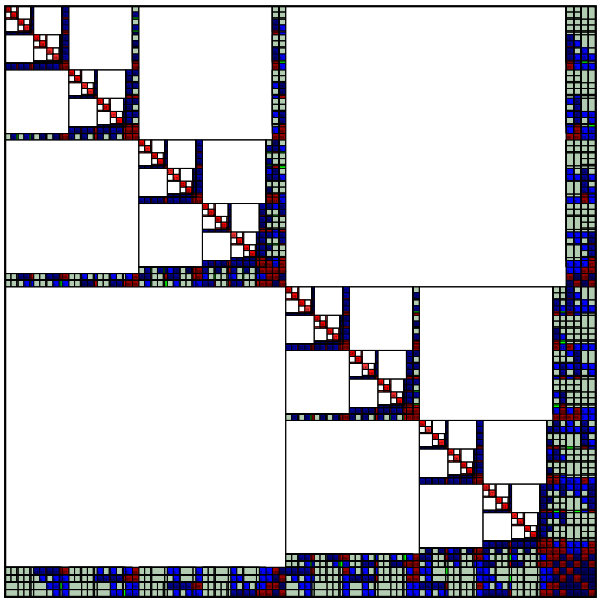


Figure 2.20: A lot of dense blocks would benefit from compression. Blue blocks represent such blocks while, as for the other figures, red blocks are Full-Matrices and green blocks are $\mathcal{R}k$-Matrices. The parameter ϵ is here equal to 10^{-10}. See legend in Fig. 2.4.

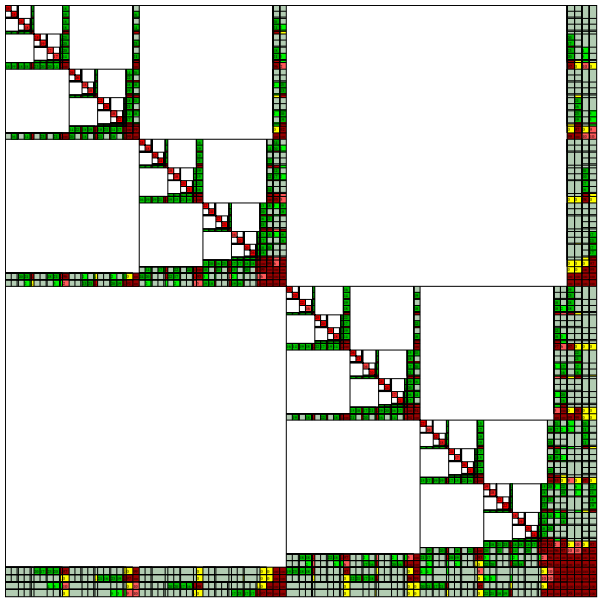


Figure 2.21: Non-separator compression for $\epsilon = 10^{-10}$. Some separator/separator blocks are still stored as Full-Matrices while being empty. See legend in Fig. 2.4.

should have been compressed. This is due to the fact that separators receive contributions mainly from other separators they are connected to. Other off-diagonal blocks are very few and sparse. We may choose a smaller threshold for the size of these blocks, as discussed in the previous section, or a sparse format (for example the Fit format) to reach the fineness of these non-zeros. But we focus here on an alternative solution: using compression for blocks that represent an interaction with one non-separator local subdomain, in addition to the blocks decided admissible by the strong criterion. We refer to this admissibility condition as *non-separator*. Instead of creating a lot of small blocks to handle, as a sparse solver would usually resort to, we keep larger off-diagonal blocks but compressed them. This is not as optimal as a sparse format in terms of memory consumption but has the advantage of running faster due to the operations involving larger blocks. An example of an $\mathcal{H}$-Matrix using this criterion may be seen in Fig. 2.21.

To measure the effectiveness of each criterion, we also implemented an *oracle* admissibility condition. After having computed the LU factorization, we determine the most effective storage format (being either a $\mathcal{R}k$-Matrix or a Full-Matrix) for each block of the $\mathcal{H}$-Matrix. We also compute a coarsening method similar to [199, III.C]. The coarsening method studies for each $\mathcal{H}$-Matrix node τ of which all children are $\mathcal{R}k$-Matrices, and compares the storage of the node τ as a $\mathcal{R}k$-Matrix to that of the sum of its children.

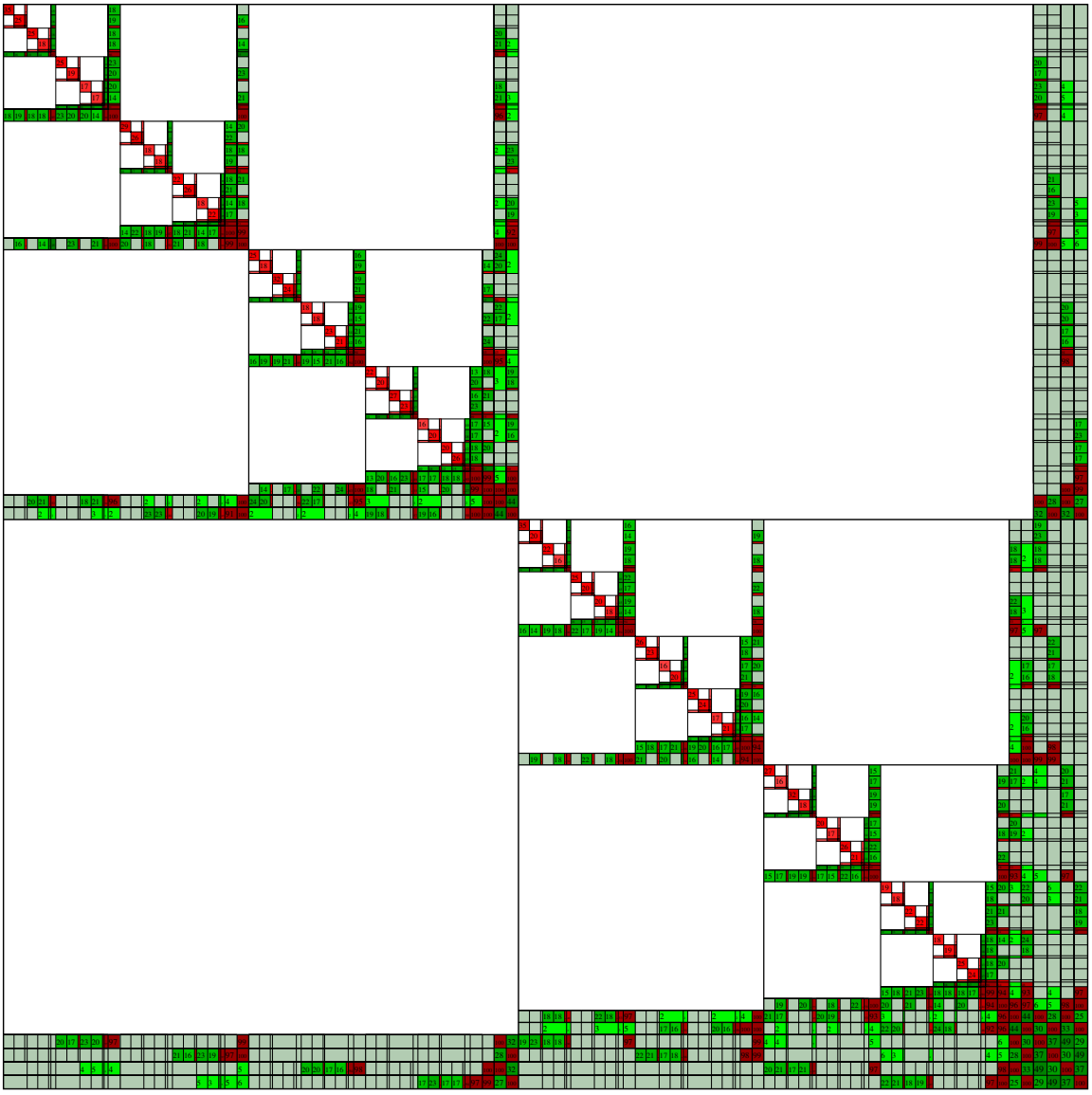
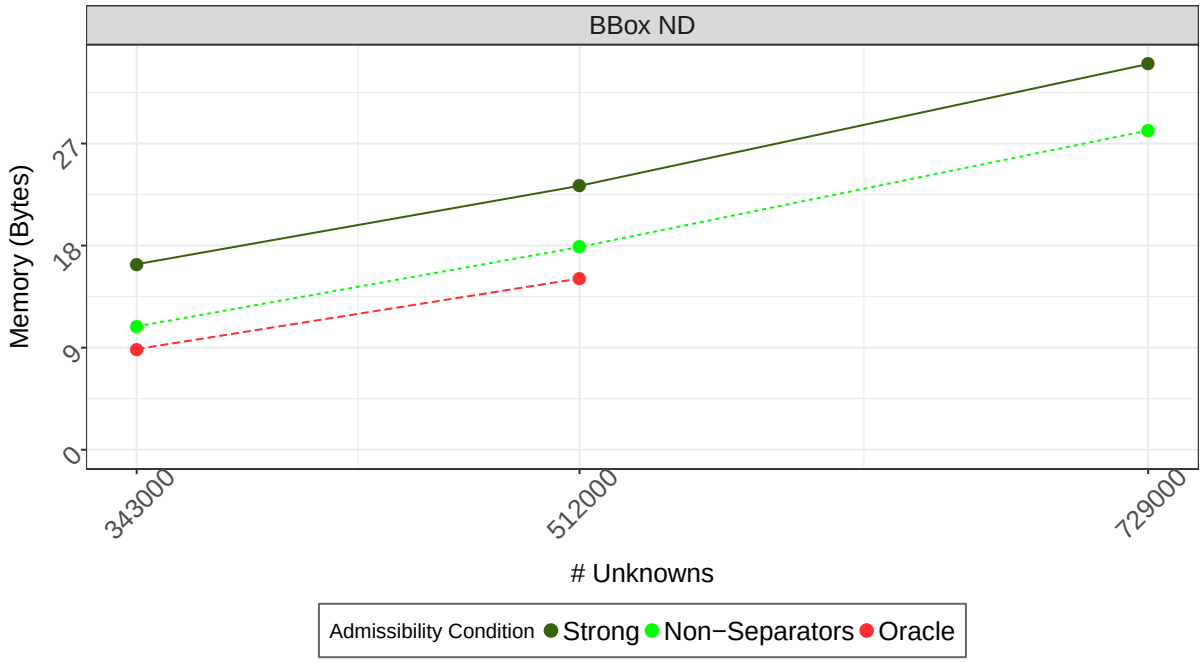
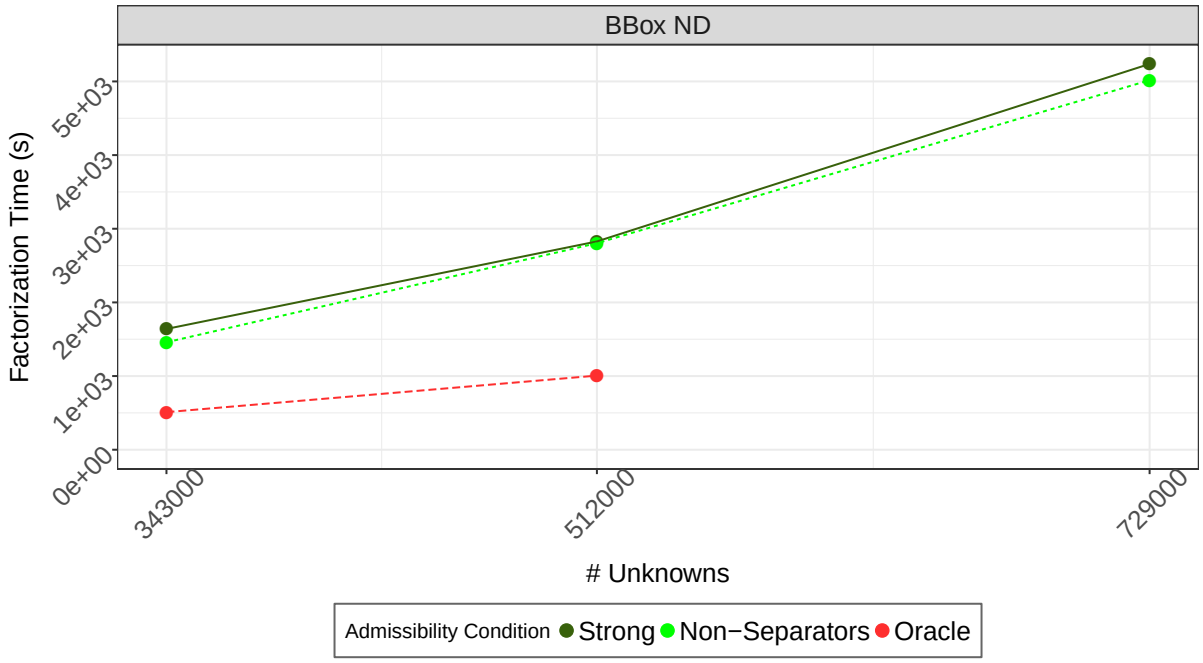


Figure 2.22: Optimal compression for $\epsilon = 10^{-10}$ leading to the minimum storage that can be achieved using this matrix partition (using an oracle). See legend in Fig. 2.4.



(a) Memory consumption of the factorized matrix.



(b) Time required for a numerical factorization.

Figure 2.23: Impact of non-separator admissibility condition on numerical factorization memory and time consumption, with the optimal scenario as a reference. Laplacian cube test case (Fig. 1.17). Sequential run on `miriel` (see § 3.2.1).

If the storage of τ as a $\mathcal{R}k$-Matrix is smaller than the sum of its children, then τ is kept as admissible and its children are discarded. Otherwise, its children are kept admissible. We can then recompute the factorization to effectively measure its time and memory consumption. We must emphasize that this oracle computes two LU factorizations and is therefore most ineffective in practice and serves only as a reference. An example of this optimized storage (in terms of compression) can be seen in Fig. 2.22. The effectiveness of the non-separator admissibility condition is studied in Fig. 2.23 compared to the original strong criterion and the oracle. It successfully reduces both the time and the memory consumption of the solver, while not reaching the optimal scenario. The oracle performance suggests that the admissibility condition may be further improved in the case of the problems investigated in this thesis. This should therefore be studied in future works.

In the following section, we do not rely on the compression of sparse or zero off-diagonal submatrices, but instead rely on the notion of symbolic factorization to further exploit the non-zero structure inherent to a sparse matrix.

2.4 $\mathcal{H}$-Matrices & Symbolic Factorization

Sparse solvers are able to precisely locate off-diagonal non-zeros in the factors due to filled-in entries through the means of a symbolic factorization (as defined in § 1.3.1.2), and use this information to compute an efficient block and sparse structure (a method not used in nested dissection-based $\mathcal{H}$-Matrices discussed in [102, 107, 138]). These sparse methods do not rely on a global hierarchy (though they may rely on a local hierarchy, as discussed in § 2.2), but have the advantage of using symbolic factorization to create matrix blocks matching the pattern of non-zeros. Therefore we may wonder how a symbolic factorization may be computed in a hierarchical context and what fundamental differences would there be between: (1) a sparse solver using $\mathcal{H}$-Matrices (or BLR) on local submatrices, and (2) a hierarchical solver using symbolic factorization to create a hierarchy aware of the non-zero and fill-in pattern.

2.4.1 Symbolic Factorization in a Hierarchical Context

Sparse solvers relying on a symbolic factorization usually compute a block structure deduced from the non-zero pattern of the original matrix. The introduction of low-rank compression methods in sparse solvers was performed through a compression method applied locally on submatrices dense and large enough for the compression to be effective. For example, we have presented in § 2.2 a sparse supernodal solver using a BLR approach for the compression of off-diagonal blocks larger than a minimal size [165] and multifrontal methods relying on a BLR compression [157] or on HSS-Matrices [95] applied on frontal matrices or Schur complements. These sparse low-rank methods already show effective gains in terms of time and/or memory consumption compared to their respective full-rank

versions. We will now discuss the differences between these methods and a hierarchical method using symbolic factorization, to give the reader a perspective of the impact of each technique.

In many respects, some of the sparse techniques introduced in § 1.3 and hierarchical techniques introduced in § 1.2.3 correspond to each other. For example, cluster trees and elimination trees are very close in what they represent, as already discussed in the introduction of this chapter. Moreover, while an $\mathcal{H}$-Matrix arises from two cluster trees, as depicted in Fig. 2.24, we can also see the storage of sparse supernodal solvers as a construction based on one column partitioning (defined by the elimination tree) which is the equivalent of the column cluster tree for $\mathcal{H}$-Matrices. Then, each supernode

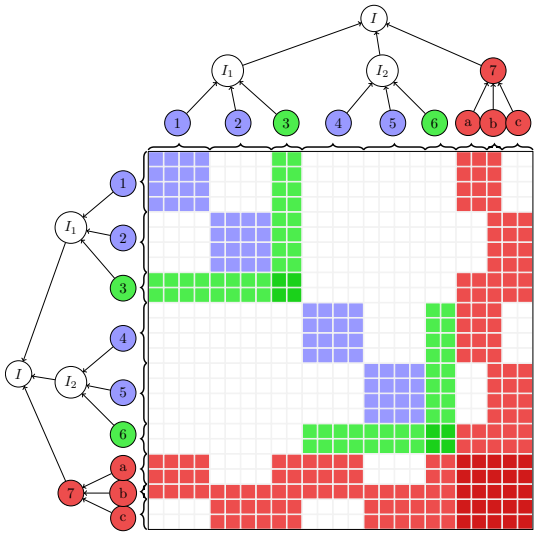
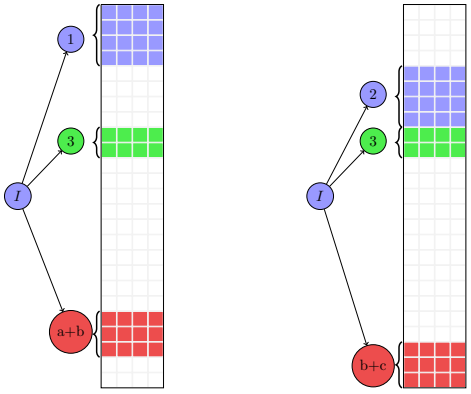


Figure 2.24: Nested dissection row and column cluster trees resulting in a sparse matrix.



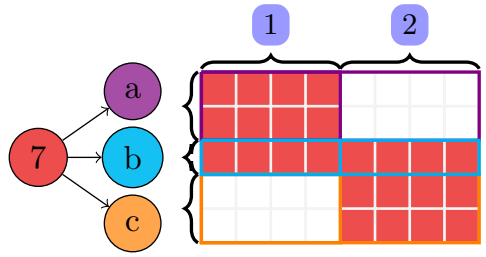
(a) Supernode 1. (b) Supernode 2.

Figure 2.25: List of blocks for two supernodes, arranged as cluster trees.

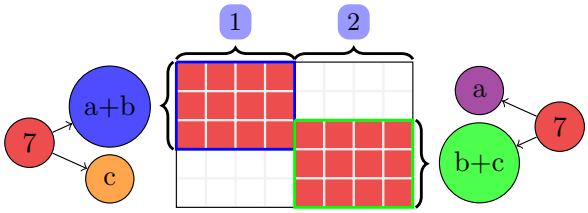
(equivalent of a leaf of the column cluster tree) is partitioned, as illustrated in Fig. 2.25, using a structure which we introduced in § 1.3.1.2.2 as a list of intervals of rows. A cluster is also represented by an interval of indices (see § 1.2.3.2.1). Consequently, instead of using a global row cluster tree as the $\mathcal{H}$-Matrices do, a supernodal method may be seen as a construction with one single-level cluster tree for each supernode. This enhances how fine a solver is by locating more precisely blocks of zeros, independently of the block columns they are located in. But this is also a source of "incompatible" operations, in that blocks will contribute to blocks with different ranges of rows, as we will discuss further later (Fig. 2.27).

On the other hand, if hierarchical methods tend towards a symbolic factorization by using a global cluster tree grouping nodes by their pattern of interactions, even in a separator, each separator would be divided into a list of smaller clusters (which can be related to supernodes). In the example of Fig. 2.24, there are only three clusters, a, b, and c in the largest separator (numbered 7 here and colored red). Large matrices however lead

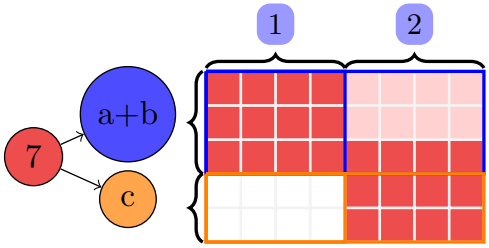
to a separation into many clusters, which can rapidly deteriorate the performance. This gives us an incentive to resort to relaxation, such as briefly introduced for supernodes in § 1.3.1.1.3. We focus in Fig. 2.26 on the interactions of the first two supernodes with separator 7, i.e., the bottom left of the matrix. The supernode 1 contains unknowns 1 to 4, supernode 2 contains unknowns 5 to 8 and separator 7 the unknowns 21 to 25, as shown in Fig. 1.31c (corresponding to Fig. 2.24). From the partition of separator 7 based on its interactions with the first two supernodes into a, b and c, we would have six submatrices as depicted in Fig. 2.26a. The equivalent of the sparse supernodal method would lead to



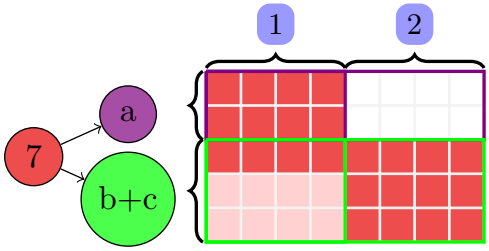
(a) Partition into three sets.



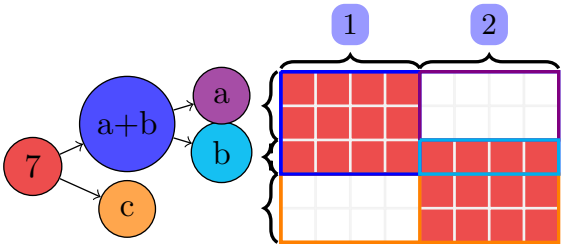
(b) Partition into different sets per supernode.



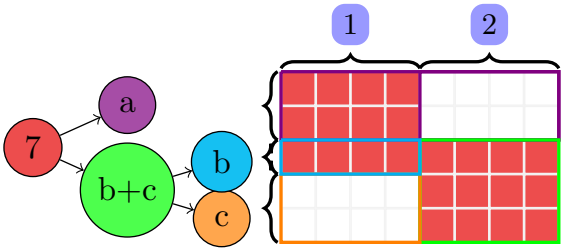
(c) a and b are amalgamated together.



(d) b and c are amalgamated together.



(e) $a + b$ as a parent of a and b.

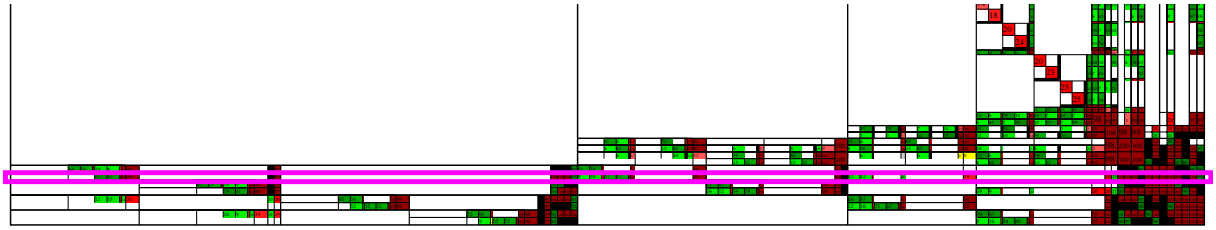


(f) $b + c$ as a parent of b and c.

Figure 2.26: Possible hierarchical partitions and relaxations of clusters for separator 7, based on its interactions with supernode 1 and 2. Light-red colored blocks represent zeros storage.

a different cluster tree on each supernode, and here, the separator 7 would be divided into two sets $a+b$ and c for supernode 1 while it would be divided into a and $b+c$ for supernode 2, as illustrated in Fig. 2.26b. But if we intend to keep a single global cluster tree, we may use an amalgamation of a and b together as $a+b$ (Fig. 2.26c). However, this would mean that the block resulting from the interactions of $a + b$ and the supernode 2, which could be denoted by $M_{a+b,2}$, would store the interactions of the supernode 2 with unknowns 21

(a) In a sparse matrix, blocks are not usually aligned on a given range of rows.



(b) Compatibility of operations in an $\mathcal{H}$-Matrix: all rows and all columns are aligned on the grid defined by the product of the row cluster tree and the column cluster tree, i.e., the block cluster tree. The row highlighted in pink for example consists exclusively in blocks with the same range of rows.

Figure 2.27: Compatibility of operations & alignment of blocks.

and 22, while they are, in fact, null. This is represented by the light-red-colored blocks. A similar version would amalgamate blocks together beginning by the last ordered blocks, here for example b and c, as shown in Fig. 2.26d. Finally, a better version could be achieved by inserting $a + b$ as an intermediate step between the partition of 7 into a and b. This is illustrated in Fig. 2.26e. However, this version would stop the recursion at the block $M_{a+b,1}$ (in the first supernode) while $M_{a+b,2}$ would be divided into $M_{a,2}$ and $M_{b,2}$ (in the second supernode), which may not correspond to the initial implementation of a hierarchical solver. Indeed, $M_{a+b,1}$ would only be considered a leaf if it is admissible. The same partition may be computed for its equivalent $b + c$ (Fig. 2.26f). This would lead to operations between misaligned blocks to be performed, in a way similar to the partition by supernodes performed in a sparse solver. One could argue that subdividing $M_{a+b,1}$ into two smaller matrices $M_{a,1}$ and $M_{b,1}$ would not have a significant impact on performance and would avoid this problem of compatibility.

We have therefore several choices of possible amalgamations which would lead to larger blocks and reduce the number of blocks to store. This impacts the performance due to the fact that large operations are more efficient but also due to compression, as compression is usually more efficient for larger blocks. Contrary to the partition of a sparse solver, operations are most often compatible in a hierarchical solver (see Fig. 2.27), in that rows and columns of the submatrices involved usually match (if one has not been judged admissible at a higher level in the hierarchy). Indeed, we can for example see in Fig. 2.27a that the blocks involved in the operations of the highlighted block row tend to have different row sizes. In Fig. 2.27b, the highlighted blocks all share the same set of rows. To conclude, a globally hierarchical-based method may have an advantage in terms

of compression while a globally sparse-based method may have an advantage in terms of storage.

In the following paragraphs, we give an overview of methods to compute a symbolic factorization in a hierarchical context, which may be put in perspective with § 2.1.4. In § 2.4.2, we further investigate the idea of detecting fill-in at each level of recursion of the block cluster tree by applying a hierarchical algorithm to avoid storage of zeros at the highest possible level in the tree. These symbolic factorizations rely on the interactions between clusters from a quotient graph (introduced in § 1.3.1.1.3) and lead therefore to a block symbolic factorization. We rely on the structure $\texttt{Reach}(K)$, which may be decomposed into the right-looking interactions $\texttt{Reach}^+(K)$ and left-looking interactions $\texttt{Reach}^-(K)$ of the cluster/supernode K. It may be either computed for a quotient graph G/P under the form $\texttt{Reach}^+_{G/P}(K)$ or $\texttt{Reach}^+_{(G/P)\to G}(K)$, leading to algorithms 15 and 16, p. 63 and p. 65, for the right-looking algorithms, respectively. It may also be computed under the left-looking form $\texttt{Reach}^-_{G/P}(K)$, leading to Algorithm 19. In fact, we will use $\texttt{Reach}(K)$ as a generic notation for any of these in the following algorithms, meaning it may be replaced by any of the right-looking/left-looking, quotient/scalar forms (if not specified). The hierarchical algorithms described here are based on the aforementioned algorithms (for example algorithms 15 and 16 for the right-looking variants).

We then discuss in § 2.4.3 the notion of separator clustering based on left-looking interactions, i.e., the structure $\texttt{Reach}^-(x)$ that represents the set of elements that *reach* x, introduced in § 1.3.1.2.4. $\texttt{Reach}^-(x)$ describe the interactions of an entity x (unknown or cluster) with the rest of the elements constituting the considered adjacency graph, with the specificity of being ordered before x (hence its "left-looking" attribute). Here, we will focus on interactions with quotient graphs, i.e., on the computation of the structure $\texttt{Reach}^-_{G/P}(x)$ for the quotient graph (G/P). While $\texttt{Reach}^+_{G/P}(K)$, referred to as $BStruct(L_{*K})$ in [119], corresponds to the list of supernodes that supernode K can reach (or update), $\texttt{Reach}^-_{G/P}(K)$ is the reverse list, i.e., the list of supernodes by which K is reached (or updated), also referred to as $BStruct(L_{K*})$ in [119]. It is also related to the structure $\texttt{Key}(K)$ introduced in [125]. Using this structure, we can then divide a separator S following their interactions with other clusters to create a new partition of smaller supernodes inside the separator. In other words, the node from the quotient graph corresponding to S is split between smaller clusters with the same outgoing edges in the quotient graph. This algorithm has first been applied in a "flat" format (presented in § 2.4.3.2), meaning each separator is divided into one list of smaller supernodes, each corresponding to a list of indices that have the same interactions with other leaf clusters. Therefore, by construction, the hierarchy inside separators is lost. This is why a second variant has been developed to apply the symbolic factorization in a hierarchical way (§ 2.4.3.3).

2.4.2 Hierarchical Symbolic Factorization (HSF)

The techniques discussed in this section may be put in perspective with the sparsity pattern described in § 2.1.4. Fill-in is computed for each matrix block in order to predict before the numerical factorization if the block should be stored, to avoid on-the-fly management of fill-in. We heavily rely on the notion of quotient graph introduced in § 1.3.1.1.3 and the block symbolic factorization introduced in § 1.3.1.2.2. The symbolic factorization is not limited to the leaf clusters here as we examine the propagation of fill-in also higher in the cluster tree hierarchy. This enables the combination of symbolic factorization and compression at higher levels in the hierarchy. Compression is not limited to each individual block column such as in a supernodal solver.

To detect non-zero blocks and fill-in, the function defined in Eq. (1.12), based on left-looking information, may be used. However, this function induces the not-so efficient left-looking Algorithm 18 if all the fill-in is to be taken into account. Therefore, the efficient algorithms introduced in § 1.3.1.2 are here re-adapted for a hierarchical construction such as the $\mathcal{H}$-Matrices. In terms of graphs, this hierarchical symbolic factorization may be described as the computation of the elimination of a quotient graph at each level of the hierarchy of the cluster tree.

2.4.2.1 Definition of Cluster Symbolic Information

For the algorithm to be able to determine if a submatrix $M_{\sigma\times\tau}$ is null (and will be null in the factors) or if it must be stored, the symbolic information we will compute and associated with each cluster τ (or σ) has to be sufficient. For efficiency consideration, it should also be the minimal information necessary for this task. In order to store this information, we have identified two suitable data structures. Either the symbolic information associated with τ contains exactly the list of clusters with which it interacts or the list of scalar vertices with which it interacts. Therefore we differentiate here two methods, the **Cluster-Cluster Hierarchical Symbolic Factorization (CC-HSF)** and the **Cluster-Vertex Hierarchical Symbolic Factorization (CV-HSF)**, depending on the information stored on each cluster. We have named the two methods following the computed row and column partition used for the symbolic factorization. Let us remind the reader that $\mathcal{H}$-Matrices are based on a row cluster tree and a column cluster tree. We rely on the column cluster tree for the column partition (further discussed in § 2.4.2.1.3). The row partition is then chosen as either the row cluster tree, leading to the Cluster-Cluster (CC) algorithm, or a vertex-wise information from the adjacency graph, leading to the Cluster-Vertex (CV) algorithm. In the CC-HSF approach, we may construct a structure leading to Fig. 2.26a, 2.26c and 2.26e, in which we may avoid the storage of zeros, due to the fact that we have the symbolic information of which cluster interacts with which cluster. This method is to be compared to the method discussed in § 1.3.1.2.2 under the name of *Block Symbolic Factorization*, adapted for a hierarchical framework. In our context, the (flat) Block Symbolic Factorization method could be called Cluster-Cluster Flat Symbolic Factorization (CC-FSF) due to the computed block (Cluster-Cluster) information. In the CV-HSF approach, we will rely on a subdivision

discussed in § 2.4.3.4 to obtain a partition closer to a supernodal approach, such as illustrated by Fig. 2.26b, in an effort to reach the fine granularity of a sparse solver. With this latter method, a local clustering is used on each column cluster. It may be compared to the method discussed in § 1.3.1.2.3 under the name of *Block Column Symbolic Factorization*, also adapted for a hierarchical framework. In our context, this (flat) Block Column Symbolic Factorization might be named Cluster-Vertex Flat Symbolic Factorization (CV-FSF) due to the computed block column (Column-Vertex) information.

2.4.2.1.1 Cluster-Cluster Symbolic Information (CC-HSF)

The goal of the CC-HSF method, as mentioned above, is to extend the Block Symbolic Factorization (or CC-FSF) algorithm introduced in § 1.3.1.2.2 to a hierarchical framework. It computes symbolic information between two clusters of the row and column cluster trees that will later be used in the construction of the corresponding $\mathcal{H}$-Matrix to avoid the storage of zeros.

Using the notion of adjacency introduced in § 1.3.1.1.1 extended to supernodes in § 1.3.1.2.2, we can define the symbolic information of a cluster using its adjacency with clusters from the same depth in the overall cluster tree. For a cluster τ, with the scalar graph $G = (V, E)$ and a partition P to be determined satisfying $\tau \in P$, this cluster adjacency is defined as:

$$\texttt{Adj}_{G/P}(\tau) = \{\sigma \mid (\exists\, i \in \tau) \wedge (\exists\, j \in \sigma) \wedge ((i,j) \in E) \wedge (\sigma \in P)\}. \tag{2.8}$$

In other words, the adjacency $\texttt{Adj}_{G/P}(\tau)$ of the cluster τ in the scalar graph G associated with the partition P (containing τ) is thus the set of clusters σ of which some unknowns are adjacent (in G) to unknowns contained in τ. For the purpose of computing a symbolic factorization, we only need to compute that information for one and only one partition for each τ. The union of all clusters inside the partition should match the set I of unknowns of the root of the cluster tree, i.e., all the unknowns of the problem:

$$\bigcup_{\sigma \in P} \sigma = I. \tag{2.9}$$

In the context of usual $\mathcal{H}$-Matrices, the block cluster tree is considered to be a level-conserving product of two cluster trees, i.e., satisfying Eq. (1.10). Each submatrix $M_{\sigma\times\tau}$ in such an $\mathcal{H}$-Matrix corresponds therefore to the clusters σ and τ at the same depth, i.e., $\texttt{Depth}(\sigma) = \texttt{Depth}(\tau)$. The partition P may then simply be defined as one level of the cluster tree, i.e.,

$$P = \{\sigma \mid \texttt{Depth}(\sigma) = \texttt{Depth}(\tau)\}. \tag{2.10}$$

The only exception to this rule in the context of this thesis is the construction presented in § 2.3.2. Therefore, we here consider clusters σ and τ to be at the same depth if they lead to a submatrix $M_{\sigma\times\tau}$, *except* if either one of the clusters is a leaf. In that case, it is possible for σ and τ to be at different depths. The simplest way to handle this type of

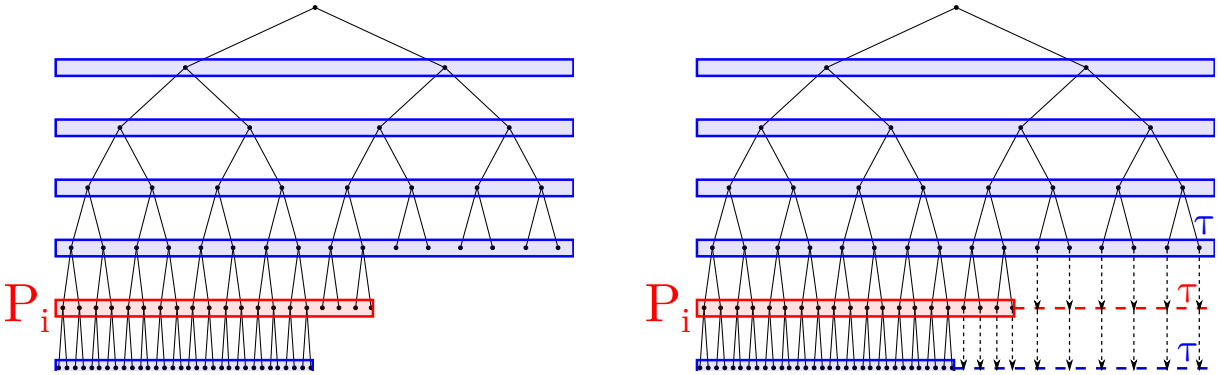


Figure 2.28: Unbalanced cluster tree (left) and reuse of leaves from shorter branches in deeper levels (right), represented by the dashed lines. Partitions at each level are highlighted in blue and an example of an extended partition P_i is colored red.

construction is to use the partition of leaf clusters if τ is a leaf. Also, if a branch of the cluster tree is shorter than the branch of τ, the leaf of that branch should be included in the partition, to ensure Eq. (2.9) is satisfied. In that case, the partition P may be redefined as either leaves if τ is a leaf, or the clusters σ at the same depth as τ or that are leaves if σ is higher in the tree than τ. Formally, we have:

$$\begin{aligned} P = \{\sigma \mid & (\texttt{IsLeaf}(\tau) \wedge \texttt{IsLeaf}(\sigma)) \\ & \vee (\neg\texttt{IsLeaf}(\tau) \wedge (\texttt{Depth}(\sigma) = \texttt{Depth}(\tau) \\ & \quad \vee (\texttt{Depth}(\sigma) < \texttt{Depth}(\tau) \wedge \texttt{IsLeaf}(\sigma))))\}. \end{aligned} \tag{2.11}$$

For a matter of simplicity, we will rely on the notation P or sometimes P_i if the considered depth i of the cluster tree has to be specified. Some clusters may be included in multiple partitions if they are leaves. This representation is depicted in Fig. 2.28. An example of a leaf τ shared by multiple partitions is shown in this figure. The left part of the cluster tree is deeper than the right, so the elements on the right are reused in the lower levels in which they do not exist. However, we must emphasize that the leaves only store the interactions with other leaves and not the upper partitions they may be included in, as formally expressed in Eq. (2.11).

To compute the fill-in generated by the factorization, we rely on the computation of the set $\texttt{Reach}_{G/P_i}(\tau)$. As mentioned earlier, this structure may be right or left-looking. For a matter of conciseness, when there is no ambiguity, we may omit the index i (and the G/P reference) and note $\texttt{Reach}(\tau)$. Computing this structure for all clusters in the cluster tree therefore corresponds to the computation of the elimination of a quotient graph for each partition P_i. We consider the necessary conditions (§ 1.3.1.1.3) for $(G^*/P) = (G/P)^*$ to be satisfied. Instead of a simple block column symbolic factorization, as classically done for non-hierarchical solvers [61, 93], we aim at applying a symbolic factorization (see § 1.3.1.2.2) at each level of the $\mathcal{H}$-Matrix, in order to detect blocks of zeros at the highest possible level in the hierarchy. As an example, we focus on the two layers of graphs depicted in Fig. 2.29. An example of the construction of the corresponding hierarchical

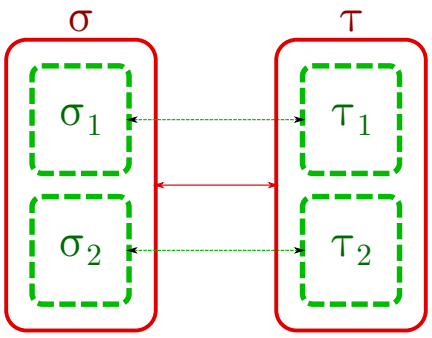


Figure 2.29: Two levels of hierarchy of an adjacency graph example associated with Fig. 2.30, leading to two quotient graphs.

structure is shown in Fig. 2.30. Here, we have two clusters σ and τ that interact with each other, forming the quotient graph of Fig. 2.30a. Their sub-clusters, σ_1 and σ_2 children of σ, as well as τ_1 and τ_2 children of τ, interact only partially with each other: σ_1 interacts with τ_1 and σ_2 interacts with τ_2. This may be represented by the second (lower) quotient graph shown in Fig. 2.30b. The corresponding two levels of hierarchy in the block cluster tree (Fig. 2.30c and 2.30d, respectively) show how these interactions correspond to non-zero blocks in the matrix.

It should be noted that only the clusters ordered after the cluster τ are necessary for the algorithm presented in § 1.3.1.2.2, due to its right-looking characteristic. Therefore, using Eq. (2.8), $\texttt{Reach}^{+}_{G/P}(\tau)$ is initialized as the set of clusters σ that are ordered after τ ($\sigma > \tau$) and that are adjacent to τ ($\sigma \in \texttt{Adj}_{G/P}(\tau)$):

$$\texttt{Reach}^{+}_{G/P}(\tau) \leftarrow \{\sigma \mid (\sigma > \tau) \wedge (\sigma \in \texttt{Adj}_{G/P}(\tau))\}. \tag{2.12}$$

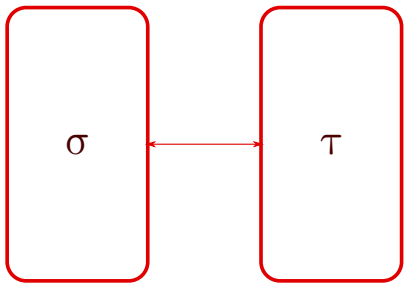
Conversely, if the algorithm were to be written in a left-looking fashion (see § 1.3.1.2.4), we would need to store the information from the clusters ordered before τ ($\sigma < \tau$), for example in a structure named $\texttt{Reach}^{-}_{G/P}(\tau)$:

$$\texttt{Reach}^{-}_{G/P}(\tau) \leftarrow \{\sigma \mid (\sigma < \tau) \wedge (\sigma \in \texttt{Adj}_{G/P}(\tau))\}. \tag{2.13}$$

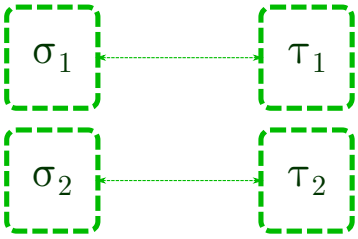
These right and left-looking structures are depicted in Fig. 2.31a and Fig. 2.31b, respectively. The algorithm for the creation of an $\mathcal{H}$-Matrix (originally, Algorithm 9, p. 37) becomes Algorithm 30 (in its right-looking form), with the simple addition of line 8, which checks whether σ' interacts with τ'. For a left-looking variant, the structure $\texttt{Reach}^{+}_{G/P}(\tau)$ may simply be replaced with $\texttt{Reach}^{-}_{G/P}(\tau)$. This additional check ensures to only store blocks that will have at least one non-zero in the factorized $\mathcal{H}$-Matrix.

2.4.2.1.2 Cluster-Vertex Symbolic Information (CV-HSF)

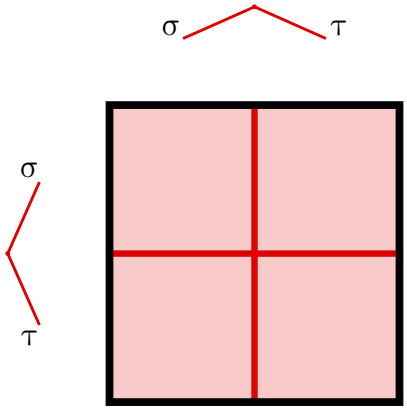
The proposed CC-HSF algorithm ensures to hierarchically store non-null blocks, i.e., blocks that contain at least one non-zero. However, the resulting hierarchical structure will



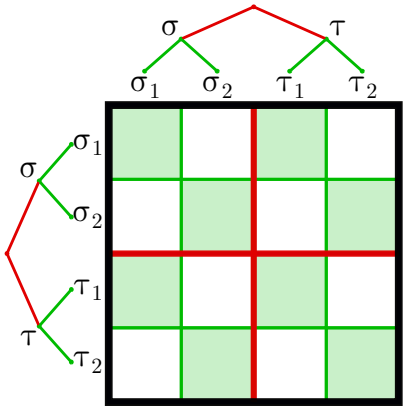
(a) Upper quotient graph.



(b) Lower quotient graph.

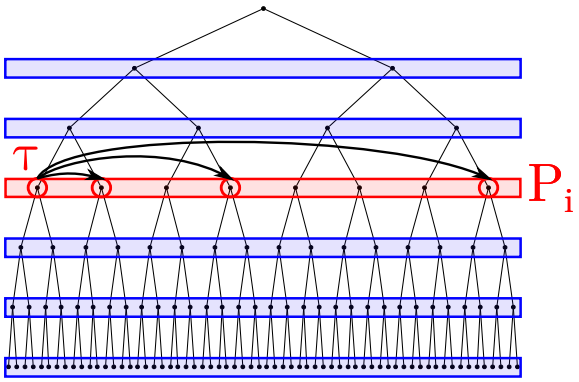


(c) Upper block matrix considered.

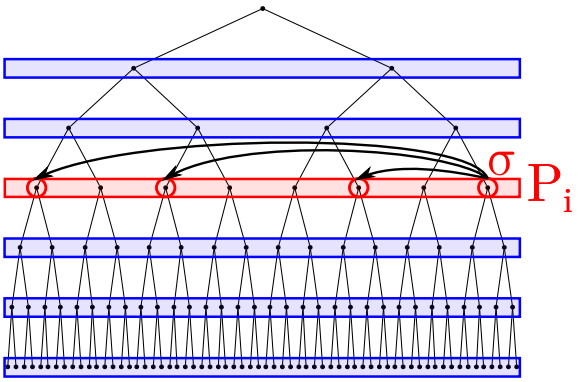


(d) Lower block matrix considered.

Figure 2.30: Two levels of hierarchy with the corresponding structures. The first level is colored red while the second is colored green. White matrix blocks represent null interactions.



(a) $\texttt{Reach}^{+}_{G/P}(\tau)$ stores the interactions of τ with clusters of larger indices.



(b) $\texttt{Reach}^{-}_{G/P}(\sigma)$ stores the interactions of τ with clusters of lower indices.

Figure 2.31: Partition P_i in a cluster tree, of which we have highlighted the different levels by blue horizontal lines. The partition is highlighted in red. This representation assumes that clusters are ordered from left to right.

Algorithm 30: Creation of an $\mathcal{H}$-Matrix with detection of fill-in using right-looking symbolic information.

```
Function CreateHMatrix(σ × τ)
1   if IsLeaf(σ × τ) then
2       if Admissible(σ, τ) then
3           Compress(σ × τ)                          ▷ Creation of a Rk-Matrix
4       else
5           CreateFullBlock(σ × τ)                   ▷ Creation of a Full-Matrix
6   else
7       for (σ', τ') ∈ Children(σ × τ) do
8           if σ' ∈ Reach+_{G/P}(τ') ∨ τ' ∈ Reach+_{G/P}(σ') then
9               CreateHMatrix(σ', τ')   ▷ Create (σ' × τ') only if σ' and τ' interact
```

have a coarser grain than the symbolic structure usually computed by a sparse direct solver (using for example the Block Column Symbolic Factorization presented in § 1.3.1.2.3) and may thus contain more zeros. As previously discussed in § 2.4.1, and especially supported by Fig. 2.26 (p. 112), the hierarchy must be constructed to an extremely fine grain (see Fig. 2.26a for example) to be capable of reaching the same granularity as a sparse direct solver (Fig. 2.26b). The practical interest of the hierarchical method is then lost and we therefore propose another solution to this problem by the computation of a row vertex symbolic information for column clusters, i.e., a Cluster-Vertex symbolic information. This symbolic information is computed using the CV-HSF method we now present.

This method relies on the same definition as the one introduced for the scalar adjacency of supernodes in § 1.3.1.2.3. The scalar (or vertex) adjacency of a cluster τ, using the adjacency graph $G = (V, E)$, is defined as the union of the adjacency of the unknowns it contains. In other words, it is composed of all the unknowns j with which at least one unknown i in τ is adjacent to:

$$\texttt{Adj}_G(\tau) = \{j \mid (\exists\, i \in \tau) \wedge (\exists\, (i,j) \in E)\}.$$

This structure describes the unknowns that interact with at least one unknown of the cluster τ. In other words, it corresponds to the non-zero rows in the block column corresponding to τ. To compute the symbolic factorization in this scenario, we rely on the structure $\texttt{Reach}^+_G(\tau)$. It is initialized as

$$\texttt{Reach}^+_G(\tau) \leftarrow \{j \mid (j > \tau) \wedge (j \in \texttt{Adj}_G(\tau))\}.$$

To compare the Cluster-Vertex information and the Cluster-Cluster information, one can compare the green and the light blue information in Fig. 2.35a, respectively. Such as discussed in § 1.3.1.2.3, the Cluster-Vertex information may be implemented as a list of intervals for efficiency considerations. The implementation of clusters also relies on intervals and the two structures are therefore implemented in the same manner.

2.4.2.1.3 Column Partition

In the case of a nested dissection, the partition of supernodes used for the symbolic factorization arises from the separators computed by the dissection. If local orderings are applied, such as a specific separator clustering (more extensively discussed in § 2.4.3) or the Minimum Fill heuristics for example on the local subdomains, the partition of supernodes is finer. However, these supernodes do not match the definition of exact supernodes given § 1.3.1.1.3 (T2 supernodes [70]). A block column associated with a separator may have different patterns of non-zeros among its columns (see "relaxed supernodes" p. 55). These supernodes correspond to the leaves of the (column) cluster tree in a hierarchical environment.

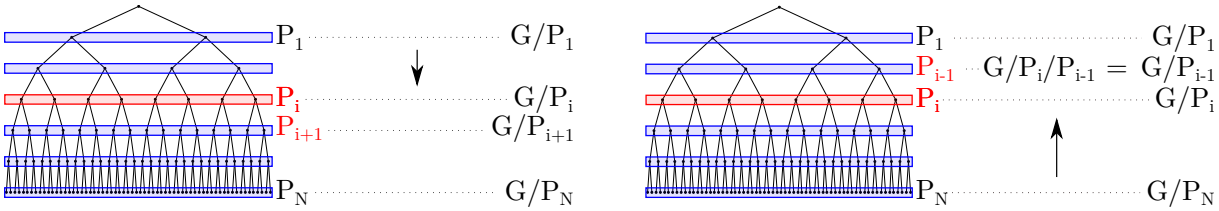
We do not investigate here the re-computation of supernodes after the initial ordering and clustering. Therefore, the symbolic factorization may be computed at multiple steps of the clustering. We may use the partition arising from (1) the nested dissection, (2) the nested dissection coupled with a local clustering of local subdomains (based on AMF in this thesis), or (3) the nested dissection and local clusterings in local subdomains and separators. By using local clusterings, the cost of computing a symbolic factorization increases, while the symbolic information computed matches more precisely the underlying pattern of non-zeros. In practice, a better symbolic information will be preferred as it leads to a reduction of the cost of the numerical factorization, even though the symbolic factorization may be slightly costlier. The cost of the symbolic factorization is indeed low compared to the numerical step, as it will be shown in the experiments in Chapter 3.

2.4.2.2 Hierarchical Symbolic Elimination

Now we focus on the calculation of the fill-in generated by the factorization and the corresponding symbolic information. This step must be performed before the creation of the $\mathcal{H}$-Matrix (Algorithm 30). The order of the computation of each symbolic factorization can be questioned. The differences between a top-down and bottom-up computation of quotient graph eliminations are first introduced in § 2.4.2.2.1. The optimization of the bottom-up algorithm is then discussed in § 2.4.2.2.2.

2.4.2.2.1 Naive Hierarchical Quotient Graph Eliminations

$\mathcal{H}$-Matrices are usually constructed in a top-down fashion. Therefore we first focus on a **top-down** elimination of quotient graphs, based on the partitions of the cluster tree P_i (with i the depth of the partition). This top-down algorithm is described in Algorithm 31 and illustrated in Fig. 2.32a. The function `SymbolicFactorization`(G, P, N) is a black-box wrapper for any symbolic factorization presented in § 1.3.1.2 (right/left-looking and computing a Cluster-Cluster or Cluster-Vertex information, such as discussed in the previous section). `Children`(P) and `Parents`(P) compute the lists of children and parents of clusters in P, respectively. Such a top-down elimination must recalculate each quotient graph G/P_i based on the interactions in G, as the information of the



(a) Top-down algorithm. Each quotient graph G/P must be recalculated from G.

(b) Bottom-up algorithm. Each quotient graph G/P can be aggregated from the one below.

Figure 2.32: The order of the computation of partitions for the Hierarchical Symbolic Factorization is given by the direction of the arrow.

upper quotient graph does not a priori provide the required information to distinguish the interactions of lower clusters. The information of the upper level is less precise in terms of location of non-zeros than the lower level. Indeed, in the example in Fig. 2.30, we can see that we cannot determine if τ_1 interacts with σ_1 (and/or with σ_2) if we only know that τ interacts with σ. We only have a partial information of which cluster it interacts with. To compute this information, we have to re-use the scalar edges and see which ones have an endpoint in τ_1 and another in σ_1. If σ does not interact with τ, we know however that σ_1 does not interact with τ_1. What we are interested in is a tool able to find the most precise pattern of non-zeros at each level. On the contrary, a **bottom-up** algorithm can simply aggregate the information of a quotient graph based on the partition P_{i+1} to form the quotient graph on the upper level P_i. The knowledge of the interaction between τ_1 and σ_1 is sufficient to deduce the interactions between τ and σ. First, for the sake of clarity,

Algorithm 31: Top-Down Hierarchical Symbolic Factorization based on the scalar adjacency graph G partitioned into a list P of supernodes. P is initialized as the topmost level of the cluster tree.

```
Function TD-HSF(G, P)
1   SymbolicFactorization(G, P, |P|)
2   if Children(P) ≠ ∅ then
3       P ← Children(P)
4       TD-HSF(G,P)
```

a naive version of the bottom-up algorithm is detailed in Algorithm 32 and illustrated in Fig. 2.32b. The possibility of using the symbolic information of lower levels is further discussed in § 2.4.2.2.2. Note that the top-down and the naive bottom-up algorithms first compute the symbolic factorization (either block or block column) of the upper and lower partitions, respectively. They then compute a symbolic factorization on the next partition (lower or upper, respectively), and iterate in the same manner until they have covered the whole cluster tree.

Algorithm 32: Bottom-up Hierarchical Symbolic Factorization based on the scalar adjacency graph G partitioned into a list P of supernodes. P is initialized as the list of leaves of the cluster tree.

```
Function BU-HSF(G, P)
1   if |P| > 1 then
2       SymbolicFactorization(G, P, |P|)
3       G ← G/P
4       P ← Parents(P)
5       BU-HSF(G,P)
```

However, in the context of unbalanced cluster trees, we must take into account the shorter branches for the lower levels in which they do not exist (see Fig. 2.28). Algorithm 31 becomes Algorithm 33, in which the function `Children(`P`)` becomes a function `Lower(`P`)`, which computes the lower quotient graph to be used. The function

Algorithm 33: Top-Down Hierarchical Symbolic Factorization for unbalanced cluster trees. P is initialized as the topmost level of the cluster tree.

```
Function TD-HSF(G, P)
1   SymbolicFactorization(G, P, |P|)
2   if Children(P) ≠ ∅ then
3       P ← Lower(P)
4       TD-HSF(G,P)
```

`Lower(`P`)` (Algorithm 34) computes the children of each element in P, except if the element does not have any child, in which case the element is kept in the partition.

Algorithm 34: Function to compute the lower quotient graph to be used in the top-down Algorithm 33.

```
Function Lower(P)
1   P' ← ∅
2   for τ ∈ P do
3       if IsLeaf(τ) then
4           P' ← P' ∪ τ                          ▷ re-use same cluster
5       else
6           P' ← P' ∪ Children(τ)
```

Regarding the bottom-up algorithm, Algorithm 32 becomes Algorithm 35, in which the function `Parents(`P`)` becomes a function `Upper(`P`)`, which computes the upper

quotient graph to be eliminated next (Algorithm 36). The function `Upper(`P`)` may be limited to the computation of the parents of P in the case of a balanced tree. But if the cluster tree is unbalanced, we must use the parents of the elements in P except those highest in the hierarchy if they are not on the same level.

Algorithm 35: Bottom-up Hierarchical Symbolic Factorization for unbalanced cluster trees. P is initialized as the list of leaves of the cluster tree.

```
Function BU-HSF(G, P)
1   if |P| > 1 then
2       SymbolicFactorization(G, P, |P|)
3       G ← G/P
4       P ← Upper(P)
5       BU-HSF(G,P)
```

Algorithm 36: Function to compute the upper quotient graph to be used in the bottom-up Algorithm 35.

```
Function Upper(P)
1   P' ← ∅
2   if ∀σ, τ ∈ P, Depth(σ) = Depth(τ) then
3                                          ▷ all clusters are on the same level
4       return Parents(P)
5   else
6       for τ ∈ P do
7           if Depth(τ) > min_{σ∈P}(Depth(σ)) then
8               P' ← P' ∪ Parent(τ)
9           else
10              P' ← P' ∪ τ                ▷ re-use same cluster
```

With the functions `Upper(`P`)` and `Lower(`P`)`, some clusters may be included in multiple layers of quotient graphs. Thus, to satisfy the definitions introduced in § 2.4.2.1, a cluster τ must simply keep either the information of (1) the leaf clusters if τ is a leaf or (2) the clusters at the same depth as τ (including the leaves of shorter branches) .

2.4.2.2.2 Bottom-Up Transmission of Elimination

As mentioned earlier, some computations in the previous naive bottom-up (and a fortiori top-down) algorithms are redundant. Indeed, the edges computed through each symbolic factorization are computed in the lower levels too. Instead of computing the elimination of the quotient graph of each level, we can use the information of the symbolic factorizations of the lower levels to compute that of an upper level. In other words, we can transfer the information of the edges of a quotient elimination graph $(G/P_{i+1})^*$ to the

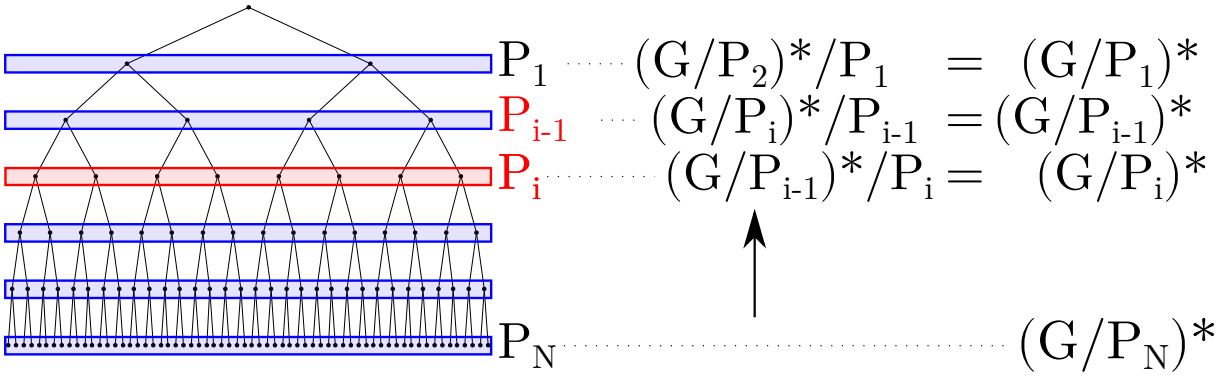


Figure 2.33: Aggregation of the edges of the lower elimination quotient graph $(G/P_{i-1})^*$ to form the upper quotient graph $(G/P_i)^*/P_{i+1} = (G/P_{i+1})^*$.

parents of the nodes of this graph, instead of re-computing the edges of the upper quotient graph $(G/P_i)^*$. Therefore, with the elimination graph $G' = (G/P_{i+1})^*$, the upper quotient elimination graph can be computed through G'/P_i . We provide a more efficient way to compute the upper symbolic factorizations directly by using the parents of each cluster listed in the interaction structures $\texttt{Reach}^+(\tau)$ or $\texttt{Reach}^-(\tau)$ to match the interactions of the parent of τ. This leads to Algorithm 37, illustrated by Fig. 2.33. In this

Algorithm 37: Simplified Bottom-Up Hierarchical Symbolic Factorization. P is still initialized as the list of leaves of the cluster tree τ.

Function `HSF`(G, τ)
1 $P \leftarrow$ `Leaves`(τ)
2 `SymbolicFactorization`$(G, P, |P|)$ ▷ Elimination of the leaves
3 `computeInteractions`(τ) ▷ Recursive elimination of the upper levels

algorithm, we compute a symbolic factorization once on the leaves of the cluster tree with the root τ (line 2). This symbolic factorization does not need to be any specific algorithm (scalar, block, block column, left-looking,...), although we focus in this thesis on the use of a block symbolic factorization or block column symbolic factorization, leading to the Cluster-Cluster and Cluster-Vertex variants. In fact, in considerations of the left-looking structure needed in § 2.4.3, we may prefer to use a left-looking cluster-vertex symbolic factorization.

After this first symbolic factorization, we hierarchically compute the interactions of each sub-cluster in τ by aggregating the interactions of its children. Therefore, the information of the leaves is transmitted to their parents and so on. Algorithm 38 describes the recursion to compute the interactions on all clusters. The recursion is done on line 2 and the transmission to the parents on line 3. Algorithm 39 describes how the symbolic information of the children of τ is used for the computation of the symbolic information of τ in the case of CC-HSF. Algorithm 38 assumes that the initial `Reach`(τ) (initialized as `Adj`(τ)) is already consistent with the block cluster tree (see Eq. (1.10)),

Algorithm 38: Bottom-up hierarchical computation of interactions of a cluster tree τ using the interactions of its children.

```
Function computeInteractions(τ)
1   for τ' ∈ Children(τ) do
2       computeInteractions(τ')                          ▷ Recursion
3       Reach(τ) ← Reach(τ) ∪ Upper(τ, Reach(τ'))        ▷ Transmission to upper
```

Algorithm 39: Function to compute the interactions of τ based on its children to be used in Algorithm 38.

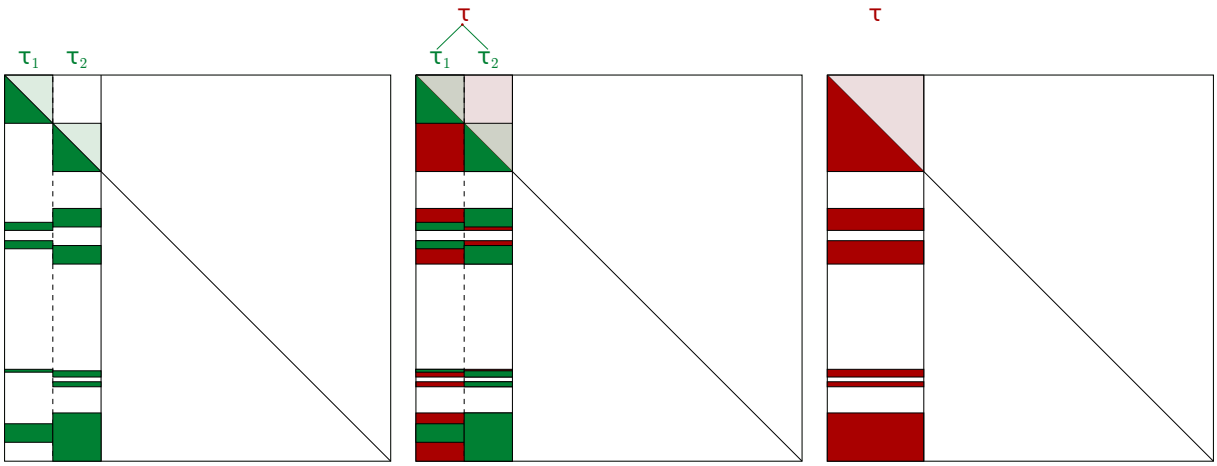
```
Function Upper(τ, P)
1   P' ← ∅
2   for σ ∈ P do
3       while Depth(σ) > Depth(τ) do
4           σ ← Parent(σ)          ▷ Get ancestor until we reach desired depth
5       P' ← P' ∪ σ                ▷ σ and τ on the same level
```

i.e., $\sigma \in \texttt{Reach}(\tau) \Rightarrow \texttt{Depth}(\sigma) = \texttt{Depth}(\tau)$. If $\texttt{Reach}(\tau)$ is not initialized as such, we can use Algorithm 39, which computes $\texttt{Upper}(\tau, \texttt{Reach}(\tau))$, to level the structure to the same level as τ. It relies on computing, for each element of this structure, the ancestor at the same depth as τ.

2.4.2.2.3 Specificities of the CV-HSF Bottom-Up Elimination

In the case of CV-HSF, the transmission of information to the upper levels of the cluster tree may be computed in two ways. The first way is to simply implement the function $\texttt{Upper}(\tau, P)$ to return P. In that case, a row vertex information is transmitted to each cluster belonging to the column cluster tree and therefore the Cluster-Vertex characteristic is maintained in the whole hierarchy. The second way is to transform the row vertex information into a cluster row information, leading to a hybrid structure with Cluster-Vertex information on the leaves of the column cluster tree and Cluster-Cluster information on the upper levels. The two ways are illustrated in Fig. 2.34 and Fig. 2.35, respectively. In this thesis, only the first one has been implemented. Thereby, CV-HSF only uses Cluster-Vertex information and we may fully study its differences with CC-HSF and the advantages of both methods.

An $\mathcal{H}$-Matrix constructed using CC-HSF is shown in Fig. 2.36. The exploitation of the information of CV-HSF (hence with a finer grain) is further discussed in § 2.4.3.4. The advantage of using a symbolic factorization may be observed through the reduction of the number of dense empty submatrices (colored yellow) here. With this Hierarchical Symbolic Factorization, we are thus able to detect filled entries and submatrices, and

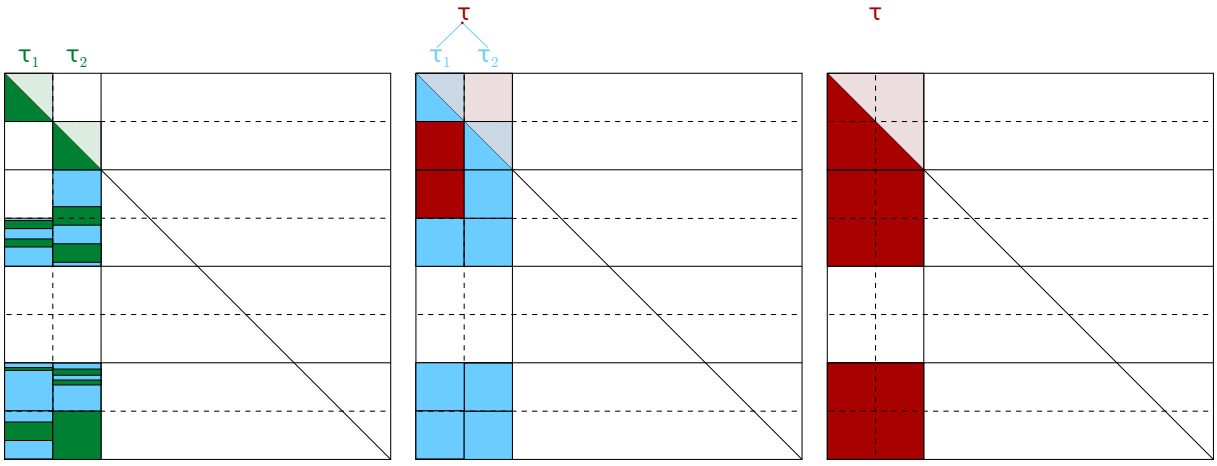


(a) Row vertex information on leaves of the cluster tree.

(b) Merging this information on the upper level.

(c) Upper level information.

Figure 2.34: Transmission and merging of Cluster-Vertex information. Green blocks represent the vertex information of each leaf cluster (τ_1 and τ_2 here). The red blocks represent the vertex information of the upper level (merged together) cluster (τ). The actual storage of the $\mathcal{H}$-Matrices is located only on the leaves of the $\mathcal{H}$-Matrix so ultimately, only the green blocks may be stored. Red blocks only give the information to the algorithm that non-zeros exist in this branch of the hierarchy and that it must recurse on lower levels to find more precise information.



(a) Row vertex information transformed into cluster information on leaves of the cluster tree.

(b) Merging this information on the upper level.

(c) Upper level information.

Figure 2.35: Transmission and merging of block information. Green blocks represent the Cluster-Vertex information. Light blue blocks represent the information of each leaf cluster (τ_1 and τ_2 here) and stored by the actual $\mathcal{H}$-Matrix in a Cluster-Cluster manner. Red blocks represent the information of the upper cluster (τ).

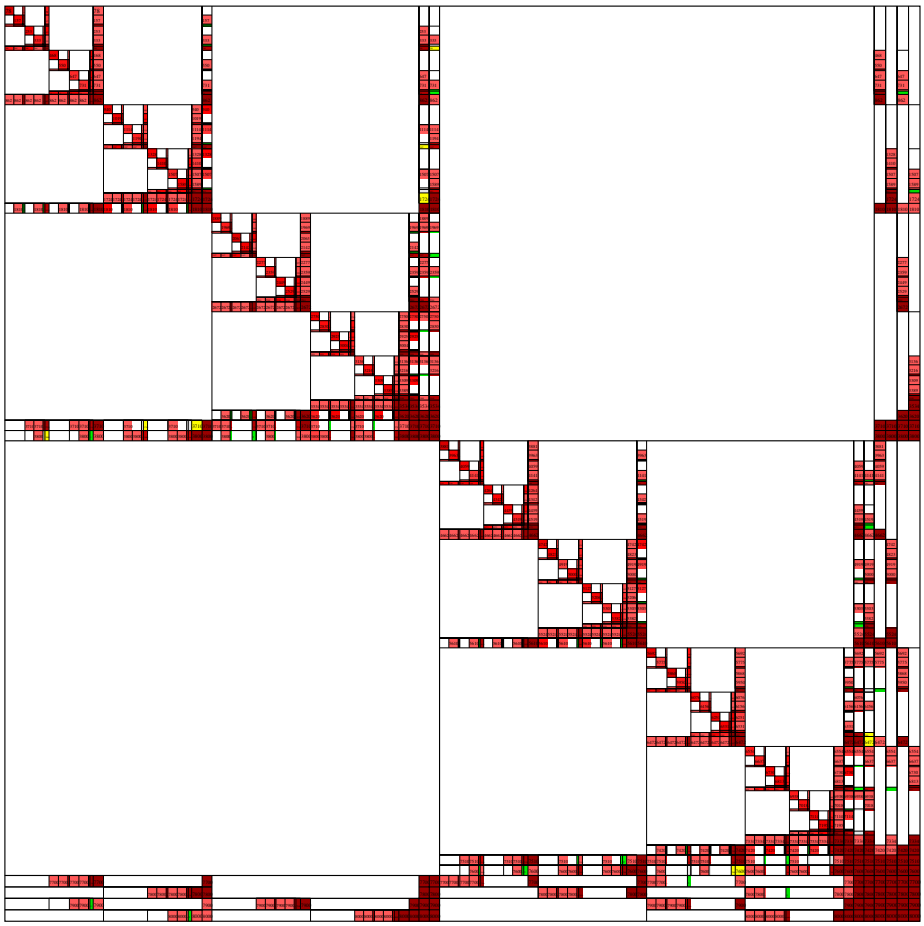


Figure 2.36: An $\mathcal{H}$-Matrix constructed using nested dissection and detection of the location of zeros (using CC-HSF). We can see that zero blocks (full of zeros until the very end of the computations) previously stored as dense (yellow) or $\mathcal{R}k$-Matrix (gray) in Fig. 2.14 are no longer stored here. See legend in Fig. 2.4.

avoid the storage of exclusively zero-filled blocks, even at higher levels in the hierarchy. In a usual $\mathcal{H}$-Matrix, a zero block is often stored as a $\mathcal{R}k$-Matrix, and more rarely, as a Full-Matrix. This method avoids the heavy cost of dense storage of some null matrices as a Full-Matrix, but also the computation involving a $\mathcal{R}k$-Matrix with a rank equal to 0. As our solver relies on task-based parallelism, this also implies that no task is required to be executed by the runtime system in charge of managing the tasks.

2.4.2.3 Complexity of the Hierarchical Symbolic Factorization

The arithmetic complexity of the simplified bottom-up algorithm presented in § 2.4.2.2.2 is therefore equal to the sum of (1) the complexity of the symbolic factorization computed on the quotient graph $G/P = (P, E/P)$, with P the leaves of the cluster tree, and (2)

the aggregation of the edges of this quotient graph to the upper levels. The aggregation complexity can be bounded by $\mathcal{O}(|E/P|)$ for each level, even though the actual complexity should be lower for higher levels in the hierarchy. The complexity of the (block) symbolic factorization is also $\mathcal{O}(|E/P|)$. Therefore the total complexity can be bounded by $\mathcal{O}(|E/P| \times \log_2 n)$, with n the number of unknowns and Eq. (1.7). It is therefore more costly than a simple (flat) symbolic factorization. This complexity also ensures that it is not more costly than the numerical factorization though in practice its cost is much less than that of the numerical factorization. If we make the assumption that the number of edges $|E/P|$ is divided by at least two at each level, then the complexity may even be reduced to

$$\mathcal{O}\left(|E/P| \times \sum_{k=0}^{k=\log_2 n} 2^{-k}\right) = \mathcal{O}\left(|E/P| \times \frac{1-2^{-(\log_2 n+1)}}{1-2^{-1}}\right)$$
$$= \mathcal{O}\left(|E/P| \times 2(1-\frac{1}{2n})\right) \leq \mathcal{O}\left(2|E/P|\right). \tag{2.14}$$

In that case, the hierarchical symbolic factorization has a much lower arithmetic complexity than the numerical factorization. A similar reasoning can be applied for the bottom-up elimination in the case of the CV-HSF variant presented in § 2.4.2.2.3, where the number of edges $|E/P|$ is transformed into an asymmetric structure indicating interactions between clusters and vertices, i.e., edges in $V/P \to V$.

Furthermore, if we take into account the assumption that the number of off-diagonal blocks grows with the number of unknowns n associated with the matrix, following Theorem 2.1 from [61], the algorithm has a linear complexity, i.e., in $\mathcal{O}$(n). This assumption is verified at least in the CV-HSF algorithm using a column partition arising from the nested dissection (see § 2.4.2.1.3).

2.4.3 Separator Clustering

Let us consider a cluster tree constructed using a global clustering method computing a nested dissection (coupled or not with a local subdomain clustering such as AMF) as discussed in § 2.3.1. We now discuss separator clustering methods.

The interaction of a separator with itself is considered dense. Consequently, the approach of the $\mathcal{H}$-Matrix community divides the separators using recursive bisection, in the same manner as domains are divided for dense applications (§ 1.2.3.2.2). In an effort to compute a structure matching the underlying pattern of non-zeros, we also consider the clustering of the separator based on the interactions of other clusters, i.e., the information computed by symbolic factorization, in a flat manner § 2.4.3.2 and in a hierarchical manner § 2.4.3.3. It should be noted that the ordering inside separators does not impact the fill-in generated by factorization. Therefore the considerations we have to take into account here rely on matching the pattern of non-zeros to reduce storage, and compression and cache efficiencies. To better understand the effect of each method, only a subpart of the whole matrix is displayed in this section, indicated in red in Fig. 2.37, along with the corresponding sub-cluster tree.

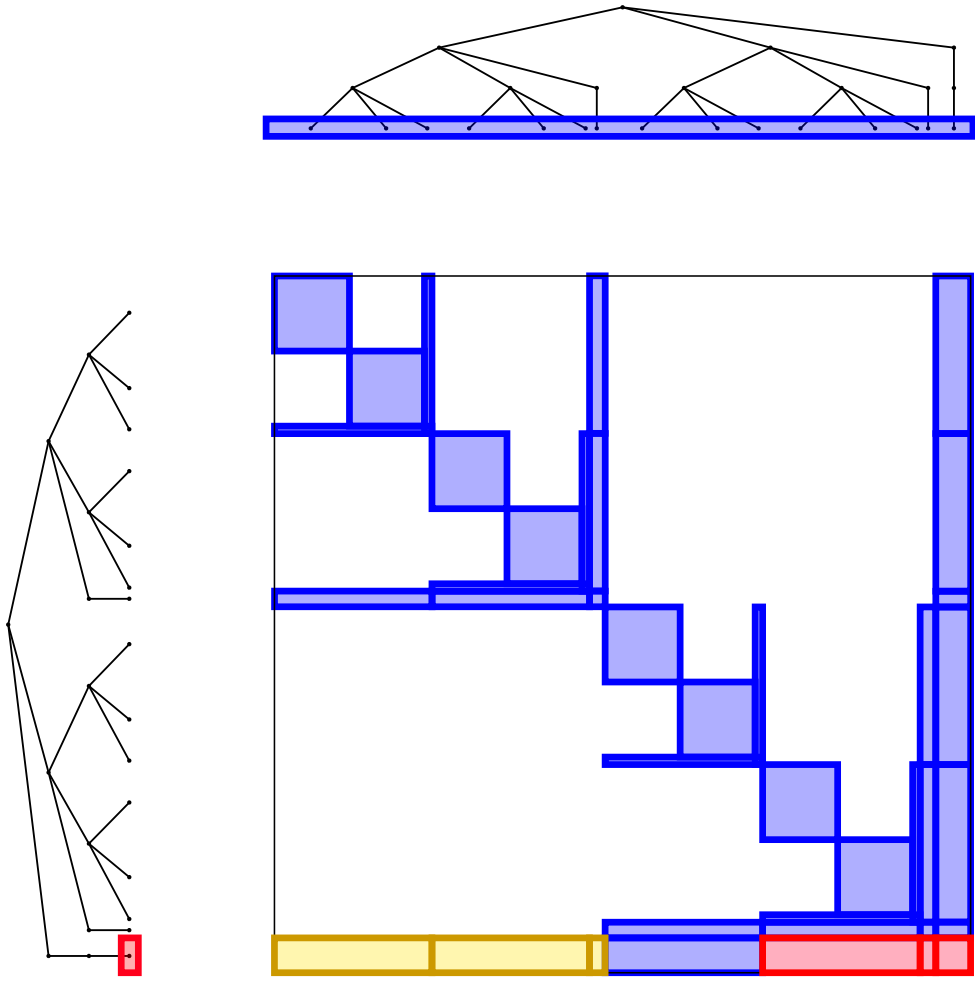


Figure 2.37: Subparts of the matrix considered in red and yellow, as well as the considered subpart of the cluster tree, also in red.

2.4.3.1 Interactions Oblivious (IO) Separator Clustering

Geometric approach In an effort to lead to an efficient compression in separators, $\mathcal{H}$-Matrices relying on nested dissection (§ 2.1.2) apply a recursive bisection to divide the separator and create a binary subtree [138, § 3.3]. An example is given in Fig. 2.38.

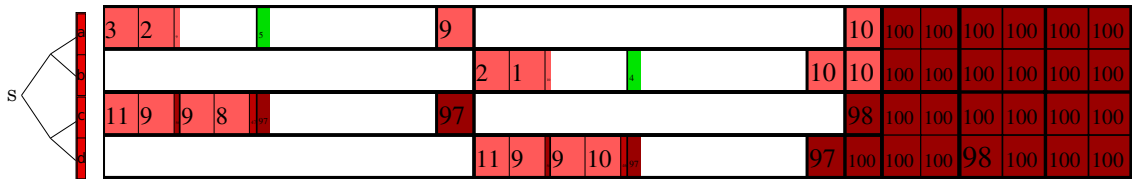


Figure 2.38: Separator clustering based on recursive bisection. Right (red) subpart of Fig. 2.37.

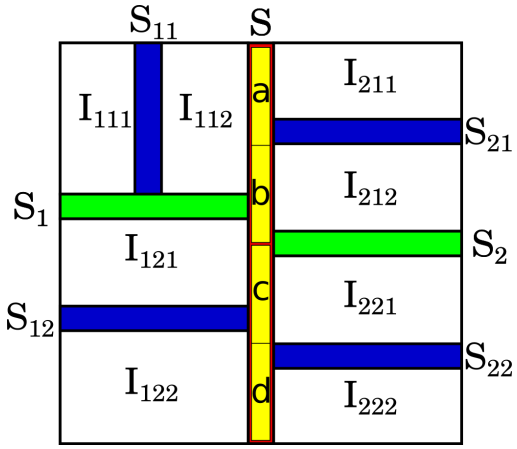


Figure 2.39: Adjacency graph of a nested dissection with recursive bisection applied on separators.

a	$[I_{112}, I_{211}, S_{21}]$
b	$[I_{112}, I_{121}, S_1, I_{212}, S_2]$
c	$[I_{121}, I_{122}, S_{12}, I_{221}, S_2]$
d	$[I_{122}, I_{222}, S_{22}]$
$\{a, b\}$	$[I_{112}, I_{121}, S_1, I_{211}, I_{212}, S_{21}, S_2]$
$\{c, d\}$	$[I_{121}, I_{122}, S_{12}, I_{221}, I_{222}, S_{22}, S_2]$

Table 2.2: Examples of left-looking interactions for sub-clusters of S in Fig. 2.39, i.e., the list of leaf clusters that reaches them.

This differs from the usual sparse methods based on nested dissection. However, it is related to the notion of K-way partitioning discussed in [133] for example and used in [164] on separators. Recursive bisection is applied due to the consideration that diagonal blocks resulting from the interactions of a separator with itself lead to dense submatrices after factorization. Therefore the standard division of $\mathcal{H}$-Matrices may be used to get large compression blocks and efficient parallelism such as may arise in a dense application. However, this does not take into account the off-diagonal blocks formed with this construction. Indeed, recursive bisection may lead to a possible overlap between a separator's subclusters interactions and neighboring clusters due to an irregular division such as shown in Fig. 2.39. We want to minimize the number of neighbor clusters contributing to one separator subcluster. Note that such neighbor clusters are always ordered before the separator. Examples of lists of such left-looking (or row) interactions $\texttt{Reach}^{-}(\tau)$ of a cluster τ are listed in Table 2.2, among a partition P chosen as the leaf clusters.

For example, in Fig. 2.39, d does not interact with I_{221}. But we can see here that the cluster named c, computed through the recursive bisection, does not match exactly its neighbor I_{121} and I_{221}: it is also connected to the cluster I_{122} (among others) even though they have only little interactions. This is a factor of unnecessary zero storage, that will lead us to use a partitioning aware of the outer interactions of a cluster as detailed in §§ 2.4.3.2 to 2.4.3.4.

Topological approach In the topological SCOTCH variants discussed in §§ 2.3.1.2 and 2.3.1.3, we may apply different strategies for the local separator clustering. As discussed in appendix A, the strategies considered for separators are here either the 'simple' strategy, which orders the unknowns in the separator in their natural order (such as given by the user), or the 'Gibbs-Poole-Stockmeyer' method [96], which tries

to minimize the number of off-diagonal blocks. The algorithm relies on the ordering computed by SCOTCH and applies a recursive bisection to cluster the unknowns in a hierarchical manner. Note that this recursive bisection does not reorder the unknowns. Example of these clusterings are shown in § 2.4.4.2. We may also try to combine the methods included in this approach with the Interactions-Aware clustering methods discussed in §§ 2.4.3.2 to 2.4.3.4.

2.4.3.2 Interactions Aware Flat Separator Clustering (IA-FSC)

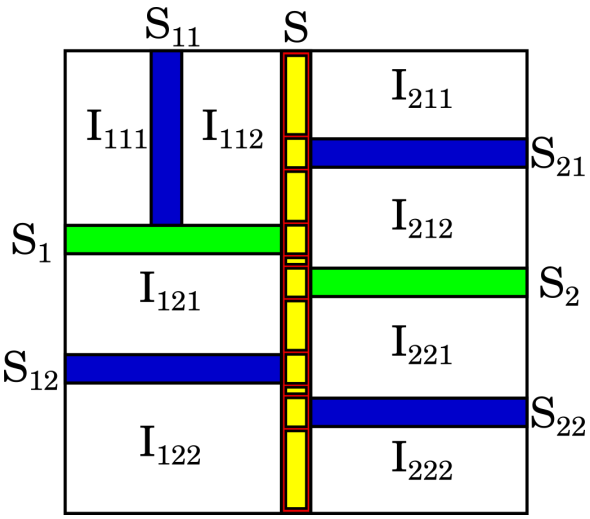
With the hierarchical symbolic analysis described in § 2.4.2, we lack the possibility to create a structure fitting the pattern of non-zeros. We therefore investigate here the possibility to divide separators to match more precisely this pattern, based on the symbolic information of the interactions between clusters. As these clusters rely on the interactions with their predecessor, i.e., rows in the (lower) factor L of the LU factorization, they are essentially T4 symmetric supernodes such as presented in [70]. However, we first compute supernodes using nested dissection, leading to supernodes matching the definition of (relaxed) T2 supernodes. Then we use the (left-looking) information of each row to compute the T4 supernodes. This means that these supernodes depend on the computed T2 supernodes, and especially relaxation will play an important part in their structure.

We have mentioned earlier that the block c, a descendant of the first separator S, is linked not only to its adjacent neighbor I_{121} and I_{221} but to I_{122} as well, as shown in Fig. 2.39 and Table 2.2. In order to minimize the overlap between separator clusters and adjacent clusters, the separator is divided according to the interactions of its unknowns: the unknowns with the same interactions (Fig. 2.40a) are grouped together to form a unique child of the separator in the cluster tree (Fig. 2.40b and 2.40c). Algorithm 40 shows the procedure used to divide each separator following the interactions of each unknown with other leaf clusters.

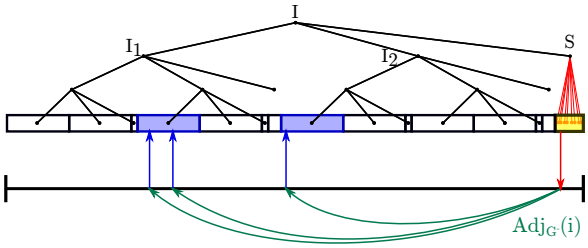
Algorithm 40: Interactions Aware Flat Separator Clustering.

```
Function IA-FSC(τ)
1   if IsSparse(τ) then
2       for τ' ∈ Children(τ) do
3           IA-FSC(τ')
4   else
5       P ← InitialPartition()                                   ▷ Leaves
6       ComputeInteractionsWithQuotientGraph(τ, P)   ▷ Reach⁻_{G/P}(i), ∀i ∈ τ
7       Children(τ) ← SplitByInteractions(τ)                     ▷
         ∀i, j ∈ Child, Reach⁻_{G/P}(i) = Reach⁻_{G/P}(j)
```

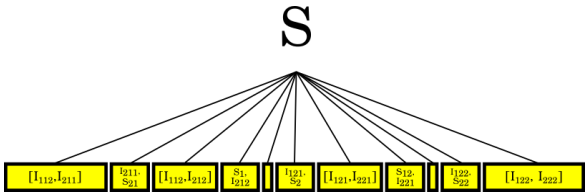
The function `ComputeInteractionsWithQuotientGraph`(τ, P) assigns a list of interactions to each unknown in τ based on their (left) interactions through the quotient



(a) Separator subdivided following its interactions with the whole domain division.



(b) Clustering of the separator according to its interactions with the cluster tree leaves. The quotient adjacency (green arrows) of an unknown i (red arrow) is first computed. The list of clusters with which i interacts is highlighted in blue. The segment under the cluster tree represents the scalar list of unknowns. The same principle is applied to the other separators.



(c) Subtree under the first separator.

Figure 2.40: Interactions Aware Flat Separator Clustering. Following a global clustering (such as nested dissection), separators are divided using symbolic information.

graph. In other words, for a quotient graph $G/P = (P, E/P)$ an unknown i is attributed the structure $\texttt{Reach}^-_{G \to G/P}(i)$, or $\texttt{Reach}^-_{G/P}(i)$ for convenience, consisting in a list of clusters of P, ordered before i. The partition P is chosen as the leaves of the cluster tree in order to have the best precision on the location of non-zeros in the resulting matrix. Each step of the computation of the structure $\texttt{Reach}^-_{G \to G/P}(i)$ is shown in Fig. 2.40b. The computation of the quotient interactions based on the scalar interactions may be performed in two ways. We can either (1) search for the cluster containing all the elements in the scalar list among the whole partition P or (2) use a pointer for each unknown to the leaf cluster containing this unknown. As the memory used by pointers may be freed once the construction of the $\mathcal{H}$-Matrix is finished, we rely on the second option, of which the search can be computed in constant time.

Then, τ is divided into a list of smaller clusters by $\texttt{SplitByInteractions}(\tau)$, grouping the unknowns by their list of interactions. In order to do that, we must first sort the unknowns by their interactions, either using a lexicographic order or a shortlex order. The latter first orders elements by their length and then follows a lexicographic order. Let a, b and c be clusters from a partition P, and unknowns i, j and k have the following lists of interactions:

- $\texttt{Reach}(i) = [a, b]$;
- $\texttt{Reach}(j) = [a, b, c]$;
- $\texttt{Reach}(k) = [a, c]$.

In a lexicographic order, we have $[a, b] < [a, b, c] < [a, c]$ while the shortlex order leads to $[a, b] < [a, c] < [a, b, c]$. The lexicographic order thus leads to the ordering $i < j < k$ while the shortlex order leads to $i < k < j$.

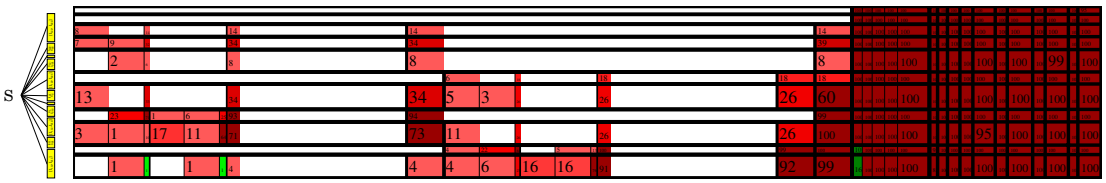
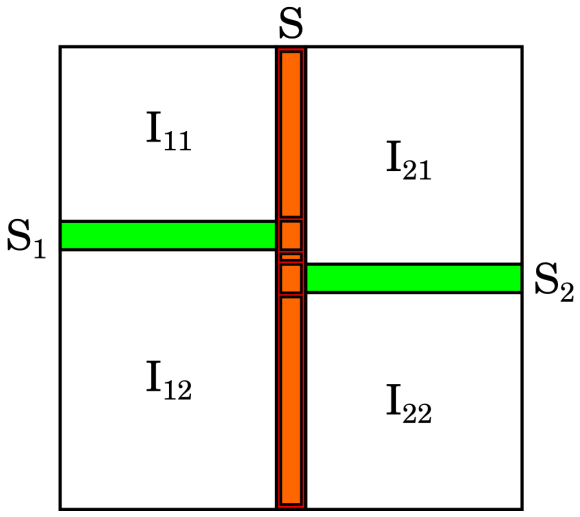


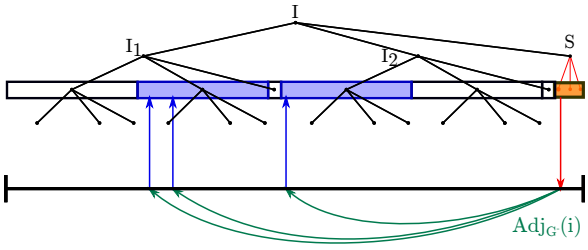
Figure 2.41: Separator clustering based Interactions-Aware Flat Separator Clustering. Right (red) subpart of Fig. 2.37. Legend in Fig. 2.4.

As the separator is not hierarchically divided with this method (its children being the finest possible straightaway), we refer to this clustering as an *Interactions Aware Flat Separator Clustering* (IA-FSC). This method may be compared to sparse direct solvers relying on BLR compression, as there is only one level of recursion in separators. It should be noted that our implementation of this approach can be categorized as a right-looking method: updates are applied directly on the destination panel, without any accumulation such as presented in [44]. Fig. 2.41 shows a part of the $\mathcal{H}$-Matrix constructed using IA-FSC (including the first separator).

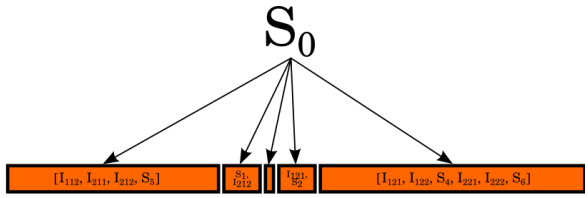
There are two downsides relative to this method: the separator is divided into too many and too small parts; and each of them is at the same level as the other ones in the hierarchy. The loss of hierarchy in the separators leads to a poor compression ratio. We will discuss below (in § 2.4.3.3) a hierarchical variant of this algorithm. Regarding the other problem, we may resort to relaxation to amalgamate blocks together to reduce the number of parts in the clustering. Some clusters typically consist of only two to three unknowns (similar to what has been observed in § 1.3.1.1.3). A threshold is set by the user, for example to a value $N_{relax} = 20$, under which we merge some of the computed clusters together until we reach this threshold. Another solution could be an aggregation computed when creating the block cluster tree instead of the cluster tree. For example, we could merge some row children clusters together to form one larger row cluster if there are too many on a specific submatrix. This is close to what is discussed in § 2.4.3.4, with the difference that the aggregation discussed there is performed on a scalar level, instead of on clusters. To achieve an amalgamation consistent with the hierarchy, the partition P of clusters used to calculate $\texttt{Reach}^-_{G/P}(i)$ could also be chosen as being higher in the hierarchy. A higher partition induces larger clusters to be formed. Indeed, the higher the partition is, the fewer elements the partition consists of. From a mathematical point of view, there are fewer possible combinations of elements from a small set than from a larger set. So if P is smaller, there are more unknowns i with an identical $\texttt{Reach}^-_{G/P}(i)$.



(a) Partition (level) of the cluster tree considered.



(b) Clustering of the separator according to its interactions with the upper partition.



(c) Subtree under the separator S.

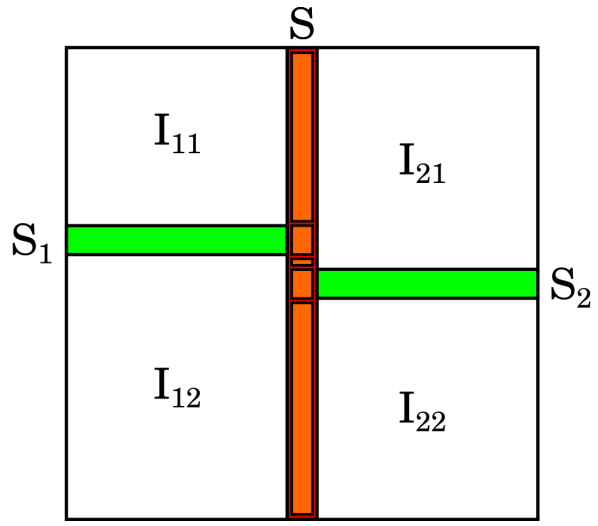
Figure 2.42: Higher partition chosen to create larger blocks.

However, this has not been further investigated in this thesis. We rather focus on a hierarchical clustering such as discussed in the following paragraph.

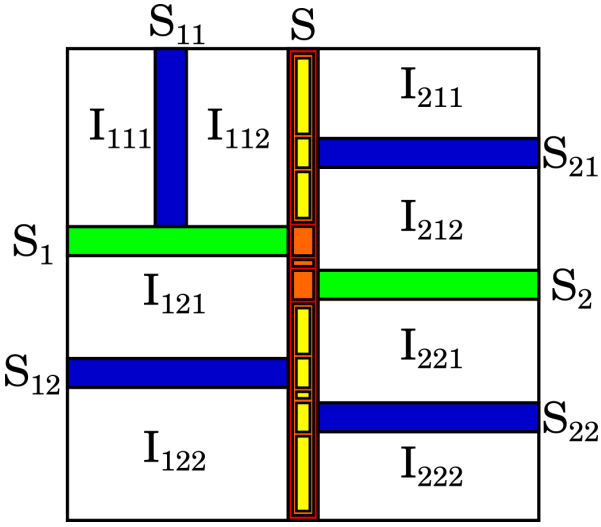
2.4.3.3 Interactions Aware Hierarchical Separator Clustering (IA-HSC)

To benefit from the hierarchical properties of $\mathcal{H}$-Matrices using nested dissection without any introduction of extra zeros (using symbolic information), we introduce a new algorithm: an Interactions-Aware Hierarchical Separator Clustering (IA-HSC). Starting from the partition of IA-FSC into well-separated clusters (using their interactions), the objective is to create a hierarchy between this partition and the separator cluster, i.e., introduce intermediate levels of hierarchy in Fig. 2.40c. There are two ways to achieve this: a bottom-up algorithm relying on aggregation, and a top-down algorithm relying on recomputation. Both algorithms are closely related to the top-down and bottom-up hierarchical block symbolic factorizations presented in § 2.4.2.

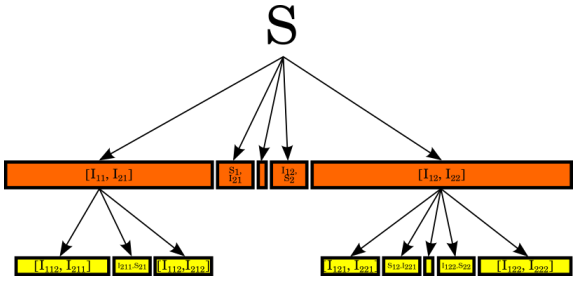
We start from a nested dissection resulting into a partition (τ_1, τ_2, τ_S). To perform a **top-down** clustering, we recursively divide each cluster under the separator τ_S (starting with τ_S) according to the interactions of its unknowns with the descendants of the cluster siblings of τ_S, i.e., of τ_1 and τ_2 (Fig. 2.43b and 2.43c). We can see on these figures the first two levels of hierarchy of this algorithm, thus reaching the same level of granularity (yellow clusters) as for IA-FSC. The procedure is defined in Algorithm 41. It mainly relies on the same recursion as IA-FSC, i.e., searching the cluster tree to find separators (which are considered as not sparse-labeled clusters), but relies on another recursive function on



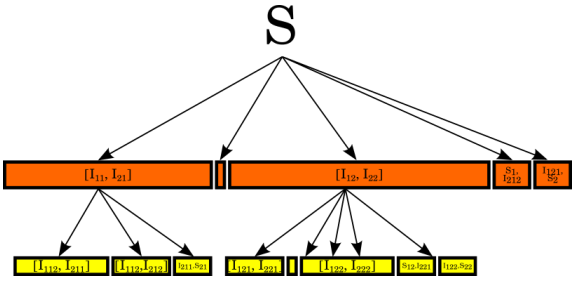
(a) The first level of hierarchical domain division based on symbolic factorization.



(b) The second level of hierarchical domain division based on symbolic factorization.



(c) Subtree under the first separator. The same principle is applied to the other separators. Clusters are ordered from left to right following their geometric position.



(d) Shortlex (and lexicographic here due to the representation) order of (c). Theoretically, there could be a small cluster in the middle (second children of S). In practice, as separators are often aligned, the shortlex order accumulates small clusters towards the end, while the lexicographic order accumulates them between the two large clusters.

Figure 2.43: Interactions Aware Hierarchical Separator Clustering. Following nested dissection (resulting in a ternary cluster tree where the leaves are separators and the independent subdomains), we divide the separators using a symbolic factorization.

each separator: `DivideSeparator`(τ_S), defined in Algorithm 42. This function divides a cluster into a list of sub-clusters depending on the interactions of its unknowns with a partition P (a list of clusters) given by the user. This partition is essentially a quotient graph, as discussed previously in § 2.4.2. Here, the initial partition P is set as the nephews of the separator τ_S, i.e., the children of τ_1 and τ_2. Then, at each new level of recursion, a new partition is computed. Note that this initial partition ensures the clusters are all

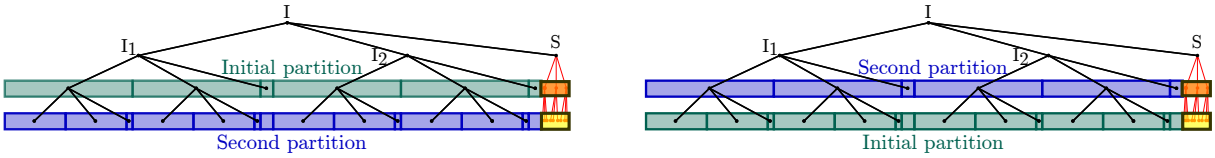
Algorithm 41: Top-Down Interactions Aware Hierarchical Separator Clustering.

```
Function TD-IA-HSC(τ)
1   if IsSparse(τ) then
2       for τ' ∈ Children(τ) do
3           TD-IA-HSC(τ')
4   else
5       P ← InitialPartition()                          ▷ Nephew clusters
6       DivideSeparator(τ, P)
```

Algorithm 42: The top-down recursive clustering of a separator τ used in the Interactions-Aware Hierarchical Separator Clustering, based on the interaction of τ with a partition P of clusters.

```
Function DivideSeparator(τ, P)
1   if |τ| > N_leaf then
2       ComputeInteractionsWithQuotientGraph(τ, P)   ▷ Compute unknowns
          interactions based on P
3       Children(τ) ← SplitByInteractions(τ)  ▷ Split τ into a list of children
          clusters with the same interactions with P
4       P' ← Children(P)                              ▷ list of children of P
5       for τ' ∈ Children(τ) do
6           DivideSeparator(τ', P')
```

ordered before τ_S. Therefore the interactions are all left interactions, i.e., we compute $\texttt{Reach}^-_{G/P}(i)$ for each $i \in \tau_S$, and group all unknowns i into one child cluster if they have the same structure of interactions. This top-down algorithm and partition is illustrated in Fig. 2.44a. The entire hierarchy is constructed from the top towards the bottom. This is the classical way to construct a hierarchy in the $\mathcal{H}$-arithmetic. Though, we may



(a) Top-down algorithm: descendants of separators are created following the interactions of the partition from the same level.

(b) Bottom-up algorithm: children are grouped together to form one parent with the same upper interactions.

Figure 2.44: Initial and following partition used on the cluster tree in the Hierarchical Separator Clustering method. The interactions of a cluster are composed of clusters from the same partition.

prefer to use a **bottom-up** algorithm to optimize computations. Indeed, merging lists of interactions together is far less costly than having to recompute the interactions of lower levels, as it means we have to recompute the list $\texttt{Reach}^-_{G/P}(i)$ of each unknown i to be able to group them into a cluster for each partition P. As mentioned in § 2.4.2, this would mean searching through the scalar edges of the adjacency graph G and look which endpoints are in the clusters in P. The general algorithm (without relaxation) is given in Algorithm 43. For the function defined in Algorithm 44 and used in the

Algorithm 43: Bottom-Up Interactions-Aware Hierarchical Separator Clustering.

```
Function BU-IA-HSC(τ)
1   if IsSparse(τ) then
2       for τ' ∈ Children(τ) do
3           BU-IA-HSC(τ')
4   else
5                                                  ▷ τ is a separator
6       P ← InitialPartition()                     ▷ Leaf clusters
7       ComputeInteractionsWithQuotientGraph(τ, P)
8       children ← SplitByInteractions(τ)
9       BottomUpSeparatorClustering(τ, children)
```

Algorithm 44: The bottom-up recursive clustering of a separator τ used in the Hierarchical Separator Clustering, based on the parent interactions of each set in S. Each cluster is assumed to be implemented as a non-ordered set of indices.

Function `BottomUpSeparatorClustering`(τ, S)

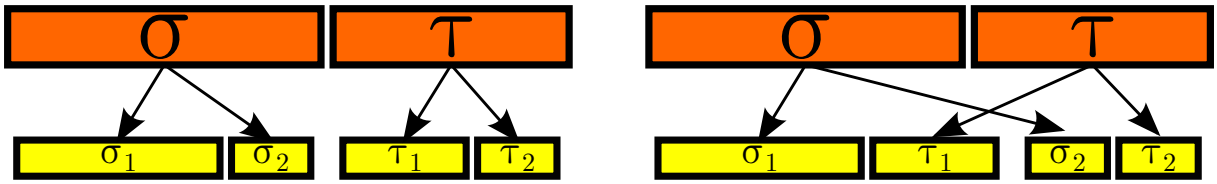
if $|S| < 2$ **then**
  `Children`$(\tau) \leftarrow S$ ▷ Complete hierarchy between τ and S
else
  `Parents` $\leftarrow \emptyset$
  while $S \neq \emptyset$ **do**
    $\sigma \leftarrow S[1]$ ▷ Iterate over S
    $R \leftarrow$ `Upper(Reach`$^-(\sigma)$`)` ▷ Upper interactions R
    `Set` $\leftarrow \{\rho \mid$ `Upper(Reach`$^-(\rho)) = R\}$ ▷ Clusters with the same parent
    $\sigma' \leftarrow \bigcup_{\rho \in Set} \rho$ ▷ Create parent with interactions R
    `Reach`$^-(\sigma') \leftarrow R$
    `Children`$(\sigma') \leftarrow$ `Set` ▷ Connect children with their parent
    $S \leftarrow S \backslash$`Set` ▷ Remove children from S
    `Parents` $\leftarrow$ `Parents` $\cup \sigma'$
  `BottomUpSeparatorClustering`$(\tau,$ `Parents`$)$

bottom-up Algorithm 43, we have an issue of implementation to consider. Once a cluster is created, its offset and size are usually fixed. All indices included in a cluster should match the indices of the cluster's children indices (the indices of one cluster correspond to the union of the indices of its children). A cluster's size is therefore the sum of the size of its children. If clusters are implemented as intervals of indices, i.e., an offset and the size of the cluster, we cannot rely solely on Algorithm 44. This algorithm makes the assumption that the clusters are implemented as sets of indices. This issue with intervals is depicted in Fig. 2.45. We assume here that clusters are ordered from left to right. The problem arises from the fact that we order the clusters from the (yellow) sub-level before their parents. These clusters are ordered following their interactions. Following the lexicographic or the shortlex order, we may have an order for a level (here σ_1, τ_1, σ_2, τ_2) that does not match the order of their parents (σ and τ). For example, let us assume that we have $a_1 < a_2 < a_3 < b_1 < b_2 < b_3$. We also define `Children`$(a) = (a_1, a_2, a_3)$ and `Children`$(b) = (b_1, b_2, b_3)$, so that $a < b$. If we have:

$$\begin{aligned} \texttt{Reach}(\sigma_1) = [a_1, a_2],\ \texttt{Reach}(\sigma_2) = [a_1, a_2, a_3], \\ \texttt{Reach}(\tau_1) = [b_1, b_2],\ \texttt{Reach}(\tau_2) = [b_1, b_2, b_3], \end{aligned} \tag{2.15}$$

the shortlex order leads to $\sigma_1 < \tau_1 < \sigma_2 < \tau_2$. However, we would have:

$$\texttt{Reach}(\sigma) = [a],\ \texttt{Reach}(\tau) = [b]. \tag{2.16}$$



(a) Clusters from the sub-level are ordered in the same order as their parents.

(b) Clusters from the sub-level are not aligned with their parents.

Figure 2.45: This representation assumes that clusters are ordered from left to right. Using intervals, the order of the yellow blocks in (b) is problematic. Indeed, σ_1 and σ_2 have the same parent σ but are not ordered one after another. The offset of σ should be that of σ_1 and its size the sum of σ_1 plus σ_2. However, with τ_1 in the middle, the interval is disrupted. In (a), the order of the yellow blocks follows the order of the upper (orange) partition and clusters may therefore be represented as intervals.

The shortlex order leads to $\sigma < \tau$. An interval could therefore not represent σ as the union of its children. We need more information to know which unknowns are associated with σ. Sets do not require the unknowns to be contiguous and could therefore be used to implement this algorithm. All of this relies on the assumption that we cannot reorder clusters once they have been created. If we abandon this principle, we may simply create temporary clusters of which we know only the size and not the offset. Once we have created a whole level, we simply create parent clusters of which the size is the sum of the sizes of its children. The parents list is sorted again by their interactions. This is described in Algorithm 45. Once the whole hierarchy has been constructed, a reordering step is performed by computing the offsets of each cluster, in a top-down fashion following the order of each list of children. Algorithm 46 describes the order of operations of this reordering. The indices are moved only when we reach a leaf, i.e., once the leaf cluster is at the right place. The function `MoveIndices(Offset(`τ'`),Size(`τ'`), offset)` moves all indices (and the corresponding reverse indices) from `Offset(`τ'`)` to `Offset(`τ'`)+Size(`τ'`)` towards their new location, which starts at `offset`. For example, let us assume the algorithm computed a hierarchy with the order of Fig. 2.45b, it will be able to move σ_2 from its original position in this example to its new and rightful position in Fig. 2.45a.

With this bottom-up or top-down method, each level of the subtree consists of clusters built according to their interactions with a partition of clusters, usually of the same level (see § 2.3.2 for an example of irregular recursion). An example of an $\mathcal{H}$-Matrix constructed using the top-down algorithm is depicted in Fig. 2.46. An example is given in Fig. 2.38. It benefits from both the hierarchy of an $\mathcal{H}$-Matrix inside separators *and* having a coherent clustering of separators based on the interactions of its unknowns, contrary to the IA-FSC method which would rely only on the quality of the domain division and lost the hierarchy characteristics of $\mathcal{H}$-Matrices inside separators. Moreover, numerous zeros are separated from the rest higher in the hierarchy, allowing us to avoid unnecessary hierarchy over zeros.

Algorithm 45: The bottom-up recursive clustering of a separator τ used in the Hierarchical Separator Clustering, based on the common parent interactions of elements in S. Clusters are assumed to be implemented as a continuous set of indices. Clusters have no offset following this algorithm. The offsets are therefore recomputed afterwards in a top-down fashion using their position in the tree (see Algorithm 46).

Function `BottomUpSeparatorClustering`(τ, S)

1 **if** $|S| < 2$ **then**
2 $\quad$ `Children`$(\tau) \leftarrow S$ ▷ Link hierarchy with τ
3 **else**
4 $\quad$ `Parents` $\leftarrow \emptyset$
5 $\quad$ **while** $S \neq \emptyset$ **do**
6 $\quad\quad$ $\sigma \leftarrow S[1]$ ▷ Iterate over S
7 $\quad\quad$ $R \leftarrow$ `Upper(Reach`$^-(\sigma)$`)` ▷ Upper interactions R
8 $\quad\quad$ `List` $\leftarrow \{\rho \mid$ `Upper(Reach`$^-(\rho)$`)` $= R\}$ ▷ Clusters with the same parent
9 $\quad\quad$ $\sigma' \leftarrow$ `Cluster`$(0, \sum_{\rho \in List}$ `(Size`(ρ)`))` ▷ Create parent with no offset
10 $\quad\quad$ `Reach`$^-(\sigma') \leftarrow R$
11 $\quad\quad$ `Children`$(\sigma') \leftarrow$ `List`
12 $\quad\quad$ ▷ Connect children with their parent
13 $\quad\quad$ $S \leftarrow S \backslash$`List`
14 $\quad\quad$ ▷ Remove children from S
15 $\quad\quad$ `Parents` $\leftarrow$ `Parents` $\cup \sigma'$
16 $\quad\quad$ `SortByInteractions(Parents)`
17 $\quad$ `BottomUpSeparatorClustering(`τ`, Parents)`

Algorithm 46: Reordering step. Recompute the offsets of the whole hierarchy under a separator τ and reorder the unknowns once we reach the leaves following this new offset.

Function `ReorderClusters`(τ)

1 `offset` $\leftarrow$ `Offset`(τ) ▷ We will update the offset of each child of τ
2 **for** $\tau' \in$ `Children`(τ) **do**
3 $\quad$ **if** `IsLeaf`(τ') **then**
4 $\quad\quad$ `MoveIndices(Offset`(τ')`,Size`(τ')`, offset)` ▷ Reorder indices following the new clustering
5 $\quad$ `Offset`$(\tau') \leftarrow$ `offset`
6 $\quad$ `offset` $\leftarrow$ `offset` $+$ `Size`(τ') ▷ Update the offset of the following child
7 $\quad$ `ReorderClusters`(τ')

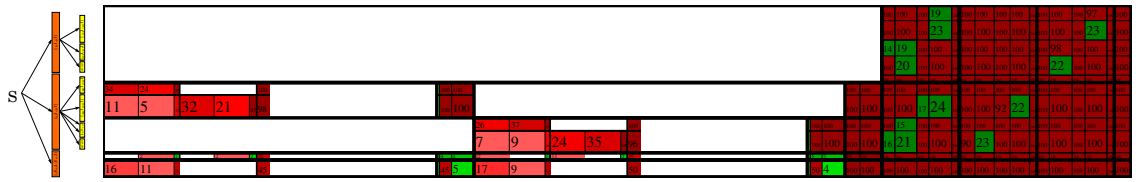
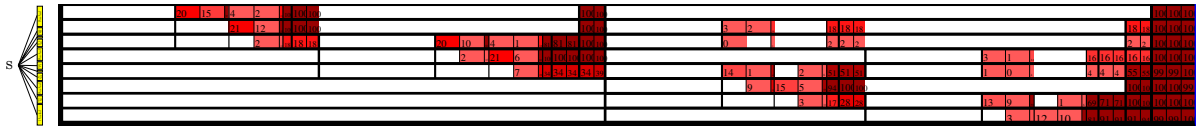
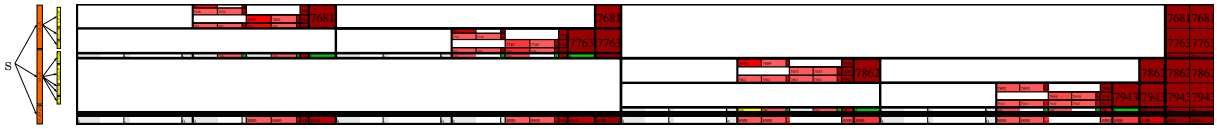


Figure 2.46: Separator clustering based Interactions-Aware Hierarchical Separator Clustering. Right (red) subpart of Fig. 2.37. Legend in Fig. 2.4.

Another advantage of the hierarchical clustering compared to the flat clustering is the ability to order the unknowns in an adequate pattern. Indeed, each clustering (in the Interactions-Aware flat or hierarchical method) orders unknowns using the lexicographic or shortlex order. We show the resulting structure of a subpart of a $\mathcal{H}$-Matrix in Fig. 2.47 following the separator clustering method. Using a lexicographic order, the unknowns



(a) IA-FSC with lexicographic order. The first non-zero blocks are ordered in a "staircase" shape.



(b) IA-HSC with shortlex order. The same staircase shape occurs at each iteration of the algorithm, but located on the subpart of the considered unknowns.

Figure 2.47: Left (yellow) subpart of Fig. 2.37 for IA-FSC (a) or IA-HSC (b).

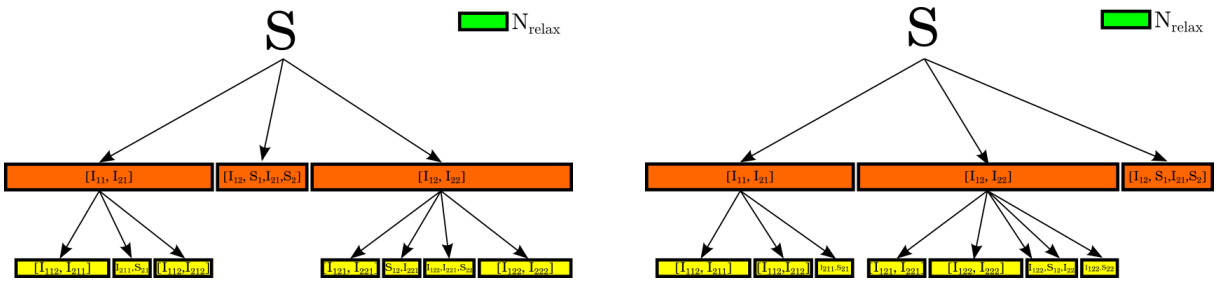
are well ordered within the first clusters of the global (column) clustering (the nested dissection partition). In Fig. 2.40b and 2.44, it corresponds to the left-most clusters of each partition. Therefore, the flat clustering will cluster the unknowns in an appropriate manner for the leaf clusters ordered first and will tend to ignore largest-ordered leaf clusters. The hierarchical variant will cluster the unknowns in an appropriate manner for each level of recursion. On the higher partition, it will divide the separator into a small list of clusters. Once this largest clustering is computed, each following division will benefit from this higher clustering and refine the clustering of the unknowns of one sub-cluster. However, each new clustering will also order the unknowns better for the first-ordered column clusters (the left-most clusters in our example figures). This also means that the largest-ordered clusters tend to be ignored in the lower clusterings. But this will have less impact due to the higher clusterings better taking the largest-ordered clusters into account.

Another way to "simply" explain this difference is to consider the ordering of a dictionary. A list of interactions `Reach(`i`)` can indeed be identified as a word. The problem of ordering these words is therefore related to the usage of the lexicographic or

the shortlex order. The lexicographic order leads to a good ordering of the first letters of each word. The shortlex order will have the same property but will also well separate words of different lengths. In the Interactions-Aware flat clustering, therefore, the flat clustering will have a nice incremental ordering of the first non-zero block in each separator. This may be identified as a staircase/triangular shape of the first rows of each separator in Fig. 2.47a. The lexicographic order has been used here to highlight this effect. In the Interactions-Aware hierarchical clustering, the same shape may be identified in a nested manner in Fig. 2.47b. In this example, a shortlex order is used, leading to the last rows being grouped together due to their largest number of interactions. These last rows correspond to the unknowns located in the center of the separator, usually connected to more clusters than other unknowns (after factorization). This will be discussed more at length in § 2.4.4.2, supported more specifically by Fig. 2.51e.

As mentioned earlier, these clusterings may be compared to the computation of T4 supernodes. In the same manner as we have previously discussed the relaxation of T2 supernodes in § 1.3.1.1.3, we now discuss the relaxation of clusters computed with Interactions-Aware methods.

In both the top-down and bottom-up algorithms, some computed clusters may be too small and have a negative impact on the efficiency of the solver. Therefore, we may have to resort to relaxation, such as depicted in Fig. 2.48. We consider the relaxation, or



(a) Unordered amalgamation. (b) Desired output with shortlex order.

Figure 2.48: Relaxation or amalgamation of Fig. 2.43c with the minimum size N_{relax} (width) highlighted in green.

amalgamation, of each partition independently (yellow and orange levels in the figure). We may amalgamate leaf clusters together quite easily. If a list of k clusters $(\tau_i)_{1<i<k}$ with children are to be amalgamated into a new cluster τ, we may either merge the clusters τ_i together and delete their children, or merge the different branches of the subtrees of each τ_i together into one branch under τ. We rely on the second technique in this work. In the case of a shortlex order (see p. 134), this relaxation, is computed starting from the end of the list of the ordered clusters. As long as a cluster τ is not sufficiently large (larger than a chosen minimum size N_{relax}), it is extended using the following cluster in the list, i.e., ordered before τ.

Definition 2.2. N_{relax} is the threshold used for relaxation using naive amalgamation of clusters.

The relaxation is performed starting from the end of the list. Small clusters indeed tend to have either longer lists (because they are at the intersection of multiple clusters) or lists containing separators and therefore ordered last (following Definition 1.6). Using a lexicographic order, only the small clusters interacting with separators are ordered last. The other clusters interact with non-separator subdomains and are therefore larger and ordered before. We do not want to amalgamate large clusters. To achieve this relaxation without introducing zeros in large clusters, we must amalgamate clusters only if the total size of the resulting cluster is under a specified threshold. The top-down algorithm can simply be stopped for a fixed limit size N_{relax} during the recursion.

However, in the bottom-up algorithm, we need to be able to choose an adequate initial partition P (in Algorithm 43) to avoid the computation of unnecessary lower levels. The same problem is discussed at the end of § 2.4.3.2. We would need to be able to relate the size of all the clusters computed by `SplitByInteractions`(τ, P) to the size of the clusters from the input partition P. In other words, if a partition P_i leads to a clustering of the separator into clusters of size smaller than N_{relax}, we may want to skip the computation of the lower partitions $P_j, j > i$. Yet, the computation of lower levels may have an impact on the reordering of the unknowns and therefore on the number of off-diagonal blocks. To use the same relaxation as the top-down algorithm, the simplest method may be to explore the cluster tree again, once the whole clustering has been computed, and detach from the tree all clusters (and their descendants) with a size smaller than N_{relax}. It should be noted that this relaxation does not change the reordering of the unknowns and therefore the bottom-up reordering of lower levels has an impact even though they are deleted. In particular, this feature may impact the local clustering discussed in § 2.4.3.4.

The algorithms discussed here may be put in perspective with the nested dissection method. Indeed, the hierarchical ordering of a separator τ usually leads to a good separation of two large sub-clusters with a sub-separator with a smaller dimension (it will be one-dimensional if the separator is two-dimensional). This sub-separator will match the projection of external separators.

The Interactions-Aware hierarchical separator clustering may also be compared to the methods discussed in § 2.1.3. More precisely the top-down algorithm shown in Fig. 2.43 may be compared to [53, Fig. 2.7]. However, while the strategy of [53] is to use this kind of clustering to eliminate a specific level of nested dissection, the approach discussed in § 2.4.3.3 also takes into account local orderings such as the Minimum Fill method. As mentioned in § 2.2.2, separator clusterings based on a similar idea have also been discussed in [164, chapter 5].

2.4.3.4 Interactions-Aware Local Separator Clustering (IA-LSC)

In an effort to further take advantage of the sparsity of the symbolic information computed by CV-HSF (see § 2.4.2.1.2), we discuss here the notion of a local row clustering specific to each leaf column cluster. This may also be seen as the continuation of the search for a sparse format discussed in § 2.3.3 and the search for a sparse admissibility condition (the non-separator admissibility condition) discussed in § 2.3.4. This is very different

from the IA-FSC and IA-HSC methods (§§ 2.4.3.2 and 2.4.3.3) as the granularity of the local row clustering may be very small without impacting the other column clusters, as it is not shared by the rest of the column clusters. However, the IA-LSC method is complementary of the IA-FSC and IA-HSC methods and may be combined with them. In this case, IA-FSC or IA-HSC is used at a coarse granularity whereas IA-LSC handles a finer granularity.

The symbolic information of $\texttt{Reach}_G^+(\tau)$ computed by CV-HSF may be used to subdivide the matrix block $M_{\sigma\times\tau}$ along the rows, if σ is ordered after τ, i.e., $\sigma > \tau$. The matrix is considered to have a symmetric symbolic structure. Therefore the symbolic information may be exploited in asymmetric matrices and the structure $\texttt{Reach}_G^+(\sigma)$ may be used to subdivide $M_{\sigma\times\tau}$ along the columns if $\sigma < \tau$. Here, this symbolic information is used only on leaf clusters, thus reproducing a format close to a sparse supernodal solver such as [119]. From the point of view of a sparse symmetric solver, this corresponds to using a different row partition for each block column independently from one another. However, we lose the clustering compatibility between block columns when using this method. Instead of implementing a new format for the solver to handle, we simply insert children in the leaves of the $\mathcal{H}$-Matrix based on the information listed in $\texttt{Reach}_G^+(\tau)$ or $\texttt{Reach}_G^+(\sigma)$. This is an extension of the hierarchical format: once an $\mathcal{H}$-Matrix has been constructed with other clustering methods, we may post-process this $\mathcal{H}$-Matrix using this IA-LSC technique to possibly reduce the storage of off-diagonal leaves. Algorithm 47 describes how the creation of an $\mathcal{H}$-Matrix may be adapted (lines 5,6) for this subdivision. Algorithm 48 creates a list of local row (or column) clusters based on the intervals of

Algorithm 47: Creation of an $\mathcal{H}$-Matrix with subdivision of leaves using Cluster-Vertex symbolic information. Adaptation of Algorithm 9, p. 37.

```
   Function CreateHMatrix(σ × τ)
1      if IsLeaf(σ × τ) then
2          if Admissible(σ, τ) then
3              Compress(σ × τ)                          ▷ Creation of a Rk-Matrix
4          else
5              if IsSparse(σ) ∨ IsSparse(τ) then
6                  LocalInteractionsDivision(σ × τ)
7              else
8                  CreateFullBlock(σ × τ)               ▷ Creation of a Full-Matrix
9      else
10         for (σ′, τ′) ∈ Children(σ × τ) do
11             if σ′ ∈ Reach(τ′) ∨ τ′ ∈ Reach(σ′) then
12                 CreateHMatrix(σ′, τ′)
```

$\texttt{Reach}_G^+(\tau)$ (or $\texttt{Reach}_G^+(\sigma)$). The blocks resulting from the interactions of these local row

Algorithm 48: Division of σ according to the symbolic information of τ (or division of τ using the symbolic information of σ) and creation of multiple dense blocks based on that division.

```
Function LocalInteractionsDivision(σ × τ)
    if σ > τ then
        Rows ← SplitByReach(σ, τ)                    ▷ Split σ using Reach_G^+(τ)
        for σ' ∈ Rows do
            Children(σ × τ) ← Children(σ × τ) ∪ (σ' × τ)
            CreateFullBlock(σ' × τ)
    else
        Cols ← SplitByReach(τ, σ)
        for τ' ∈ Cols do
            Children(σ × τ) ← Children(σ × τ) ∪ (σ × τ')
            CreateFullBlock(σ × τ')
```

clusters and τ (or σ and local column clusters) are inserted as children of the node $\sigma \times \tau$ in the $\mathcal{H}$-Matrix. The function `SplitByReach`(σ, τ) simply returns the intersection of the indices of σ and the structure $\texttt{Reach}_G^+(\tau)$ under the form of a list of clusters. An example of such a construction is shown in Fig. 2.49. This figure is to be compared with Fig. 2.38. One can see that the submatrices in Fig. 2.38 have been subdivided in Fig. 2.49 following the information computed by symbolic factorization.

This technique drastically changes the recursion of some operations. For example, the recursive GEMM operation must be rewritten in order to take into account the various potential situations that may arise, some of which are illustrated in Fig. 2.50. This leads to new hierarchical operations to be implemented in the hierarchical solver, such as the situation depicted in Fig. 2.50b. In order to compute a $\mathcal{H}$-GEMM on non-standard $\mathcal{H}$-Matrices A, B and C, we must recurse on all children of each row or column cluster of the involved matrices that will result in actual fill-in. For example, we need to find clusters for the row clusters of A and C that intersect one another. To simplify the reading of the algorithm, the matrices are assumed to be leaves in Algorithm 49. Also, the common dimension between A and B is assumed to be equal. However, the procedure

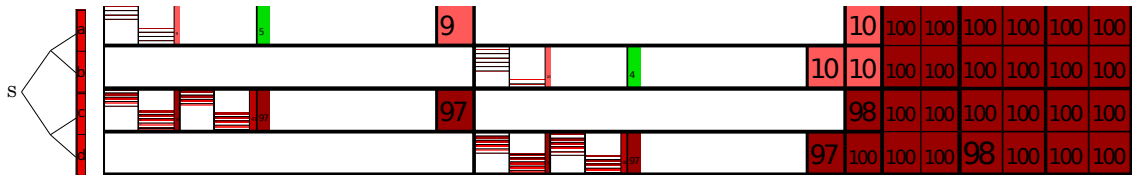
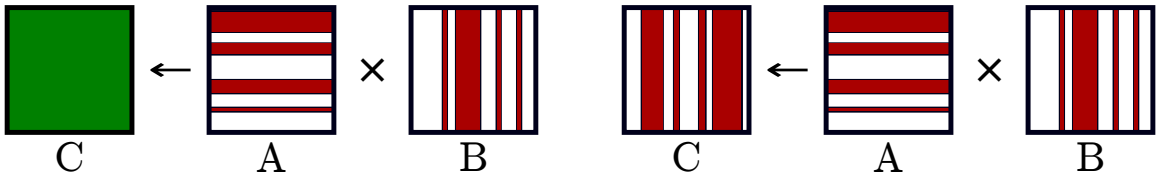


Figure 2.49: Interactions-Aware Local Separator Clustering used on non-separators off-diagonal submatrices. We can see that each column cluster has a different, independent row clustering. Right (red) subpart of Fig. 2.37. Legend in Fig. 2.4.



(a) The destination matrix C is compressed. The children of A and B are smaller than C, therefore we need to compute the subset of a compressed matrix. This is not a trivial operation if it must be performed efficiently.

(b) The destination matrix C does not have the same rows as A and the same columns as B.

Figure 2.50: Example of unusual operations arising from the subdivision of leaves using symbolic information.

may also be written to handle unusual cases where these conditions do not apply. These cases will not be addressed here. To also make the algorithm more readable, we rely on the function `rowChild`(A, i), which returns the i-th child of the row cluster of A. The function `Leaf-GEMM`(C, α, A, B, β) handles the other cases of multiplication (with at least one leaf among A, B or C), as discussed in § 1.2.3.3.3.

The function `UpdateIndices`(σ, τ, i, j) used at the end of the algorithm updates the indices for both loops. If the clusters intersect, we should indeed continue to use the one ordered first, in case it overlaps a second cluster of the other list of children. For example, if the row cluster of A has children equal to $[0, 5][5, 10]$ and the row cluster of C has children equal to $[0, 1][4, 7][15, 20]$, the first multiplication will operate on $\sigma = [0, 5]$ and $\tau = [0, 1]$. We must keep $\sigma = [0, 5]$ for the second loop and use the next $\tau = [4, 7]$ as they intersect and the associated multiplication will result in fill-in.

The ordering of the separator is closely related to the efficiency of this format. The new problematics arising from this local clustering are now closer to the question of the reduction of the number of off-diagonal blocks, such as discussed in § 2.2.2. We are now faced with two main categories of efficiency linked to the separator clustering: (1) the low-rank interactions between clusters, and (2) the minimization of the number of off-diagonal blocks. This topic has for example been discussed with both aspects in mind in [164]. The separator clusterings presented in this section are mainly designed for the first point. We have not investigated the second point in this thesis.

2.4.4 General Remarks

2.4.4.1 Order of Computations

The symbolic factorization is usually computed once the whole cluster tree has been constructed. Therefore, the separator clustering should be computed *before* the symbolic factorization. However, the interactions-aware separator clusterings heavily rely on the use of symbolic information and the quotient graph G/P. This means that the symbolic information of the interactions between clusters (before factorization) is computed twice.

Algorithm 49: Recursive GEMM operation multiplying matrices A and B and adding them to C. Adaptation of Algorithm 11, p. 40.

Function $\mathcal{H}$-GEMM(C, α, A, B, β)

```
1  if IsRoot(C) then
2      C ← βC
3  if ¬IsLeaf(σ × τ) ∧ ¬IsLeaf(σ × ρ) ∧ ¬IsLeaf(ρ × τ) then
4      i_A ← 1, i_C ← 1
5      while i_A ≤ nrRow(A) ∧ i_C ≤ nrRow(C) do
6          while rowChild(A, i_A) < rowChild(C, i_C) do
7              i_A ← i_A + 1
8              if i_A > nrRow(A) then
9                  return
10         while rowChild(C, i_C) < rowChild(A, i_A) do
11             i_C ← i_C + 1
12             if i_C > nrRow(C) then
13                 return
14         j_B ← 1, j_C ← 1
15         while j_B ≤ nrCol(B) ∧ j_C ≤ nrCol(C) do
16             while colChild(B, j_B) < colChild(C, j_C) do
17                 j_B ← j_B + 1
18                 if j_B > nrCol(B) then
19                     return
20             while colChild(C, j_C) < colChild(B, j_B) do
21                 j_C ← j_C + 1
22                 if j_C > nrCol(C) then
23                     return
24             k ← 1
25             while k ≤ nrCol(A) do
26                 if (rowChild(A, i_A) ∩ rowChild(C, i_C)) ∧ (colChild(B, j_B) ∩
                    colChild(C, j_C)) then
27                     H-GEMM(Child(C, i_C j_C), α, Child(A, i_A, k), Child(B, k, j_B), 1)
28             UpdateIndices(colChild(B, j_B), colChild(C, j_C), j_B, j_C)
29         UpdateIndices(rowChild(A, i_A), rowChild(C, i_C), i_A, i_C)
30 else
31     Leaf-GEMM(C, α, A, B, β)
```

Algorithm 50: Function to update indices i and j according to the mutual positions of σ and τ.

```
Function UpdateIndices(σ, τ, i, j)
1   if σ ≤ τ then
2       i ← i + 1
3   else
4       if σ ≥ τ then
5           j ← j + 1
6       else
7           i ← i + 1
8           j ← j + 1
```

Consequently, we may wonder if the separator clustering may be computed *after* the symbolic factorization. To do this, for each row index in the separators, we need a left-looking information of which column it interacts with. This is exactly the reverse information of the structure computed by CV-HSF. This method computes the structure $\texttt{Reach}_G^+(\tau)$ for each column cluster τ. This structure lists all unknowns connected to τ, i.e., all unknowns i for which $\tau \in \texttt{Reach}_{G/P}^-(i)$. This may be formulated as

$$\texttt{Reach}_G^+(\tau) = \bigcup_{\tau \in \texttt{Reach}_{G/P}^-(i)} i. \tag{2.17}$$

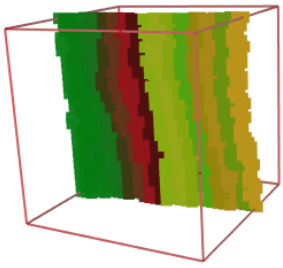
And vice versa,

$$\texttt{Reach}_{G/P}^-(\tau) = \bigcup_{i \in \texttt{Reach}_G^+(\tau) \wedge \tau \in P} \tau. \tag{2.18}$$

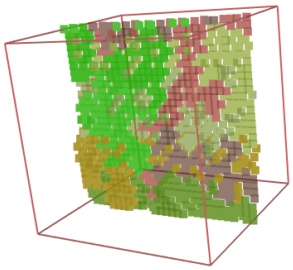
where the partition P may vary following the algorithm chosen for the separator clustering, i.e., IA-HSC or IA-FSC. In the case of a flat separator clustering, only one partition P is needed and the right-looking symbolic information $\texttt{Reach}_G^+(\tau)$ of $\tau \in P$ may be therefore stored in a unique container $\texttt{Reach}(\tau)$ and transformed into the left-looking structure $\texttt{Reach}_{G/P}^-(i)$. In the case of a hierarchical separator clustering, multiple partitions P_i will be involved. We would therefore need to store multiple pointers for each partition during the symbolic factorization before computing the separator clustering or mix the two algorithms to create the separator clustering while computing the symbolic factorization. This order of operations has not been investigated yet.

2.4.4.2 Examples of Separator Clustering

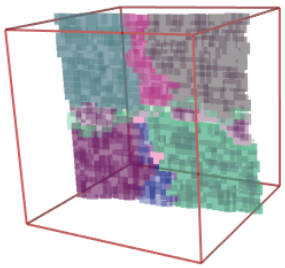
In the context of this thesis, we consider two variants of the nested dissection: a geometric-based nested dissection and a nested dissection based on SCOTCH. We can use a local ordering for the local subdomains of the nested dissection computed through



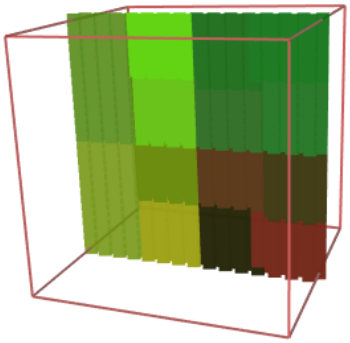
(a) IO-HSC strategy 's' used in Scotch ND.



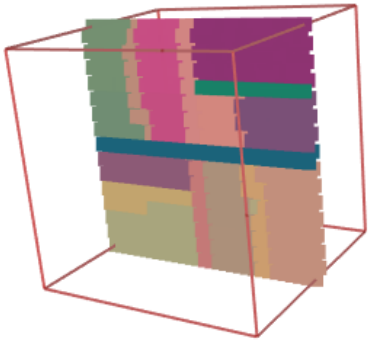
(b) IO-HSC strategy 'g' used in Scotch ND+.



(c) IA-HSC applied on a SCOTCH nested dissection.



(d) IO-HSC geometric recursive bisection used in BBox ND.



(e) IA-HSC applied on a geometric nested dissection (BBox ND).

Figure 2.51: Examples of various separator clusterings applied on the first separator of a nested dissection computed on the cube example of Fig. 1.17. Different colors indicate leaf sub-clusters inside this separator.

SCOTCH such as the AMF method, as discussed in § 2.3.1.3. Once the nested dissection has been computed, one of the previous clustering methods may be applied on each separator. Relying again on SCOTCH strategies, we can for example compute Fig. 2.51a or Fig. 2.51b. The first one is used here with a standalone nested dissection while the second one is applied in combination with the AMF method. We refer the reader to appendix A for more details on the strategies involved. Instead of relying on SCOTCH *local ordering* strategies, the IA-HSC variant may be used, of which an example is shown in Fig. 2.51c. The classical clustering used in the $\mathcal{H}$-Matrix community, the recursive bisection, can also be applied on the separators, as shown in Fig. 2.51d. Instead of applying this algorithm, we may also use the IA-HSC variant on the separators, resulting in Fig. 2.51e. In this last (geometric) example, we can see the isolated unknowns in the middle of the separator (forming one-dimensional clusters), corresponding to the interactions with other separators. This is also visible in the topological approach (Fig. 2.51c), though the shape of separators being not as regular as in the geometric nested dissection, it leads to less regular sub-clusters. we can see that, generally, the geometric nested dissection leads to more regular-shaped clusters, favorable to compression. Finally, the IA-HSC method seems to have only little differences from its IO-HSC counterpart in

the geometric approach while there are more differences between IA-HSC and IO-HSC methods in the topological approach.

2.5 Solution of the FEM/BEM Coupling using $\mathcal{H}$-Matrices

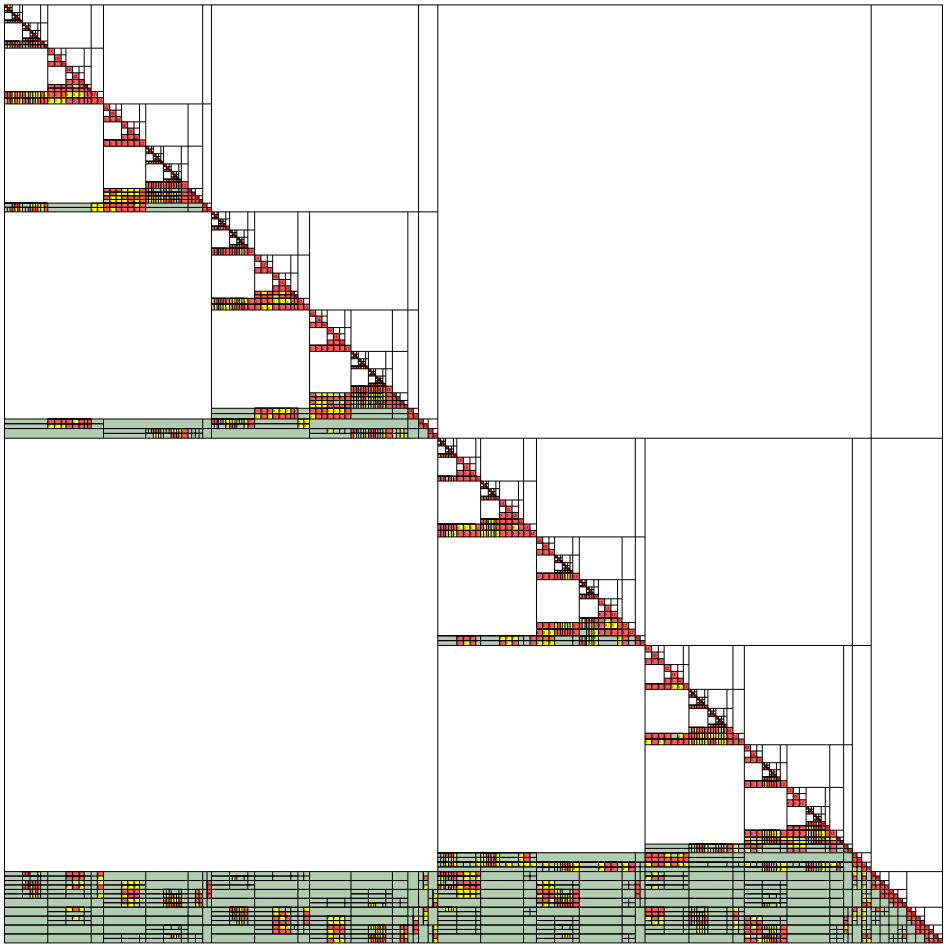
For the more general question of this thesis, we discuss here the solution of a FEM/BEM coupling (Eq. (1.2)) using the elements discussed in this chapter.

First of all, the nested dissection may be applied on the volume mesh and the recursive bisection on the surface mesh. This leads us to the overall system presented in Fig. 2.52. In this symmetric linear system, the unknowns associated with the volume mesh impact the matrix A_{vv} (Fig. 2.52a) and the matrix A_{sv} (Fig. 2.52b). While we have already discussed tall & skinny blocks in § 2.3.2, it must be noted that this problem is also present in the blocks of the matrix A_{sv}, due in part to the fact that the surface (row) mesh has less unknowns than the volume (column) mesh. However, this problem was already present in the original recursive bisection method (see Fig. 1.46b). But we may observe new rectangular (tall & skinny) blocks: the blocks arising from the interaction between separators from the volume mesh and regular clusters from the surface mesh (for example, the blocks at the far right of the matrix A_{sv}). The first (and higher) separator is here smaller than the surface mesh, creating a reversed imbalance in the hierarchy (the rectangular blocks have a longer dimension in the surface dimension). In the case of a nested dissection, the approach discussed in § 2.3.2 to prevent tall & skinny blocks may be applied on Fig. 2.52b to produce Fig. 2.53b. The discussion may be transposed to A_{vs} for asymmetric linear systems. In fact, while all the modifications introduced in this chapter have been discussed for a square sparse matrix $A \in \mathbb{C}^{n \times n}$, they may also be translated to the off-diagonal rectangle sparse matrices A_{sv} and A_{vs}.

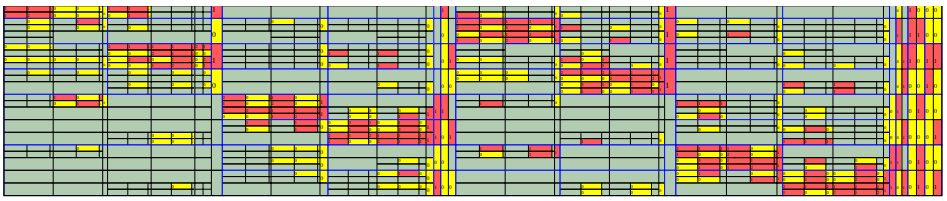
The symbolic factorization may also be used in this context but the subsystem on which it is performed may be discussed. In particular, following the discussion of § 1.3.1.2.5, a symbolic factorization may be applied on either:

1. the sparse matrix A_{vv}, the FEM-FEM submatrix;
2. the sparse matrix $\begin{bmatrix} A_{vv} \\ A_{sv} \end{bmatrix}$, with the off-diagonal BEM-FEM submatrix;
3. the overall matrix A of the FEM/BEM coupling.

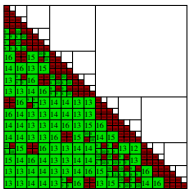
To avoid unnecessary computation, the symbolic factorization should not be computed on the whole system (case n°3), as the symbolic information is useless on the dense matrix A_{ss}. An example of the use of a symbolic factorization for the first case (on A_{vv}) is displayed in Fig. 2.53. Usual recursive bisection is used as a separator clustering in this example. Regarding case n°2, following the discussion in § 1.3.1.2.5, we may apply a symbolic factorization on a rectangular system by using Algorithm 20. However, for implementation consideration, this has not been investigated yet.



(a) A_{vv}.

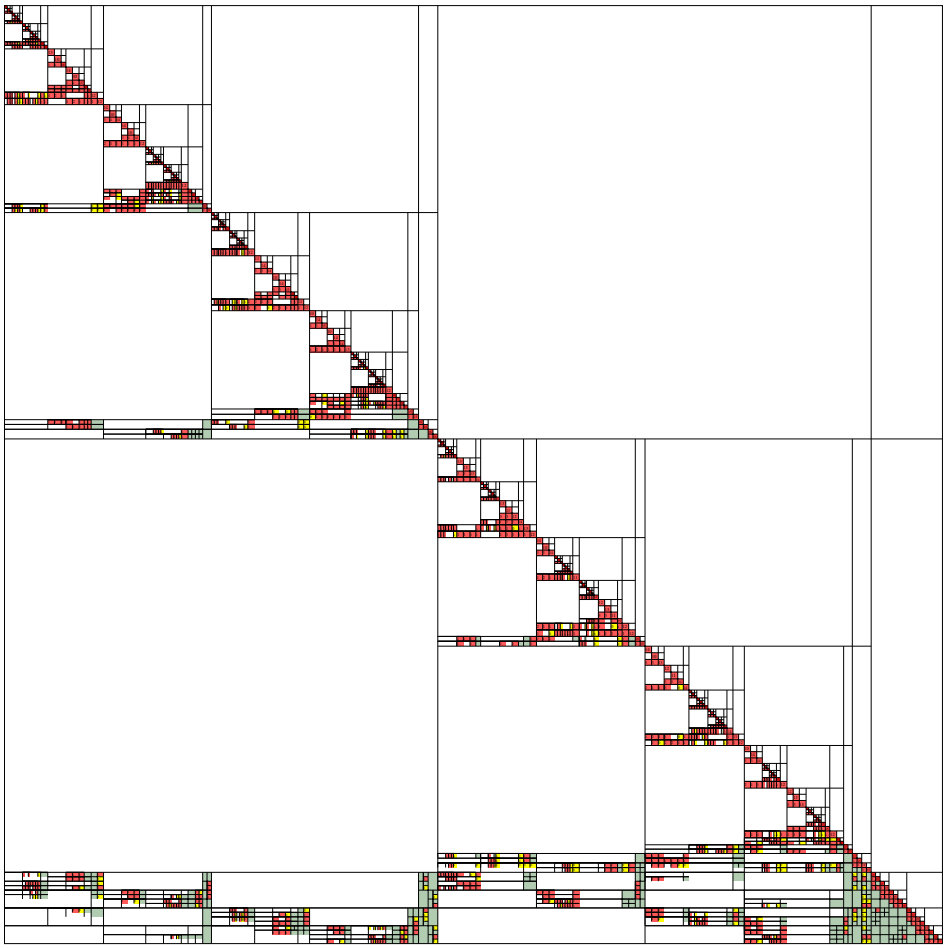


(b) A_{sv}.

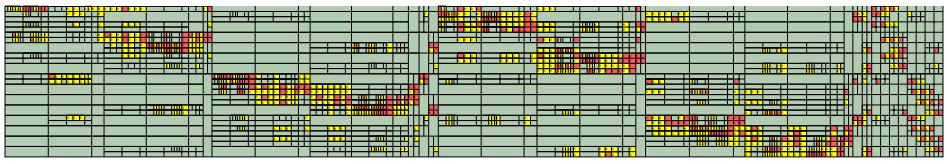


(c) A_{ss}.

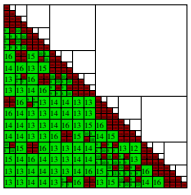
Figure 2.52: $\mathcal{H}$-Matrices involved in the solution of a symmetric linear system arising from a FEM/BEM coupling using nested dissection.



(a) A_{vv}.



(b) A_{sv}.



(c) A_{ss}.

Figure 2.53: $\mathcal{H}$-Matrices involved in the solution of a symmetric linear system arising from a FEM/BEM coupling using nested dissection and symbolic factorization on A_{vv}.

Chapter 3

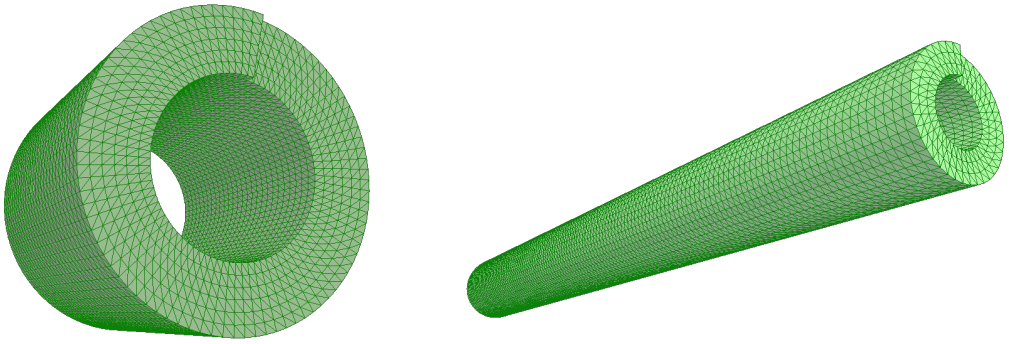
Numerical Experiments

We have presented in Chapter 2 a set of methods based on the combination of hierarchical techniques with sparse techniques. The use of nested dissection (§ 2.3.1) and symbolic factorization (§ 2.4) in a hierarchical framework were discussed for the solution of sparse matrices. In this chapter, we consider these algorithms and study their computational efficiency and their impact on the performance of the hierarchical solver. We consider only one admissibility condition (the strong criterion) in this study and leave for future work the search of other criteria such as discussed in § 2.3.4 (shown to greatly impact the low-rank compression and thus the performance of the solver). We also rely on the algebraic modification of the construction of the block cluster tree of a $\mathcal{H}$-Matrix to prevent tall & skinny blocks, such as discussed in § 2.3.2.

For the purpose of the study we intend to lead here, we detail the test cases chosen for our experiments in § 3.1. The experimental environment (hardware configuration, solvers parameters) is detailed in § 3.2. The efficiencies of the proposed methods for the clustering and analysis (symbolic factorization) steps are discussed in §§ 3.3 and 3.4, respectively. The numerical factorization of a sparse matrix is then studied in § 3.5. We first examine the effect of clustering and symbolic factorization on the numerical factorization in a hierarchical solver. We study the behavior of the CC-HSF and CV-HSF methods compared to methods not relying on symbolic factorization, as well as the behavior of the Interactions-Aware separator clusterings compared to the Interactions-Oblivious separator clusterings. We then establish a comparison between these hierarchical methods and a reference sparse direct solver (here, MUMPS). We study the full-rank and the low-rank versions of the $\mathcal{H}$-Matrix solver and the MUMPS solver. Eventually, a study of the FEM/BEM coupling is presented in § 3.6.

3.1 Test Cases

The test cases for these experiments have been chosen as a pipe such as displayed in Fig. 3.1, in an effort to coincide with the realistic application presented in Fig. 1.4 while providing reproducible examples for the community. The dimensions of a pipe may be changed by setting its radius (R) and its length (L) to different values. Studies have been



(a) Short pipe (L/R = 2). (b) Long pipe (L/R = 20).

Figure 3.1: Examples of pipes with different lengths (L). The radius (R) is set to 2 m.

led on short pipes (with a ratio $L/R = 2$) or long pipes ($L/R = 40$). The latter test case would be more realistic for an aeroacoustic problem such as the one considered here (Fig. 1.4b).

The unknowns of the matrix associated with the problem are located on the vertices of the mesh. In the FEM discretization, the unknowns interact with their immediate neighbors (connected by one edge). The BEM is applied on the outer surface of the pipe. The unknowns associated with the BEM discretization rely on the same mesh as the FEM. However, their interaction is based on the wavelength λ varying with respect to the number of unknowns of the problem. We set λ such that there will be 10 points per wavelength. For two points i and j, with $k = 2\pi/\lambda$ and r the distance between these points, the interaction a_{ij} between these two unknowns is given by Eq. (3.1).

$$a_{ij} = \frac{e^{ikr}}{r}. \tag{3.1}$$

For studies on sparse matrices (§§ 3.3 to 3.5), the short pipe test case has been used. Test cases vary from one million (1.10^6) complex unknowns and doubled up to 16 million (16.10^6) for sequential numerical factorizations. The tests on pure symbolic factorization (without computing the numerical factorization) may be performed on larger cases than for the numerical factorization, they have been therefore studied for test cases doubled up to 64 millions unknowns in sequential. If no numerical factorization is computed, the result may not be validated via a forward or backward error. The hierarchical symbolic factorization is then validated using a scalar symbolic factorization. Above 16 million unknowns, we have also studied the parallel numerical factorization of matrices by steps of 16 million up to 128 million unknowns.

For studies on the FEM/BEM coupling (§ 3.6), the long pipe test case has been used. The dimensions of the study cases for these preliminary experiments are described more precisely in § 3.6.

3.2 Experimental Environment

All solvers have been compiled with Intel compiler and linked with the Intel® Math Kernel Library (MKL) library for processing dense linear algebra (LAPACK/BLAS) operations. The version of Intel compiler and libraries used here is 17.0.4 (20170411). We also rely on the sequential version of SCOTCH 5.1.11. Table 3.1 attempts to succinctly describe the specificities of each considered approach. We consider for example multiple *global*

Considered Approach	Partitioning/Clustering						Compression				Hierarchical Format	SF
	Global			Local								
	ND	Geometric	Topological	Subdomain Strategy	Separator Strategy	Separator Clustering	∅	Algebraic	BBox	Non Sep.		
$\mathcal{H}$-Bisection [34, 39, 40, 47, 48, 112, 114]	∅	✓	✓	-	-	-	✓	✓	✓	-	$\mathcal{H}$	∅
Supernodal solver (PaStiX) [119]	✓	∅	✓	AMF	RB,g	Local	✓	✓	∅	∅	BLR	✓
Multifrontal solver (MUMPS) [157]	✓	∅	✓	s	RB,s	Local	✓	✓	∅	∅	BLR	✓
$\mathcal{H}$-ND [102, 107, 126, 138, 143] (§ 2.3.1)	✓	✓	✓	∅	RB	Shared	✓	✓	✓	✓	$\mathcal{H}$	∅
CC-HSF (§ 2.4.2.1.1)	✓	✓	✓	AMF,∅	*	*	✓	∅	✓	✓	$\mathcal{H}$	✓
CV-HSF (§ 2.4.2.1.2)	✓	✓	✓	AMF,∅	*	*	✓	∅	✓	✓	$\mathcal{H}$	✓
IO-HSC (§ 2.4.3.1)	✓	✓	✓	AMF,∅	RB,s,g	Shared	✓	∅	✓	✓	$\mathcal{H}$	✓
IA-FSC (§ 2.4.3.2)	✓	✓	✓	AMF,∅	IA	Shared	✓	∅	✓	✓	$\mathcal{H}$ \{S}	✓
IA-HSC (§ 2.4.3.3)	✓	✓	✓	AMF,∅	IA	Shared	✓	∅	✓	✓	$\mathcal{H}$	✓
IA-LSC (§ 2.4.3.4)	✓	✓	✓	AMF,∅	*	Local	✓	∅	✓	✓	$\mathcal{H}$	✓

Table 3.1: List of approaches and their respective relevant characteristics. − means the characteristic is irrelevant for this method. ∗ means any of the parameters may be applied. ∅ means that the parameter is not used/is ignored. RB: Recursive Bisection (or K-way). ND: Nested Dissection. Local/Shared: Separator Clustering is either specific to each block column or shared between all. AMF: Approximate Minimum Fill. 'g', 's': SCOTCH strategies (see appendix A). IA: Interactions-Aware. $\mathcal{H}$ \{S} means the structure is hierarchical except in separators. SF: Symbolic Factorization.

partitioning (or clustering), either relying on recursive bisection (first line) or nested dissection (the rest). The clustering may rely on geometric or topological information, such as discussed in § 2.3.1. The *local* partitioning/clustering of subdomains may rely on AMF for some of the considered approaches. To cluster the separator, multiple strategies may be used, among the Interactions-Oblivious (geometric recursive bisection or SCOTCH strategies) and Interactions-Aware methods. The separator clustering may be *shared* by

all column clusters/supernodes or specific (*local*) to one column cluster. The compression techniques that may be used include: $\varnothing$, when no compression is used; an algebraic approach, where information from the matrix is used to decide whether to compress or not; an admissibility condition based on bounding boxes; an admissibility condition based on the Non-Separator criterion, discussed in § 2.3.4. The hierarchical format column lists methods based on the $\mathcal{H}$-Matrix format or the BLR format. One of the method is listed as $\mathcal{H} \setminus \{S\}$ to emphasize the fact that no hierarchy is used in separators. The SF column indicated which method rely on symbolic factorization. The methods listed in the last rows of this table (CC-HSF, CV-HSF, ..., IA-LSC) may be applied on the $\mathcal{H}$-ND method.

The memory consumption of the solver has been studied through the hardware information measured in the file `/proc/self/status`. The real memory usage of the solver, i.e., the "resident set size", the memory held in the Random-Access Memory (RAM), corresponds to the field `VmRSS` and its peak to `VmHWM`. The virtual memory usage corresponds to the field `VmSize` and its peak to `VmPeak`. We here usually display the real and virtual memory peaks. Memory profiles (displaying the memory usage of the solver all along the execution of the program) show the real memory.

3.2.1 Hardware Configuration

Experiments have been run on machines from both GENCI [2] and PlaFRIM [6] computing facilities. Table 3.2 details the characteristics of the machines.

Facility	Machine	Processor	Nb Cores	Frequency	RAM
GENCI	`occigen`	Intel® Xeon® E5-2690 v3	$24 = 2 \times 12$	2.6 GHz	128GB
PlaFRIM	`miriel`	Intel® Xeon® E5-2680 v3	$24 = 2 \times 12$	2.5 GHz	128GB
PlaFRIM	`brise`	Intel® Xeon® E7-8890 v4	$96 = 4 \times 24$	2.2 GHz	1TB
PlaFRIM	`souris`	Intel® Xeon® E5-4620 v2	$96 = 12 \times 8$	2.6 GHz	3TB

Table 3.2: Machine specifications.

For each set of experiments, the name of the machine used will be mentioned. The machine `occigen` is a cluster composed of nodes with 64GB of RAM by default, or optionally 128GB if the user requires so. There are two physical sockets (Haswell Intel Xeon processors) on these nodes (two NUMA nodes per socket), each socket containing 12 cores for a total of 24 cores running at a frequency of 2.6 GHz. The second type of machines used, `miriel`, is part of the PlaFRIM cluster. These nodes have 128GB of RAM and also two (Haswell Intel Xeon) sockets (two NUMA nodes per socket) containing 12 cores each for a total of 24 cores running at a frequency of 2.5 GHz. The `brise` machine is a node part of the PlaFRIM cluster with 1TB of RAM and four (Broadwell Intel Xeon) sockets (two NUMA nodes per socket) containing 24 cores each for a total of 96 cores running at a frequency of 2.2 GHz. Finally, `souris` is a node part of the PlaFRIM cluster and has 3TB of RAM as well as 12 (Ivy Bridge EP) sockets (one NUMA node per socket) containing 8 cores each for a total of 96 cores running at a frequency of 2.6 GHz. Note

that this last machine also has the option of activating hyper-threading though we have not used this feature in this thesis.

3.2.2 $\mathcal{H}$-Matrix Configuration

This subsection gives some information on the parameters and environment used in our experiments in the $\mathcal{H}$-Matrix Airbus solver. This solver is referred to as HMAT when there is no ambiguity on which method is used. Table 3.3 lists the possible values for each parameter considered in this set of experiments. For instance, we rely on the ACA+

Parameter	Possible values
Symbolic Factorization	{Left, Right} × {Bottom-Up, Top-Down} × {Cluster-Cluster, Cluster-Vertex, Vertex-Vertex (Scalar)} × {True, False}
Separator Clustering	{Interactions-Aware, Interactions-Oblivious} × {Hierarchical, Flat}
Local Separator Clustering	{True, False}
Admissibility Condition	{Bounding Box, Non-separator, Optimal, Topological, No compression}
Tall & Skinny (§ 2.3.2)	{Algebraic, Geometric} × {Cluster tree, Block cluster tree}
Compression Method	{SVD, ACA Full, ACA Partial, ACA+}
Re-Compression Method	SVD
Compression Threshold ϵ	$[10^{-16}, 1[$
# Unknowns (N)	$\mathbb{N}$
N_{ND} (Definition 2.1)	$\mathbb{N}$
N_{leaf} (Definition 1.3)	$\mathbb{N}$
N_{relax} (Definition 2.2)	$\mathbb{N}$

Table 3.3: Possible values for hierarchical method parameters. Methods in red are not studied here.

variant for compression and do not study the use of SVD or ACA with full or partial pivoting. Table 3.4 describes the steps studied in this chapter that may need some clarification.

The matrices involved in our experiments are complex symmetric, therefore the LL^T decomposition is used for their factorization. The solver does not rely on pivoting. The compression method (§ 1.2.3.1.1) is set to ACA+. The accuracy parameter of the solver is usually set to $\epsilon = 10^4$ when using compression, though it will be specified for each experiment. No iterative refinement is used here, though the use of $\mathcal{H}$-Matrices as preconditioners may be the object of future research.

For the numerical factorization step, the IA-LSC feature is always activated. Indeed, in the cases when no symbolic factorization is computed or when CC-HSF is used, IA-LSC simply does lead to the same hierarchical format as without IA-LSC. Therefore, the subdivision of leaves occurs only when CV-HSF is computed. To distinguish the

Step	Description
Clustering	Computation of the cluster tree, mainly relying on domain division such as nested dissection, recursive bisection or SCOTCH (§ 1.2.3.2.1 and § 2.3.1)
Scalar Adjacency	Computation of the structure $\texttt{Adj}_G(i)$ for each $i \in V$, effectively building the graph $G = (V, E)$ (§ 1.3.1.1.1)
Quotient Adjacency	Computation of the structure $\texttt{Adj}_{G/P}(K)$ for each $K \in P$, effectively building the graph G/P (§ 1.3.1.1.3)
Fill-in	Computation of the fillin, i.e., the * operation (§ 1.3.1.1)
Transmission to Hierarchy	Transmission of the symbolic information to all clusters in the hierarchy, for bottom-up algorithms (§ 2.4.2.2.2)
SF	Symbolic factorization (§ 1.3.1.2)
Schur	Computation of the Schur complement $\mathcal{S}$ (Eq. (1.30))
FEM elimination	Elimination of the volume unknowns, prior to the factorization of $\mathcal{S}$ in the final solution in Eq. (1.29)

Table 3.4: Description of the routines studied in this chapter.

differences between the various symbolic factorizations, we therefore put the *No SF* (for "No Symbolic Factorization") algorithm in perspective with CC-HSF and CV-HSF.

In the experiments contained in this chapter, we limit the blocking sizes to the same values $N_{ND} = N_{leaf} = N_{relax} = 100$. In preliminary experiments, we have observed that this size leads to an ideal trade-off between the storage and the time required to compute a factorization due to low-rank compression. Yet, other blocking sizes could be studied to further investigate their effect on each of the methods presented here.

In the case of the $\mathcal{H}$-Matrix solver, the number of floating-point operations (Flop) displayed in these experiments are counted manually through the number of BLAS operations performed on leaves of the $\mathcal{H}$-Matrix.

3.2.3 MUMPS Configuration

As we intend to establish a comparison with a reference sparse direct solver, we give compilation and parametrization details on the MUMPS sparse direct solver used in our experiments. MUMPS [22] has been run with version 5.1.2 (2017/11/2), with or without BLR [157]. According to the documentation, the BLR method used in this release follows the FSCU strategy (see § 2.2.1). The low-rank (BLR) variant is expected to have better results in terms of factorization time but not in memory consumption nor the solve time in this version, compared to the full-rank solver. The research work [157] enabling memory compression has indeed only been released on 2019/04/18 while finalizing the submitted version of this manuscript. The strategy used by MUMPS for SCOTCH is also noted Scotch ND+ in the experiments. It is slightly different (see Algorithm A.3, p. 206), mainly affecting the local ordering, but results in a very similar number of non-zeros. As

OMP_NUM_THREADS	MKL_NUM_THREADS	Factorization Time (s)
1	24	177.929
24	1	104.116
24	24	78.803

Table 3.5: Factorization time of a 1M-unknowns short pipe (Fig. 3.1a) for multiple configurations on 24 cores on `miriel` using BLR-MUMPS.

the considered matrices are complex symmetric, the factorization method used in MUMPS is the LDL^T decomposition by default. For multi-threading computations, the software is compiled with the flags `-openmp` and `-DBLR_MT` (following suggestions in §5.15 of MUMPS 5.1.2 user manual, and the advice of the MUMPS development team). To run on 24 cores, due to the results displayed in Table 3.5, we have chosen to set both `OMP_NUM_THREADS` and `MKL_NUM_THREADS` to 24 in our experiments.

In some experiments, we have displayed the real memory peak along with the virtual memory peak due to the optimizations performed by MUMPS. Indeed, MUMPS first performs an allocation larger than necessary in order for the computations to be more efficient. This translates into a larger virtual memory consumption though the real memory consumption of the solver is lower. This explains the behavior of the solver on larger cases where the virtual memory may exceed the RAM available on the machine but the real memory usage does not. When using MUMPS, the number of Flop are computed internally by the solver. We rely on the field RINFOG(3) for counting Flop in the information structure of MUMPS after factorization. In the case of BLR-MUMPS, the field RINFOG(14) is used, following the manual instruction and preliminary feedback of the MUMPS development team.

3.3 Clustering Methods

We first establish the efficiency of clustering (see Table 3.4) methods, later used as a reference for the variants of symbolic factorization studied in § 3.4. The methods compared here are the BBox ND 2, the Scotch ND and Scotch ND+ methods, detailed in § 2.3.1. The cost of computing a nested dissection using geometric (BBox ND 2 or BBox ND) or topological (Scotch ND) information is approximately similar as shown in Fig. 3.2. However, our strategy Scotch ND+ computing a local ordering (AMF here) takes twice as much time as the simple nested dissection.

3.4 Hierarchical Symbolic Factorizations

We now intend to compare the variants of the hierarchical symbolic factorization discussed in the previous chapter to be able to set one variant for the rest of the experiments (on the numerical factorization and on the FEM/BEM coupling). Our parameter space for the computation of an efficient hierarchical symbolic factorization has many dimensions. We

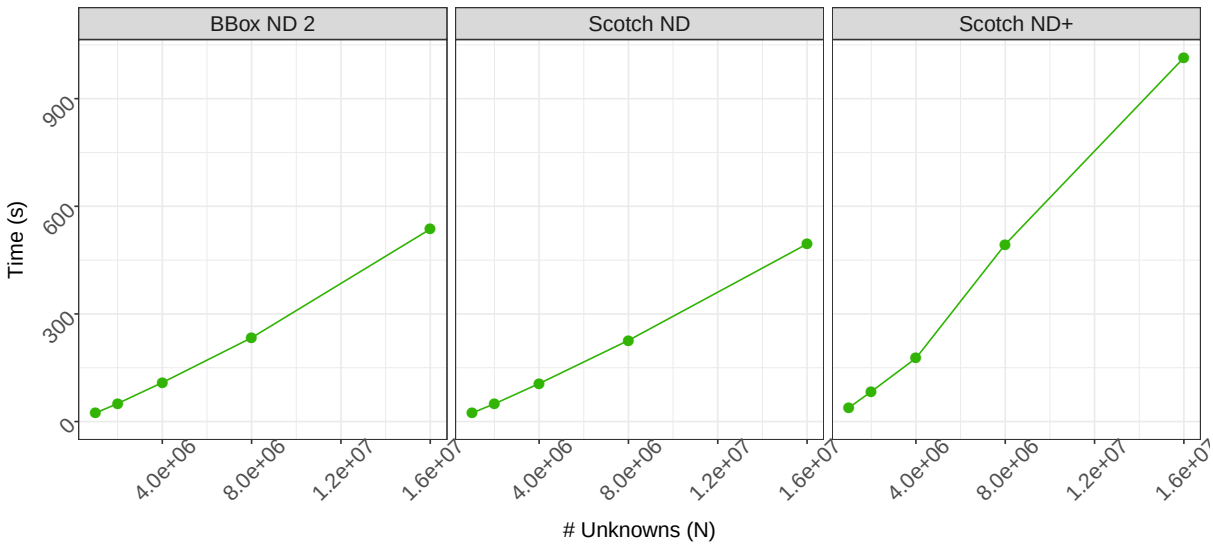
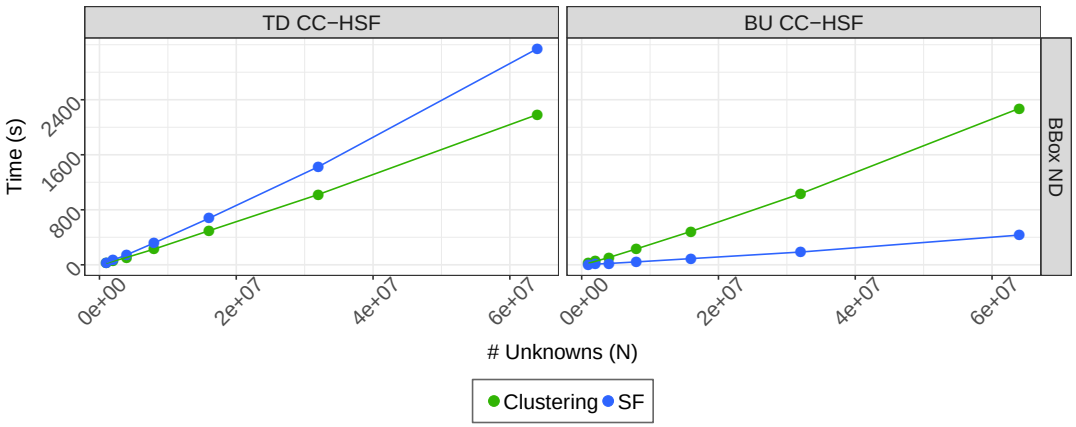


Figure 3.2: Time consumption of clustering methods. The addition of a local ordering (here AMF) doubles the time of the overall clustering in this example. Short pipe test case (Fig. 3.1a) from 1M to 16M unknowns. Sequential run on `occigen`.

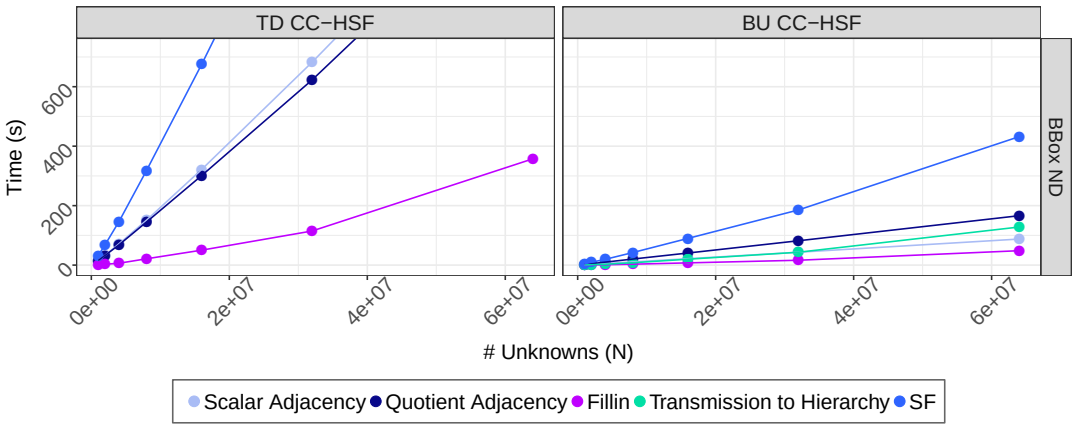
investigate here each dimension independently. In the following sections, we investigate the impact of computing a top-down hierarchical symbolic factorization compared to a bottom-up version of the algorithm (§ 3.4.1), the cost of using a left-looking algorithm compared to a right-looking variant (§ 3.4.2), and finally the differences between the Cluster-Cluster Hierarchical Symbolic Factorization, or CC-HSF, and the Cluster-Vertex Hierarchical Symbolic Factorization, or CV-HSF (§ 3.4.3). The comparisons are detailed here for test cases of short pipes (Fig. 3.1a) from 1 to 64 million unknowns, displayed as the x-axis of each graph. We also provide memory and time details of the 64M-unknowns problem in Table 3.6 at p. 167, in which some differences are more noticeable. In this table, both the Scotch ND+ and BBox ND are displayed. The symbolic factorization is not really affected by the clustering method (as we can see in Table 3.6). Therefore the discussion in this section is restricted to the BBox ND, though it may be extended to Scotch ND+.

3.4.1 Top-Down and Bottom-Up Hierarchical Symbolic Factorizations

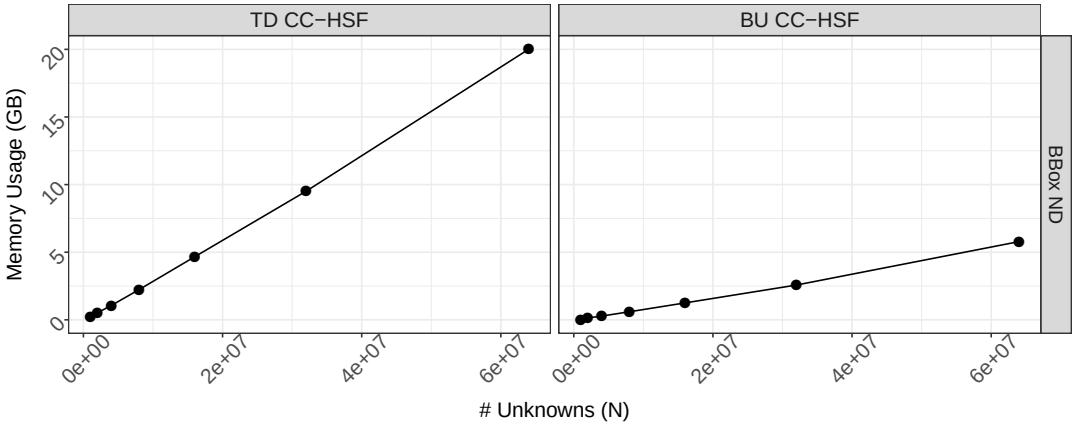
First, we lead a study on the differences between the top-down and the bottom-up variants of the hierarchical symbolic factorization, discussed in § 2.4.2.2. Both variants are here based on a right-looking variant computing cluster-cluster information (CC-HSF). The top-down hierarchical symbolic factorization (TD CC-HSF) is computed using Algorithm 33, p. 123. It computes the elimination of a quotient graph on each level of the cluster tree. The bottom-up hierarchical symbolic factorization (BU CC-HSF)



(a) Comparison of the duration of the clustering step and the symbolic factorization.



(b) Detail of the symbolic factorization step (see Table 3.4). The total computation time SF is the sum of all other steps.



(c) Memory consumption of the symbolic factorization.

Figure 3.3: Comparison of Top-Down vs Bottom-Up Hierarchical Symbolic Factorizations using Cluster-Cluster right-looking information (CC-HSF). Short pipe test case (Fig. 3.1a). Sequential run on `miriel`.

is computed using Algorithm 37, p. 125. It computes one elimination of the quotient graph associated with the leaves of the cluster tree and propagates the result to the upper clusters. On Fig. 3.3a, we can see that the cost of computing a top-down hierarchical symbolic factorization is larger than the cost of computation of the clustering step. By using a bottom-up algorithm, we are able to drastically reduce the cost of computing such a hierarchical symbolic factorization so that it takes only a fraction of the clustering step.

Details of the symbolic factorization step (see Table 3.4 for a description of these steps) are displayed in Fig. 3.4b. In addition, Table 3.6 gives the memory necessary for performing the symbolic factorization as well as the detailed times of each step for a 64M-unknowns problem. The top-down algorithm re-computes here at each level the scalar adjacency graph of the problem and transforms it into a quotient graph, thus explaining the major cost of these two particular steps in the algorithm. It should be noted that it is possible to store the scalar information in order to speed-up the algorithm with an extra cost of memory. The computation of the elimination quotient graph would then become the most time-consuming step of the algorithm. For each layer P_i, this symbolic elimination is not computed on the scalar graph and transformed into a quotient graph (G^*/P_i) but on each quotient graph instead $((G/P_i)^*)$. The cost of such a computation is already a large part of the cost of the overall symbolic factorization, and is larger than the cost of the whole bottom-up algorithm. To reduce this cost, one may think of computing the symbolic elimination only on the lowest quotient graph G/P_N. However, this leads us to essentially the same algorithm as the bottom-up algorithm, except that, for the computation of the symbolic information of an upper level P_i, we do not use the symbolic information of the nearest lower level P_{i+1} but instead use the lowest level P_N. The symbolic information on level P_N is by definition a much more precise symbolic information than that of the level P_{i+1} but this extra information is not useful for the level P_i and only leads to a larger complexity. Therefore, even with this possible improvement, the top-down algorithm would remain more time-consuming than the bottom-up algorithm.

Finally, Fig. 3.3c shows the difference of memory consumption of the solver before and after the symbolic factorization. We can see here that the memory usage of the bottom-up algorithm is also smaller than the memory usage of the top-down algorithm. Theoretically, the two algorithms should be able to consume the same amount of memory, at least in terms of the size of the computed symbolic information. Yet, the difference in memory consumption may be explained by our implementation: the top-down algorithm may store temporary data that will accumulate until the bottom of the tree is reached, while this temporary data may be freed in the bottom-up algorithm. We have not investigate this further, as this memory may be freed once the symbolic factorization has been computed and therefore should not impact the peak reached during the numerical factorization, studied in § 3.5.

Due to the results of this section, we focus on the bottom-up algorithm in the remaining of this thesis.

3.4.2 Left and Right-Looking (Bottom-Up) Hierarchical Symbolic Factorizations

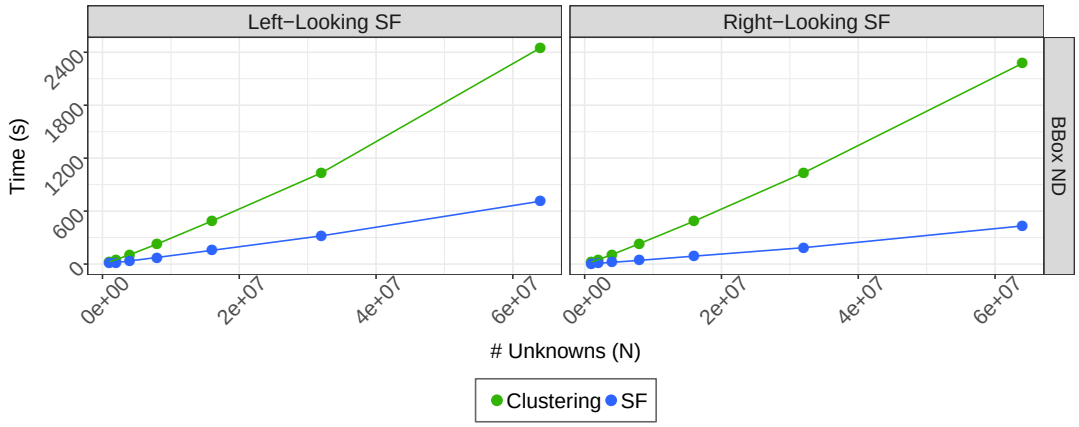
We now study the right-looking and left-looking variants of the hierarchical symbolic factorization, i.e., the computation of the right-looking $\texttt{Reach}^{+}_{G/P}(\tau)$ and left-looking $\texttt{Reach}^{-}_{G/P}(\tau)$ structures defined in Eq. (2.12) and Eq. (2.13), p. 118, respectively. Both variants rely on Algorithm 37, p. 125, in which we may simply compute either a right-looking or left-looking symbolic factorization on the leaves of the cluster tree. We can observe that both the right-looking and left-looking variants are faster than the clustering step according to Fig. 3.4a and seem to follow a linear complexity. However, the left-looking variant is slower than the right-looking version of the algorithm. The left-looking variant relies on Algorithm 19, p. 68, for the symbolic elimination of the leaves: the right-looking information is first computed to reduce the arithmetic complexity of the overall algorithm. The algorithm computes both right-looking and left-looking information (scalar and quotient adjacency steps are therefore multiplied by a factor of approximately two) and is inevitably slower than the algorithm that computes only right-looking information. The right-looking variant relies on Algorithm 15, p. 63, for this leaf symbolic factorization. This may be observed in Fig. 3.4b. Moreover, due to the storage of this extra right-looking structure, the left-looking algorithm consequently consumes more memory as shown in Fig. 3.4c. As mentioned earlier, these results are also displayed in Table 3.6. The pure left-looking algorithm (with no usage of right-looking data) presented in Algorithm 17, p. 67, was not used as the time consumption of this method increases too rapidly.

One advantage of a left-looking algorithm is that the left-looking information from the scalar edges of the adjacency graph may be used in the separator clustering algorithms presented in § 2.4.3 and studied in § 3.4.4. Indeed, these edges are the basis used for the clustering of each separator, and using this left-looking scalar information towards clusters would therefore avoid duplications of the computation of this information, as discussed in § 2.4.4.1.

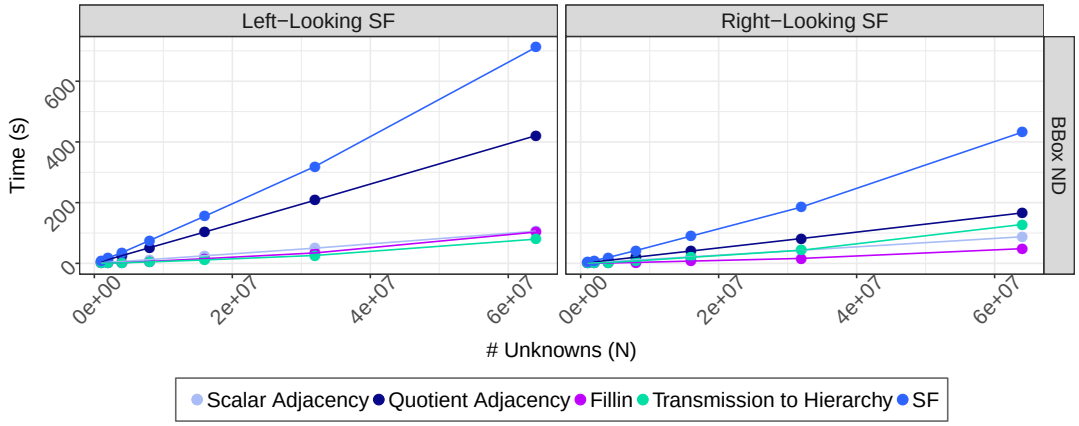
For the rest of this manuscript, we restrict the discussion to bottom-up algorithms using a right-looking variant due to the results of this section.

3.4.3 Cluster-Cluster and Cluster-Vertex Hierarchical Symbolic Factorizations

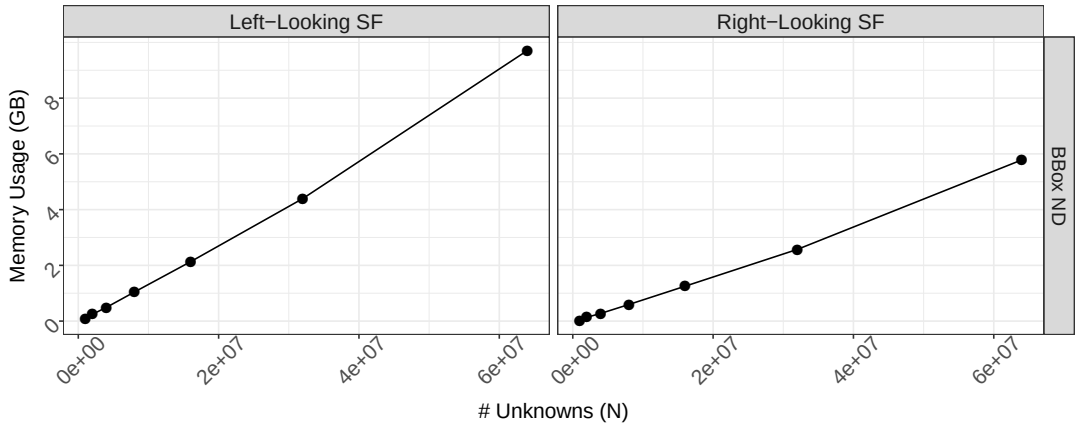
We now study the variants of hierarchical symbolic factorization computing either a Cluster-Vertex or a Cluster-Cluster symbolic information. The Cluster-Vertex approach is very similar to the Cluster-Cluster approach in terms of time consumption, as shown in Fig. 3.5a. An interesting difference between the two methods can be observed in Fig. 3.5b: the Cluster-Cluster algorithm takes more time computing quotient adjacency of clusters but spends less time in the transmission of the symbolic information to the upper levels, while the Cluster-Vertex variant follows the reverse trend. This observation may be confirmed by the results in Table 3.6 for a 64M-unknowns problem. In this



(a) Comparison of the duration of the clustering step and the symbolic factorization.

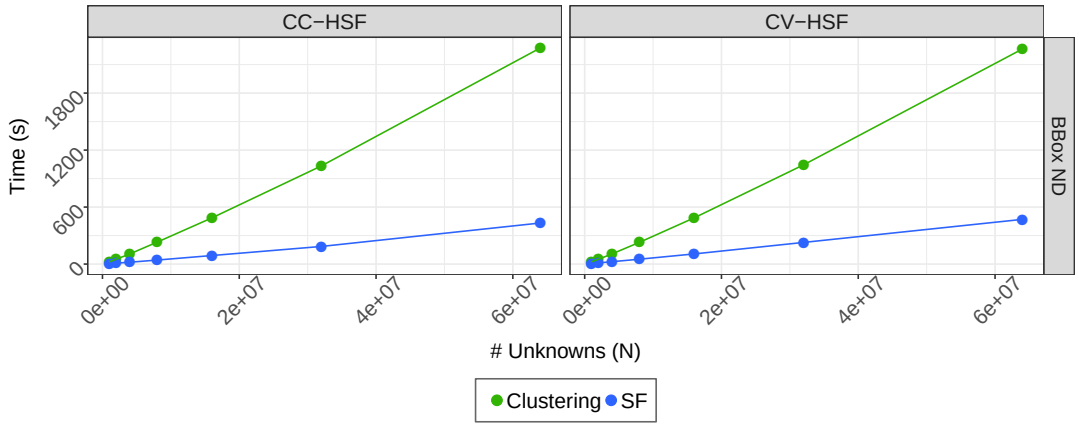


(b) Detail of the symbolic factorization step. The total computation time SF is the sum of all other steps.

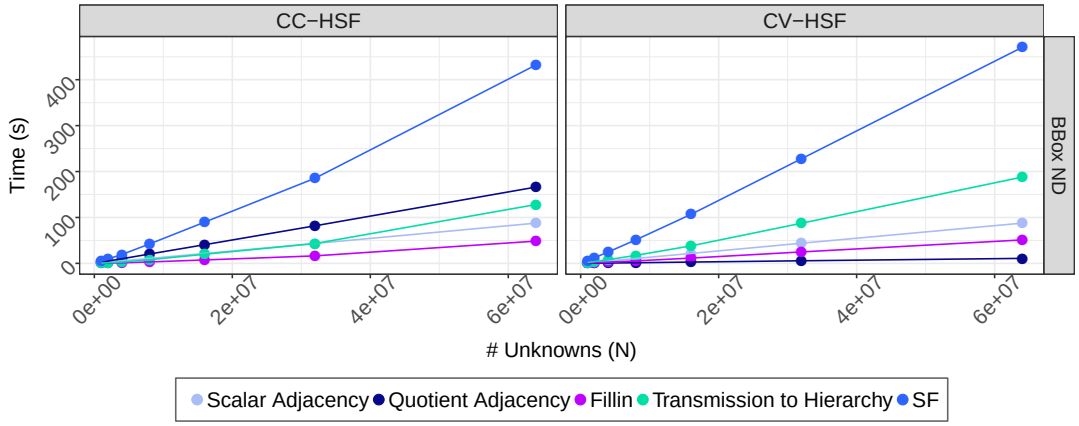


(c) Memory consumption of the symbolic factorization.

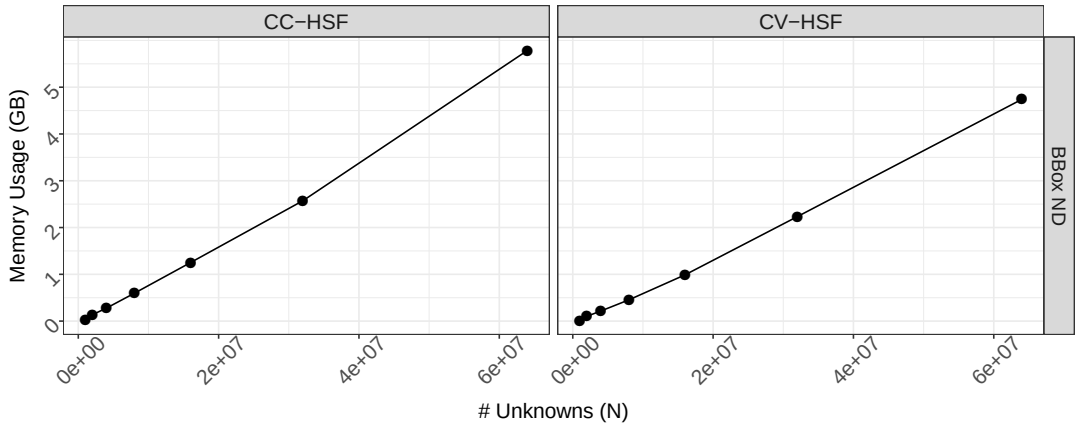
Figure 3.4: Comparison of Left-Looking vs Right-Looking (Bottom-Up) Hierarchical Symbolic Factorizations. Short pipe test case (Fig. 3.1a). Sequential run on `miriel`.



(a) Comparison of the duration of the clustering step and the symbolic factorization.



(b) Detail of the symbolic factorization step. The total computation time SF is the sum of all other steps.



(c) Memory consumption of the symbolic factorization.

Figure 3.5: Comparison of Cluster-Cluster vs Cluster-Vertex Hierarchical Symbolic Factorizations. Short pipe test case (Fig. 3.1a). Sequential run on `miriel`.

Method	Left/Right	Clustering	Memory	SF	Scalar Adjacency	Quotient Adjacency	Fillin	Transmission to Hierarchy
CV-HSF	Right-Looking SF	Scotch ND+	$3.48 \cdot 10^9$	429.27	95.42	9.42	43.36	138.33
BU CC-HSF	Right-Looking SF	Scotch ND+	$5.69 \cdot 10^9$	434.14	95.65	187.35	40.46	107.66
TD CC-HSF	Right-Looking SF	Scotch ND+	$1.84 \cdot 10^{10}$	3,566.90	1,582.76	1,563.28	398.28	–
BU CC-HSF	Left-Looking SF	Scotch ND+	$9.62 \cdot 10^9$	710.50	103.53	461.59	86.08	56.80
CV-HSF	Right-Looking SF	BBox ND	$4.74 \cdot 10^9$	470.50	87.62	10.85	51.04	187.94
BU CC-HSF	Right-Looking SF	BBox ND	$5.78 \cdot 10^9$	432.05	87.64	166.14	48.44	128.08
TD CC-HSF	Right-Looking SF	BBox ND	$2.00 \cdot 10^{10}$	3,141.16	1,442.00	1,327.81	357.43	–
BU CC-HSF	Left-Looking SF	BBox ND	$9.71 \cdot 10^9$	711.59	105.92	420.66	103.20	79.82

Table 3.6: Memory (Bytes) and time (s) required for the computation of symbolic factorizations and detailed time of each step. Short pipe (Fig. 3.1a) with 64M unknowns. Sequential runs on `miriel`.

example, the Cluster-Vertex approach computes the quotient adjacency in less than 11 seconds while the Cluster-Cluster approach takes more than 166. But the Cluster-Vertex approach takes more than 187 seconds to transmit the symbolic information to the hierarchy while it takes only around 128 seconds in the Cluster-Cluster approach. This difference comes from how these steps are computed. In the BU-CC-HSF approach, the scalar information is first transformed from a list of indices into a list of pointers towards clusters containing these points. This list is then sorted and duplicates are removed for later efficiency of searches in this list. The `Quotient Adjacency` step may therefore be faster but would later result in a slower computation of fill-in and a slower creation of the $\mathcal{H}$-Matrix that uses this information to decide to store a block or not (see Algorithm 30). Finally, the Cluster-Vertex algorithm appears to consume a little bit less memory than the Cluster-Cluster algorithm. As mentioned before, this excess of memory used by the symbolic factorization may be released once the $\mathcal{H}$-Matrix has been constructed and therefore should not impact the peak of memory reached during numerical factorization.

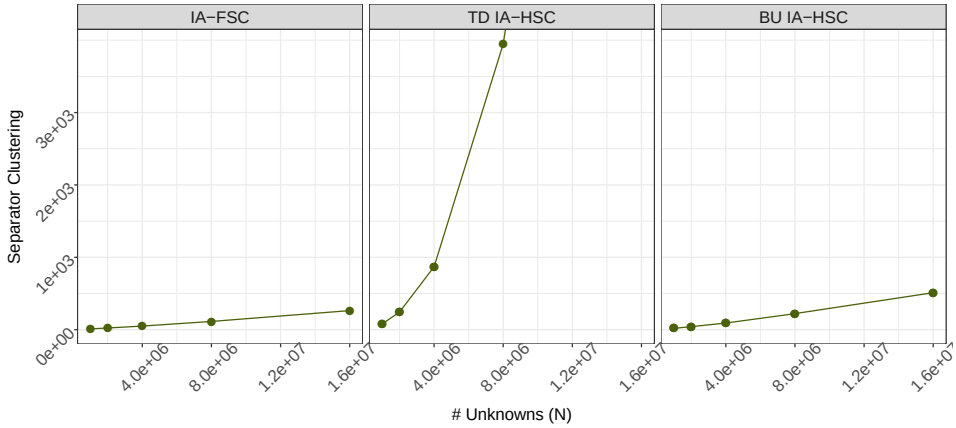
3.4.4 Separator Clustering

We now focus on the efficiency of the computation of the Interactions-Aware separator clustering methods. The top-down separator clustering approach does not scale for large matrices, as can be observed in Fig. 3.6a. Furthermore, the bottom-up hierarchical algorithm has a quite large overhead compared to the flat algorithm as one can see in Fig. 3.6b. In this figure, the methods are compared to the Scotch ND+ clustering and the CV-HSF time consumption, taken as a reference. Both methods require less time than the clustering step while requiring more time than the symbolic factorization. This is also noticeable in Fig. 3.6c, which shows the memory profile of the pre-processing computations before the actual creation of the $\mathcal{H}$-Matrix. The creation of the test case (the pipe) is not included. We can also see in this figure that both the flat and the hierarchical separator clustering have a similar memory usage.

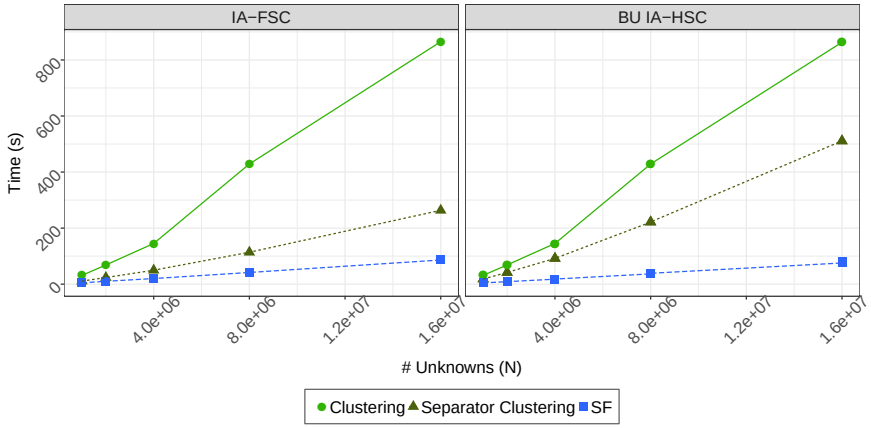
In conclusion of this section, we have compared the cost of pre-processing analysis methods developed in this thesis and established that:

- bottom-up and right-looking algorithms are faster;
- the CC-HSF and CV-HSF methods have a similar cost;
- the IA-FSC method is a bit faster than its counterpart (BU-)IA-HSC;
- efficient symbolic analysis and separator clustering cost less than the clustering step.

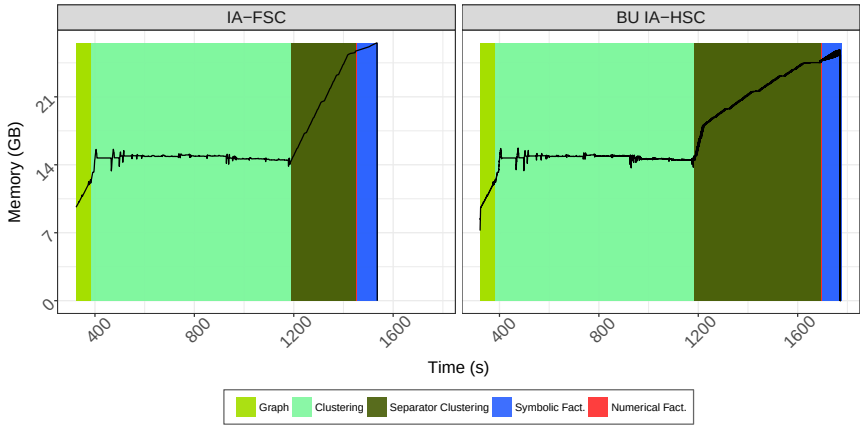
Now that we have concluded which analysis methods should be used, we must study their effect on the numerical factorization, as discussed in the following section.



(a) Separator clustering time (s).



(b) Comparison between the Flat and the Hierarchical algorithms and reference to the original clustering step and symbolic factorization.



(c) Memory profile including separator clustering.

Figure 3.6: Performance of separator clustering (using relaxation) SCOTCH nested dissection coupled with AMF. Short pipe test case (Fig. 3.1a). Sequential run on `occigen`.

3.5 Symbolic Factorization Influence over Numerical Factorization

For this set of experiments, we rely on the most efficient symbolic factorizations (in the analysis step) and compare their impact on the numerical factorization. Therefore, the symbolic methods selected for this comparison are CC-HSF and CV-HSF, with a bottom-up right-looking implementation. The separator clustering is set as IO-HSC (recursive bisection in the case of BBox ND, and strategies 's' or 'g' in Scotch ND and Scotch ND+ variants, respectively), IA-FSC and BU-IA-HSC (abbreviated IA-HSC).

3.5.1 Sequential Study of Middle-Range Test Cases

In order to validate the usage of a symbolic factorization, we must prove that:

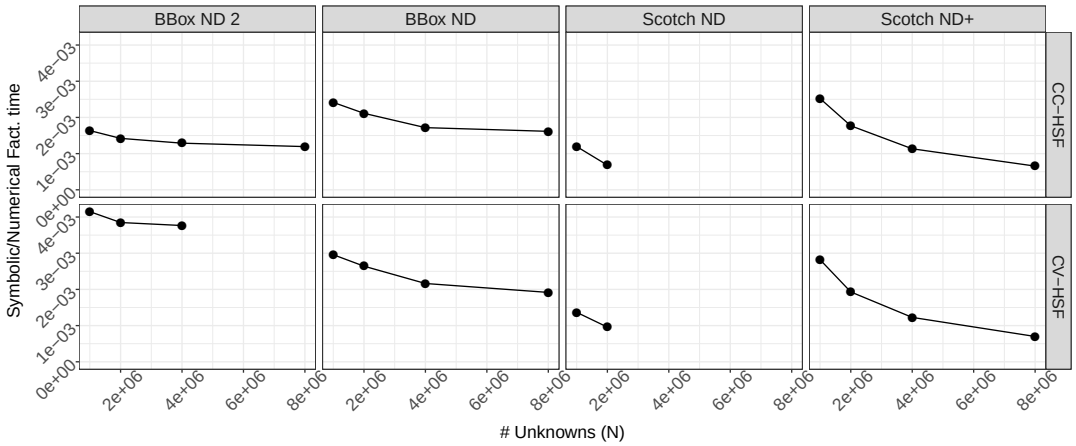
1. the cost of its computation is low and therefore does not have a large impact on the overall computation,
2. it benefits the memory and time consumption of the numerical factorization, so that the overall memory and time consumption of the solver should be reduced,
3. the accuracy of the solution is not changed.

Regarding the first point, the symbolic factorization step computed here takes only a small fraction of the computation with respect to the numerical factorization step. This fraction also gets smaller and smaller when the size of the problem grows. This is shown in Fig. 3.7a.

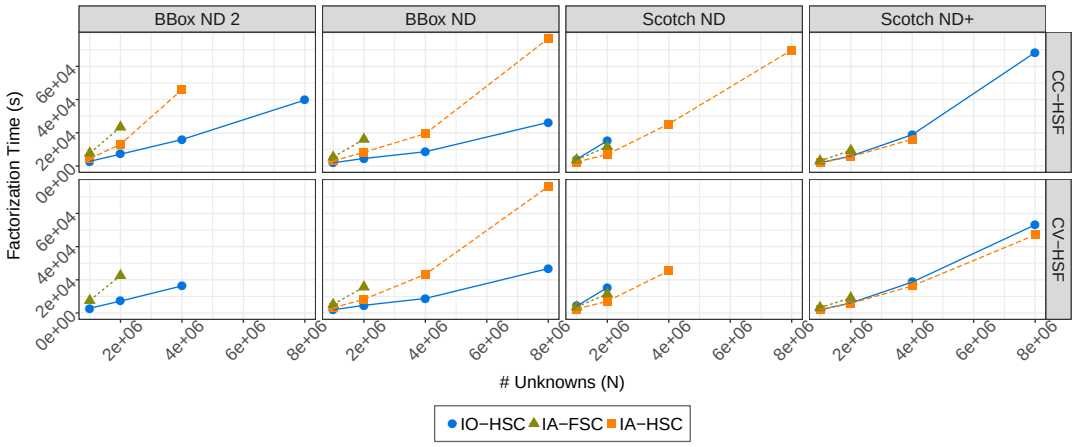
For the memory and time consumption, we study the impact of the symbolic factorizations, the separator clusterings and the global clusterings in Fig. 3.7. Fig. 3.7b displays the time required for the computation of the numerical factorization. Fig. 3.7c displays the memory consumption of the factorized matrix. The facet columns indicate the global clustering methods used, but also the local clusterings. As discussed in § 2.4.4.2, the Scotch ND separator clustering (IO-HSC) relies on SCOTCH 's' strategy. The Scotch ND+ separator clustering (also IO-HSC) relies on SCOTCH 'g' strategy, meant to reduce the number of off-diagonal blocks. The points that are not present in these results have either run out-of-memory or reached a wall-time (they would be out of the frame of these graphs). The Interactions-Aware methods used for the clustering of separators rely on relaxation to avoid too small supernodes. If no relaxation is used, the factorization time has been observed to drastically increase. Therefore, if the method would theoretically reduce the number of zero storage with no relaxation, we investigate here its practical efficiency for a relaxation $N_{relax} = N_{sparse} = 100$.

The first observation we can make is that, with relaxation, the IA-FSC method does not perform well both in terms of memory and time consumption compared to the other methods.

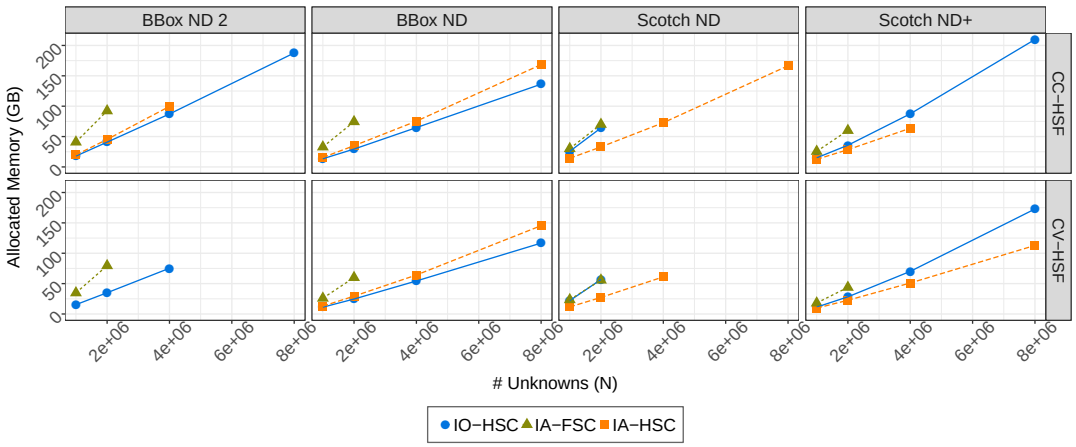
The second observation is that the performance of IO-HSC and IA-HSC depend greatly on the global clustering used. For example, in the geometric nested dissections (the first two columns), the IO-HSC method performs better than the IA-HSC method, both in terms of time consumption (Fig. 3.7b) and in terms of memory consumption (Fig. 3.7c).



(a) Ratio between the time consumption of the symbolic and numerical factorization.



(b) Numerical factorization time.



(c) Factorized matrix memory consumption.

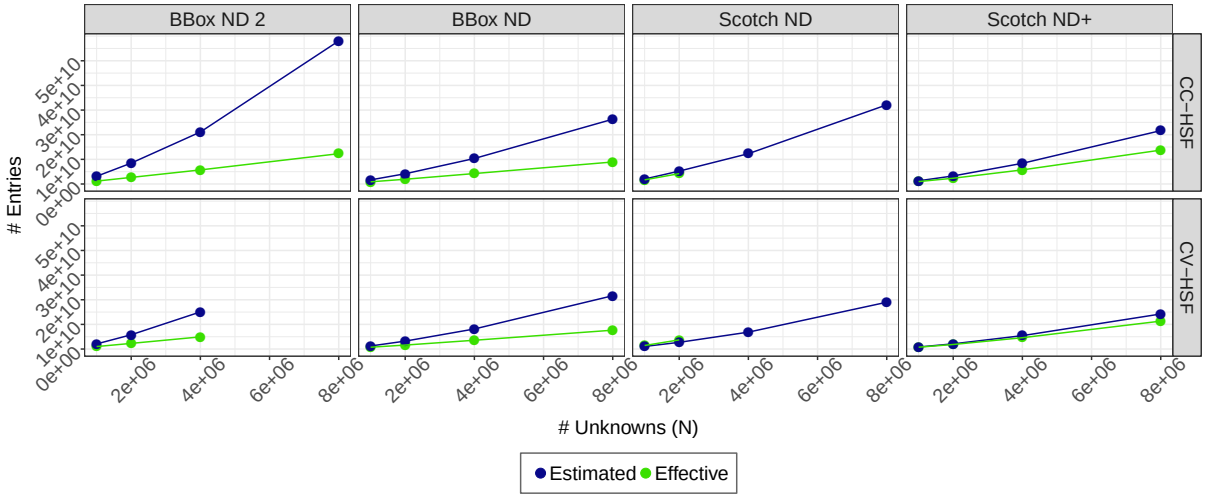
Figure 3.7: Impact of each method on numerical factorization. The X-axis indicates the number of unknowns. Separator clusterings (using relaxation) are indicated by the colors. Symbolic factorizations are displayed on facet rows. Global clusterings are displayed on facet columns. Short pipe test case (Fig. 3.1a). Sequential run on `miriel`.

In the case of topological nested dissections, the IA-HSC method performs better than the IO-HSC strategies 's' and 'g' (third and fourth column, respectively). It should be noted that using a geometric recursive bisection in the case of a topological nested dissection has not been studied here though performance should not reach that of the geometric nested dissection as the separator clustering will not match the rest of the clustering. The performance of the geometric recursive bisection (in the case of geometric nested dissection, first and second column of the figures) is in a large part due to the fact that this local clustering follows the same strategy as the global clustering (the geometric nested dissection) and separator sub-clusters interactions match external clusters well.

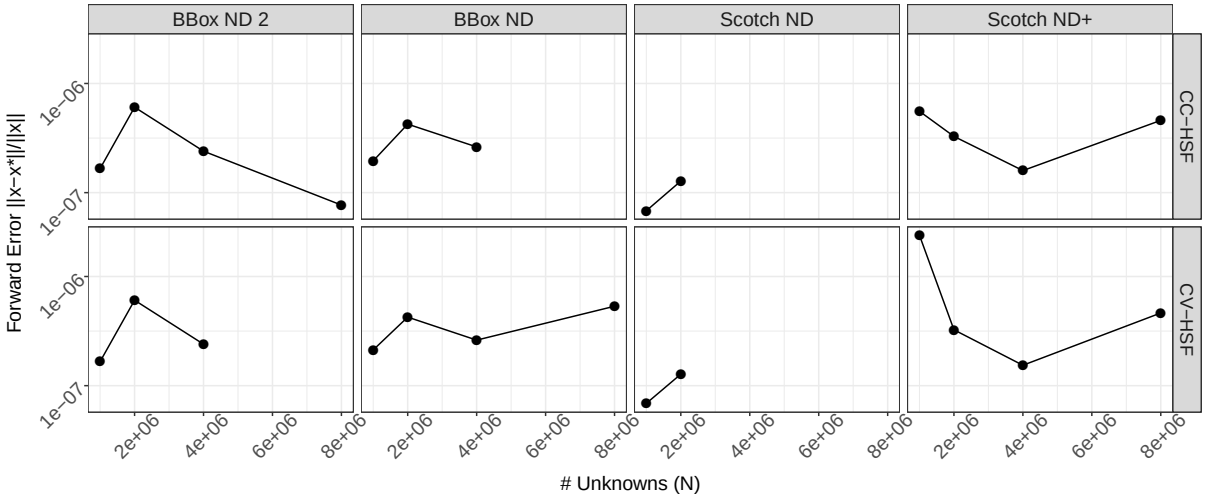
Another observation we can make on these graphs is the effect of using CV-HSF against CC-HSF (the lower row against the upper row in the figures). We may observe that there is no difference in terms of time consumption between the two methods (Fig. 3.7b). This is most certainly due to the current implementation of the operations handling subdivided $\mathcal{H}$-Matrix leaves using IA-LSC, as briefly discussed in § 2.4.3.4. We may however notice a slightly lower memory consumption of the CV-HSF methods compared to the CC-HSF (Fig. 3.7c).

To better understand the effects involved here, we focus now on the IO-HSC methods. Fig. 3.8a displays the theoretical number of entries computed by the symbolic factorization (in blue) and the effective number of entries in the compressed matrix (in green). The blue curve indicates that the Cluster-Vertex information distinctly reduces the theoretical bound of the solver storage compared to the Cluster-Cluster information. However, the effective storage of the solver remains more or less the same due to at least two factors: the current implementation of the solver between the two methods (the format ignores symbolic information in the separator-separator blocks for example) and the ignored zeros due to the admissibility condition. Indeed, if we consider a non-leaf admissible (therefore compressed) block, the underlying symbolic information is ignored in favor of compression. Yet, the Cluster-Vertex approach has a slight advantage over the Cluster-Cluster method, due to the remaining non-admissible leaf blocks that have benefited from this sparser storage.

Finally, we can also observe that compression is far more effective in the case of geometric nested dissections. The compression ratio is defined here as the ratio between the effective number of entries and the estimated number of non-zeros computed by symbolic factorization. The compression ratio is better in the case of BBox ND or BBox ND 2 than for the Scotch ND+ method, as shown in Table 3.7. The effective storage is at around one third of the number of non-zeros computed by symbolic factorization for the BBox ND method. This number drops to one fifth for BBox ND 2 while it reaches 80% for Scotch ND+. This difference in compression comes from the fact that Scotch ND+ computes better suited separators in the nested dissection and thus leads to less fill-in. The geometric nested dissection seems to compress better but the reader should be aware that this may purely come from the fact that a geometric recursive bisection is applied for the clustering of separators. In this thesis we compare a purely geometric approach (discussed in § 2.3.1.1) and a purely topological approach (see §§ 2.3.1.2 and 2.3.1.3) for the overall clustering step. As mentioned earlier, we do not study here the combination



(a) The blue curve represents the theoretical storage (in terms of number of entries) computed by symbolic factorization. The green curve shows the actual storage of the factorized matrix.



(b) Forward error. The threshold parameter is set to $\epsilon = 10^{-4}$.

Figure 3.8: Comparison between theoretical and effective number of entries in the factorized matrix, as well as the forward error of the solution. The column facets indicate which symbolic factorization is used while the row facets indicate the clustering. The separator clustering method is set to IO-HSC. Short pipe test case (Fig. 3.1a). Sequential run on `miriel`.

of a topological approach for the global clustering (nested dissection) with a geometric recursive bisection for the separator clustering, though this should be further investigated in future research. Note that the geometric approach could theoretically also be optimized to lead to less fill-in. In addition to this, BBox ND and BBox ND 2 are also better suited for the strong admissibility condition relying on bounding boxes in our implementation.

Method	Clustering	Estimated nnz	Effective storage	Compression Ratio
CV-HSF	Scotch ND+	$1.42 \cdot 10^{10}$	$1.13 \cdot 10^{10}$	0.80
CC-HSF	Scotch ND+	$2.17 \cdot 10^{10}$	$1.38 \cdot 10^{10}$	0.63
CV-HSF	BBox ND	$2.16 \cdot 10^{10}$	$7.65 \cdot 10^{9}$	0.35
CC-HSF	BBox ND	$2.63 \cdot 10^{10}$	$8.93 \cdot 10^{9}$	0.34
CV-HSF	Scotch ND	$1.90 \cdot 10^{10}$	–	–
CC-HSF	Scotch ND	$3.21 \cdot 10^{10}$	–	–
CV-HSF	BBox ND 2	$4.18 \cdot 10^{10}$	$1.08 \cdot 10^{10}$	0.26
CC-HSF	BBox ND 2	$5.80 \cdot 10^{10}$	$1.23 \cdot 10^{10}$	0.21

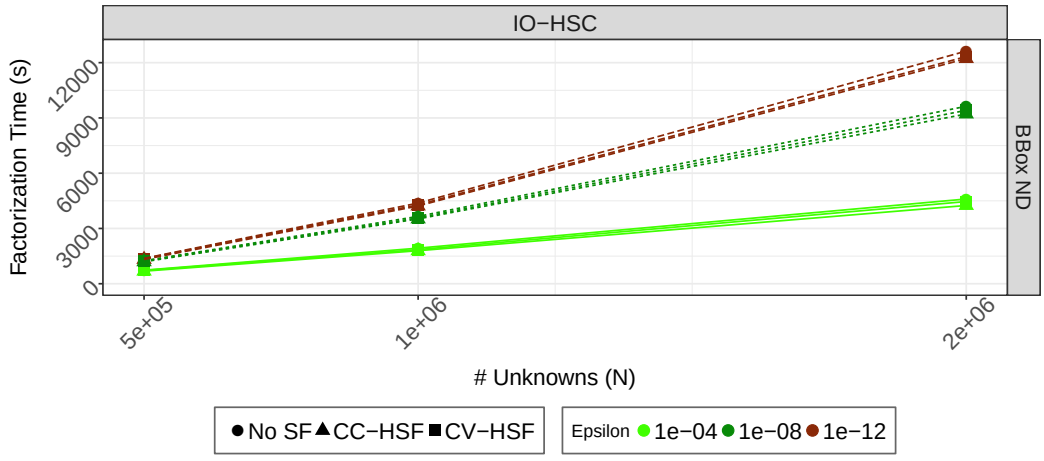
Table 3.7: Estimated and effective number of entries in the factors for multiple clusterings and symbolic factorizations. The empty cells correspond to experiments running out of memory. The compression ratio is the ratio between the effective number of entries and the estimated number of non-zeros computed by symbolic factorization. Short pipe (Fig. 3.1a) with 8M unknowns. Sequential runs on `miriel`.

As a validation of the results, the forward error of each run is shown in Fig. 3.8b. The forward error is below the desired threshold accuracy parameter $\epsilon = 10^{-4}$. In Fig. 3.9, one can also see the effect of different values of the threshold compression parameter ϵ on the numerical factorization (for CC-HSF and CV-HSF methods, using geometric IO-HSC and BBox ND). The larger the threshold is, the lower the computational requirements are. The forward error displayed in Fig. 3.9c seems to indicate that the solver current implementation may sometimes have a larger forward error than the desired threshold. This is not an expected behavior and should be further investigated.

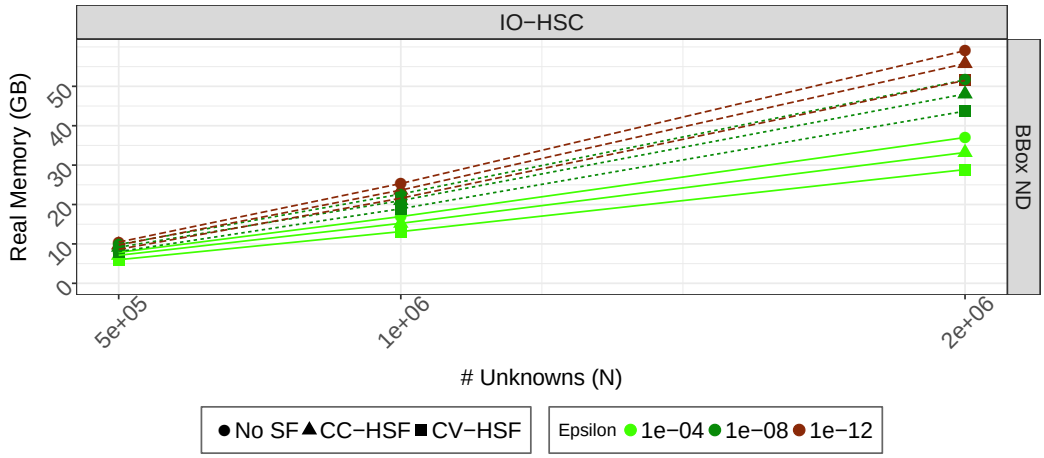
3.5.2 Comparison to $\mathcal{H}$-Matrices without Symbolic Information

One of the main goals of this thesis is to understand the effect of symbolic factorization applied in a hierarchical framework using compression. We intend to demonstrate that using symbolic factorization, while reducing the number of zeros that may be stored by a full-rank solver, also successfully reduces the compressed storage of a hierarchical solver. To that end, Fig. 3.10 shows a comparison of CV-HSF using IO-HSC or IA-HSC separator clusterings to methods that do not use any symbolic factorization.

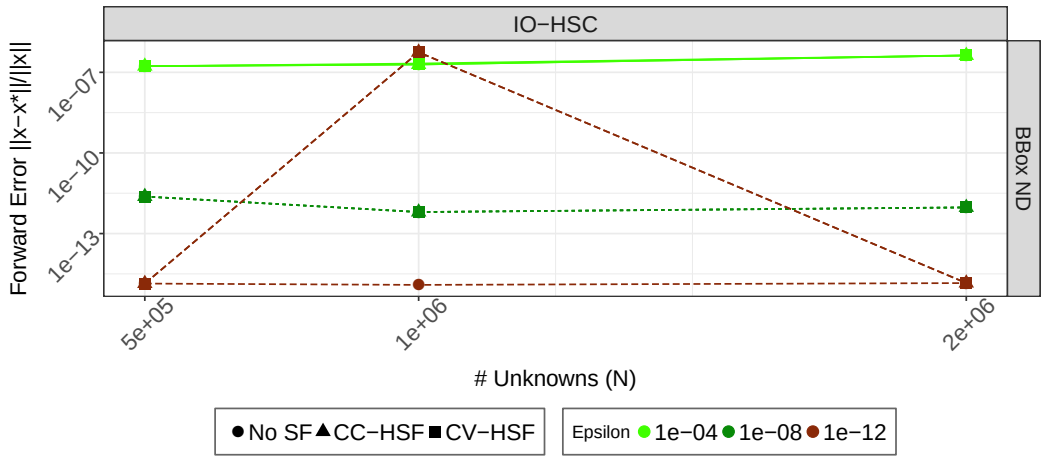
Let us first focus on the methods not using symbolic factorizations (i.e., the first row "No SF" in the graphs). Fig. 3.10a presents the time required to compute a factorization when different global clustering (columns) and separator clustering (colors) methods are



(a) Factorization time.

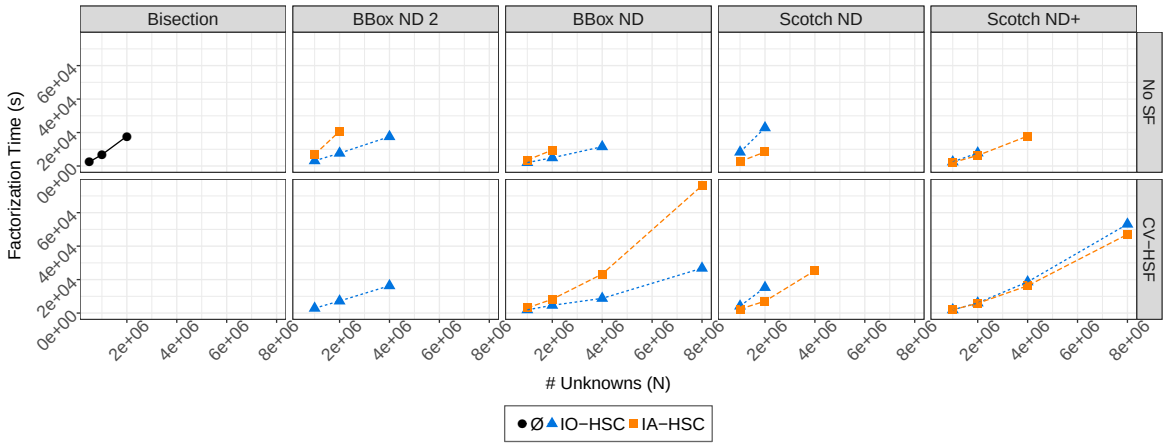


(b) Memory usage.

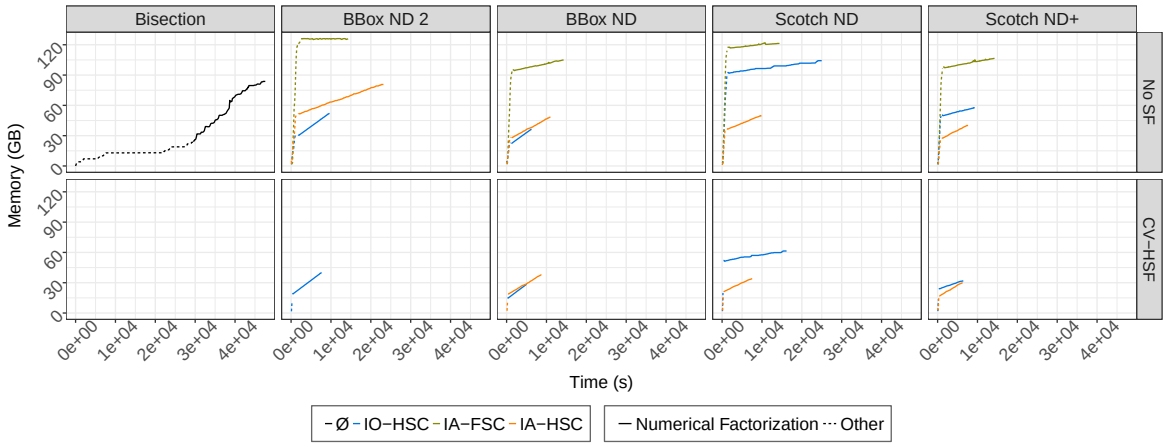


(c) Forward error.

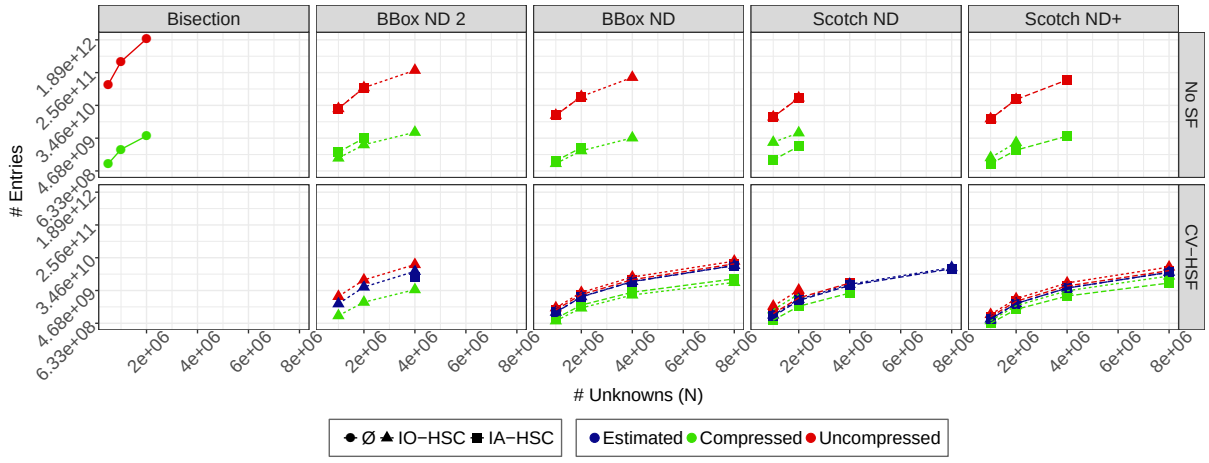
Figure 3.9: Impact of threshold parameter ϵ on numerical factorization time and memory consumption. Short pipe test case (Fig. 3.1a). Sequential run on `occigen`.



(a) Factorization time.



(b) Memory profile on a 2M-unknowns problem.



(c) Number of entries in the factorized matrix. The red curve shows the theoretical storage of the matrix if uncompressed.

Figure 3.10: Comparison of hierarchical factorizations with or without symbolic factorization. No separator clusters are computed in the recursive bisection, it is therefore noted "Ø". Short pipe test case (Fig. 3.1a). Sequential run on `occigen`.

used. One can see in this figure that clusterings based on nested dissection (second to last columns) are generally able to reduce the factorization time compared to the original clustering relying on recursive bisection (first column). Some clusterings do not reduce this time due to the fact that more zeros may be stored if no symbolic factorization is used, even though the actual number of non-zeros is lower. The impact of a nested dissection on factorization time may also greatly vary following its construction. The balance between the left and right domain or the size of the computed separator are thus a factor possibly increasing or reducing the fill-in generated by the factorization. The strong admissibility condition relying on bounding boxes may also not be suited for the nested dissection, as discussed in § 2.3.4, and a better admissibility condition may reduce the complexity of these methods, as proven by the "oracle" mentioned in that previous section. Fig. 3.10b shows the memory profile of the computation, with the numerical factorization highlighted using a continuous line. We can therefore better understand the effects of the methods. In the recursive bisection method, a great part of the computations is taken by the assembly procedure creating the $\mathcal{H}$-Matrix. This assembly step is greatly reduced by the use of nested dissection and the usage of the information that two domains separated in the algorithm do not interact anymore, as discussed in § 1.3.1.1.2. One can also see that efficient variants of the nested dissection (BBox ND, Scotch ND+ for example) reduce both the time and memory consumption of the overall solver. The methods relying on nested dissection use the knowledge that the two subdomains created by the nested dissection are independent to avoid the storage of their interactions, since this computation is trivial. This is observable in Fig. 3.10c, which presents storage measurements of the solver (colors) with different separator clusterings (point shapes/line types). In this figure, we can see that the effective storage of the solver, once uncompressed, is larger in the case of recursive bisection (which is then equal to the storage of the whole dense matrix). The difference is less significant in the compressed storage due to the efficient compression of the extra fill-in in the recursive bisection method.

Note that the methods not relying on symbolic factorization have a partial knowledge on the location of zeros (no interactions between the two independent subdomains). Thus, the effect of symbolic factorization is reduced to the fill-in generated in the separators. The effect is most noticeable in Fig. 3.10c on the uncompressed storage of the solver (in red). However, the usage of compression reduces this advantage as may be observed for example in Fig. 3.10a and 3.10b.

The impact of IA-HSC in the case of the topological nested dissection without symbolic factorization is also more clear in Fig. 3.10b, and especially in the Scotch ND strategy, as it more than halves the memory consumption of the IO-HSC method. This figure also confirms the trend discussed before: the IA-HSC method is not as efficient as IO-HSC in the case of the geometric nested dissection.

All of these informations are also summarized in Table 3.8 for a 2M-unknowns problem. In conclusion of this table and subsection, the methods with the lowest computational requirements are here relying on the geometric nested dissection, namely BBox ND, without Interactions-Aware clustering. The method using the hierarchical symbolic

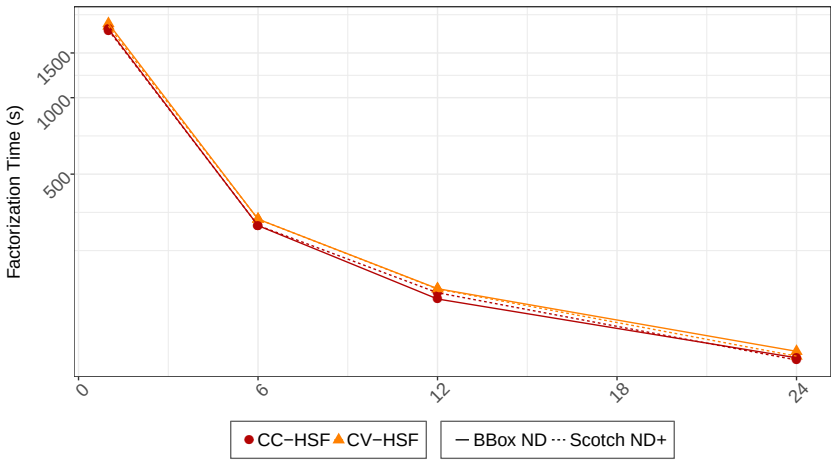
Symbolic Fact.	Clustering	Separator Clustering	Estimated nnz	Effective storage	Uncompressed storage	Factorization time (s)
CV-HSF	Scotch ND+	IO-HSC	$\mathbf{2.11 \cdot 10^9}$	$1.85 \cdot 10^9$	$\mathbf{2.83 \cdot 10^9}$	6,038.09
CV-HSF	Scotch ND+	IA-HSC	$\mathbf{2.12 \cdot 10^9}$	$\mathbf{1.48 \cdot 10^9}$	$\mathbf{2.43 \cdot 10^9}$	5,808.03
CV-HSF	BBox ND	IO-HSC	$3.21 \cdot 10^9$	$\mathbf{1.61 \cdot 10^9}$	$4.15 \cdot 10^9$	**4,633.58**
CV-HSF	BBox ND	IA-HSC	$3.17 \cdot 10^9$	$1.92 \cdot 10^9$	$3.66 \cdot 10^9$	8,274.26
CV-HSF	Scotch ND	IO-HSC	$2.72 \cdot 10^9$	$3.72 \cdot 10^9$	$4.80 \cdot 10^9$	15,288.08
CV-HSF	Scotch ND	IA-HSC	$2.55 \cdot 10^9$	$1.79 \cdot 10^9$	$2.98 \cdot 10^9$	7,095.46
CV-HSF	BBox ND 2	IO-HSC	$5.79 \cdot 10^9$	$2.29 \cdot 10^9$	$8.91 \cdot 10^9$	7,202.59
No SF	Scotch ND+	IO-HSC	–	$3.57 \cdot 10^9$	$4.96 \cdot 10^{10}$	7,798.21
No SF	Scotch ND+	IA-HSC	–	$2.26 \cdot 10^9$	$4.96 \cdot 10^{10}$	6,289.82
No SF	BBox ND	IO-HSC	–	$2.17 \cdot 10^9$	$5.95 \cdot 10^{10}$	4,929.20
No SF	BBox ND	IA-HSC	–	$2.63 \cdot 10^9$	$5.95 \cdot 10^{10}$	9,521.46
No SF	Scotch ND	IO-HSC	–	$6.48 \cdot 10^9$	$5.43 \cdot 10^{10}$	22,841.85
No SF	Scotch ND	IA-HSC	–	$2.84 \cdot 10^9$	$5.43 \cdot 10^{10}$	8,383.83
No SF	BBox ND 2	IO-HSC	–	$3.14 \cdot 10^9$	$1.02 \cdot 10^{11}$	**7,650.25**
No SF	BBox ND 2	IA-HSC	–	$4.66 \cdot 10^9$	$1.02 \cdot 10^{11}$	20,774.85
No SF	Bisection	$\emptyset$	–	$5.36 \cdot 10^9$	$2.00 \cdot 10^{12}$	17,792.52

Table 3.8: Number of entries in the factors and time required to compute the factorization for multiple clusterings with or without symbolic factorization. The "estimated" column corresponds to the storage computed by symbolic factorization. The last line corresponds to the recursive bisection original $\mathcal{H}$-Matrices. Short pipe (Fig. 3.1a) with 2M unknowns. Sequential runs on `miriel`.

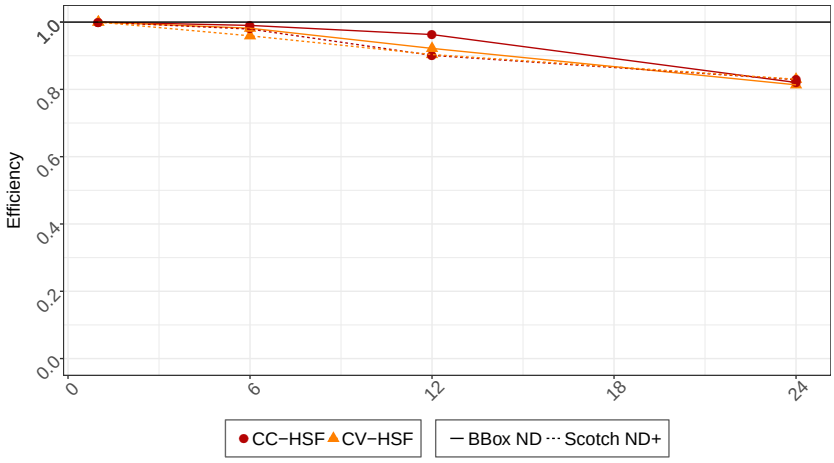
factorization named CV-HSF also has a lower time consumption and more specifically memory consumption.

3.5.3 Multi-core Efficiency

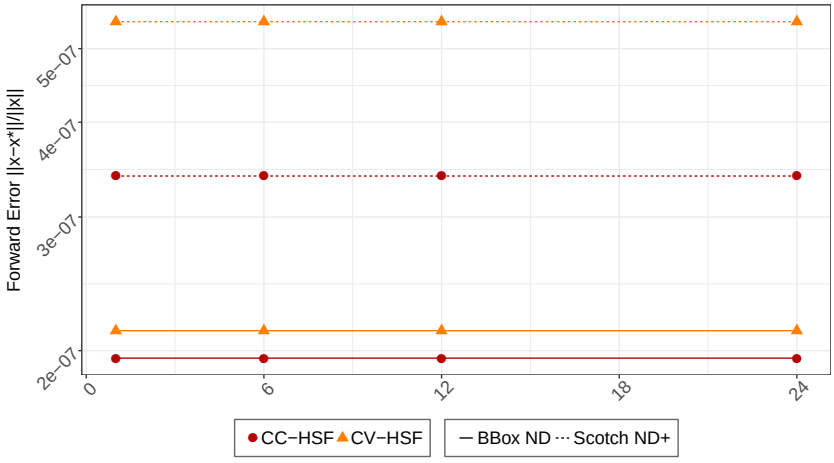
The framework of this thesis has allowed us to study the parallel behavior of the hierarchical methods presented in the previous (sequential) experiments. Only little development has been necessary to adapt the existing (task-based) parallel implementation of the solver [155] to the new algorithms discussed in this thesis, mainly residing in avoiding calling tasks on null submatrices. We therefore study here the scalability of the $\mathcal{H}$-Matrix solver relying on the main algorithms developed in this thesis. In Fig. 3.11, one may observe the factorization time and efficiency of the two symbolic factorizations, namely CC-HSF and CV-HSF scale well on 24 cores with the BBox ND and Scotch ND+ clustering methods. The runtime used in these experiments is toyRT (see § 1.4.1, p. 73). The forward error is constant when the number of threads is changed, thus validating the parallelization.



(a) Factorization time.



(b) Efficiency.



(c) Forward error.

Figure 3.11: Parallel scaling (up to 24 threads) of a problem with 1M = 10^6 unknowns using toyRT on `occigen`.

3.5.4 Parallel Study of Larger Test Cases

The complexity of hierarchical methods should prove to be more effective for very large linear systems. With this fact in mind, we lead this study a step further. Parallel experiments were run on `brise` and `souris`. These machines have a larger RAM capacity and more cores. We are thus able to study the behavior of the $\mathcal{H}$-Matrix methods proposed in this thesis on tens to hundreds of millions of unknowns.

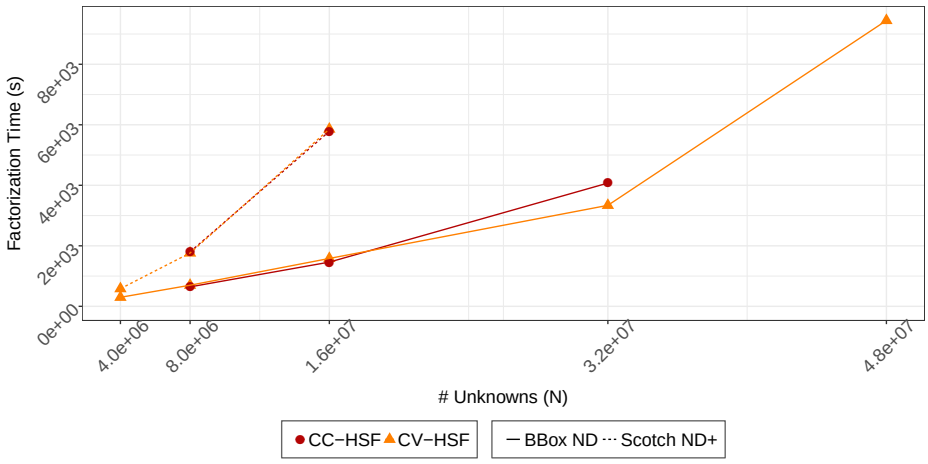
Larger cases run on `brise` (Fig. 3.12) confirm the trends discussed in § 3.5.1. For example, Fig. 3.12c shows the number of entries of the factorized matrix for the BBox ND and Scotch ND+ clusterings with IO-HSC methods. We can see that the theoretical number of entries stored using CV-HSF with Scotch ND+ would be very low compared to that of BBox ND. However, the compression ratio of this method being inferior, the compressed storage of BBox ND is eventually lower than its counterpart. The difference between CC-HSF and CV-HSF is also very distinct in the theoretical analysis while being fainter in the effective storage of the solver. We can also see how that impacts the real memory usage of the solver in Fig. 3.12b. The clustering method has also a strong effect on factorization time, as shown in Fig. 3.12a. The time required to compute the factorization is approximately equivalent for both CV-HSF and CC-HSF methods, which may be due to the current implementation, as mentioned earlier.

On `souris`, one can see the stability of the trends between CC-HSF and CV-HSF in Fig. 3.13b and 3.13c, CV-HSF consuming less memory than CC-HSF. However it appears that the time required to compute a factorization at this scale is lower for CC-HSF than CV-HSF, as shown in Fig. 3.13a. Unfortunately, it appears the Scotch ND+ method has implementation issues at this scale. Nonetheless, the symbolic factorization of this method may be computed and the number of estimated non-zeros is therefore shown in Fig. 3.13c.

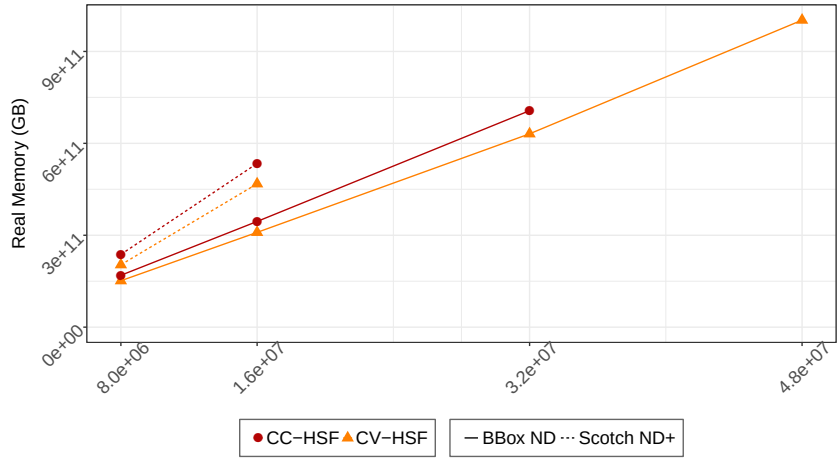
3.5.5 Comparison to a Sparse Direct Solver

In § 3.5.2, we have studied the influence of symbolic factorization on $\mathcal{H}$-Matrices. We have concluded that it is possible and beneficial to use sparse techniques such as the nested dissection and symbolic factorization to reduce the computational complexity of a hierarchical solver. Yet, sparse direct methods have also proposed to introduce hierarchical techniques of compression to the compression of dense submatrices arising in the factorization of sparse matrices. Comparisons have been performed between hierarchical solvers and sparse solvers in the past, for example in [105], using an implementation of $\mathcal{H}$-Matrices combined with nested dissection and several sparse solvers (MUMPS, PARDISO, SuperLU, UMFPACK, ...). In [193], the authors also perform a comparison of a $\mathcal{H}$-Matrix solver based on nested dissection with MUMPS and PARDISO. However, we here position the hierarchical methods proposed in this thesis to a reference sparse solver (MUMPS) also relying on low-rank compression.

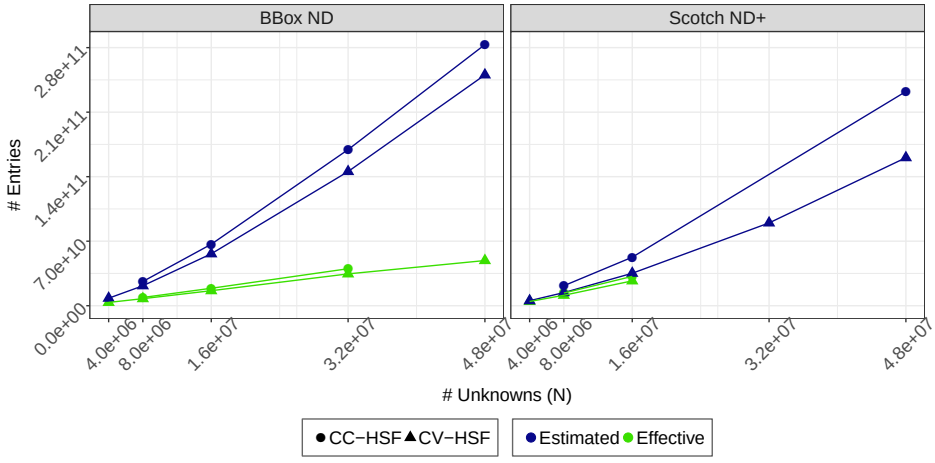
We first compare sequential runs of the best method (the CV-HSF variant with IO-HSC) of our implementation of a $\mathcal{H}$-Matrix solver, namely HMAT (implementation



(a) Numerical factorization time.

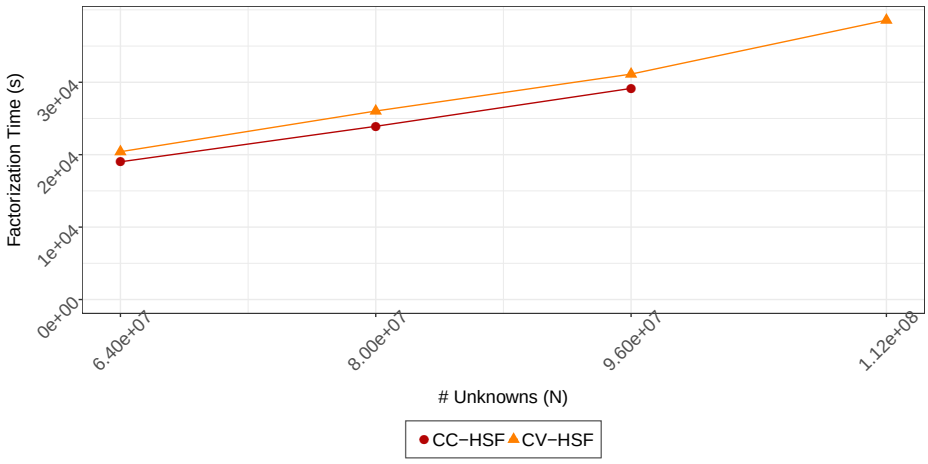


(b) Real memory peak.

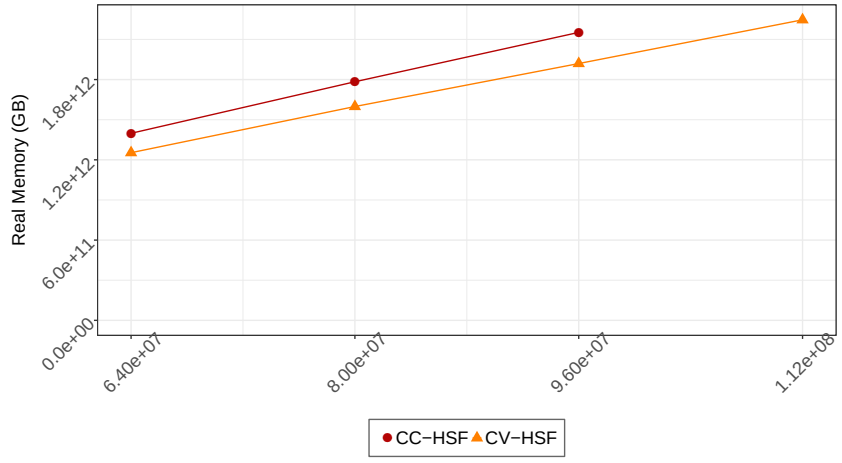


(c) Number of entries of the factorized matrix.

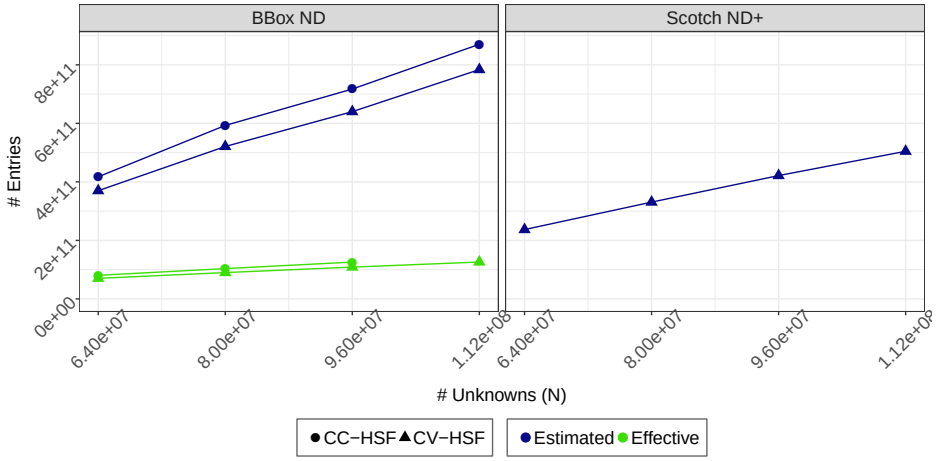
Figure 3.12: Performance for larger test cases with IO-HSC methods. Short pipe test case (Fig. 3.1a). Parallel run (96 threads) using the toyRT runtime (see § 1.4.1, p. 73) on `brise`.



(a) Numerical factorization time.



(b) Real memory peak.



(c) Number of entries in the factorized matrix.

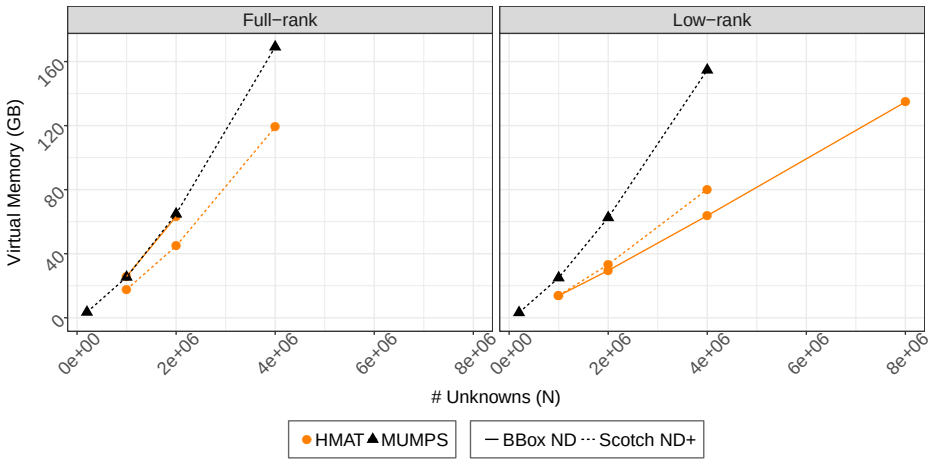
Figure 3.13: Performance for larger test cases of BBox ND methods. Partial results for Scotch ND methods (c). Short pipe test case (Fig. 3.1a). Parallel run (96 threads) using toyRT on `souris`.

details given in § 3.2.3), to the MUMPS solver (details of the configuration given in § 3.2.3), the low-rank version of MUMPS usually being referred to as BLR-MUMPS. Fig. 3.14a and 3.14b show the virtual and real memory usage of both solvers. Note that MUMPS consumes more virtual memory than the available RAM at 4M unknowns, but the real memory usage of the solver is lower. To run the HMAT solver without compression, i.e., full-rank, we simply rely on a admissibility condition that is always false, leading to the $\mathcal{H}$-Matrix leaves being Full-Matrices. The full-rank BBox ND variant consumes more memory than its Scotch ND+ counterpart, while the low-rank versions follow the reverse trend. The explanation comes from the fact that Scotch ND+ computes better suited separators in the nested dissection, leading to less fill-in, as discussed in the previous paragraph. The effects discussed above may also be observed in Fig. 3.14c. In the latter figure, the number of entries of a compressed storage for the BLR-MUMPS variant returns the same number of entries as the full-rank variant, with the version 5.1.2 of MUMPS used here (see § 3.2.3). Therefore, MUMPS results are indistinguishable from one another and are also close to the number of entries estimated by HMAT in the Scotch ND+ variant (in blue). These results are also presented in Table 3.9 for a 2M-unknowns problem. One can readily see the Scotch ND+ does not lead to the same estimated number of entries from a run to another (for example we could expect the same number for a low-rank and a full-rank variant of the same method), due to the heuristic characteristic of the method.

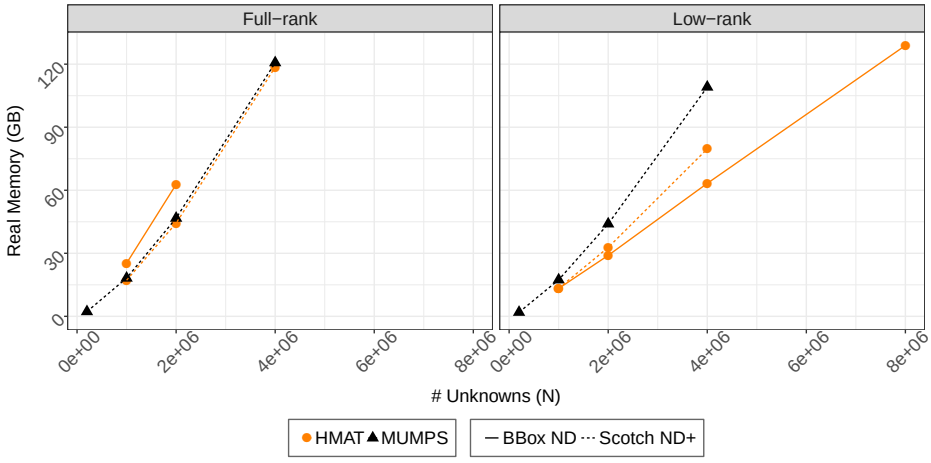
Solver	Precision	Clustering	Estimated nnz	Effective storage	Factorization time (s)
HMAT	Low-rank	Scotch ND+	$2.11 \cdot 10^9$	$1.85 \cdot 10^9$	6,038.09
HMAT	Low-rank	BBox ND	$3.21 \cdot 10^9$	$1.61 \cdot 10^9$	4,633.58
HMAT	Full-rank	BBox ND	$3.21 \cdot 10^9$	$3.84 \cdot 10^9$	7,463.92
HMAT	Full-rank	Scotch ND+	$2.05 \cdot 10^9$	$2.65 \cdot 10^9$	3,959.58
MUMPS	Low-rank	Scotch ND+	$2.19 \cdot 10^9$	$2.19 \cdot 10^9$	459.19
MUMPS	Full-rank	Scotch ND+	$2.15 \cdot 10^9$	$2.15 \cdot 10^9$	1,667.20

Table 3.9: Storage and factorization time for HMAT and MUMPS solvers. HMAT is run with CV-HSF. The "estimated" column corresponds to the storage computed by symbolic factorization. Short pipe (Fig. 3.1a) with 2M unknowns. Sequential runs on `miriel`.

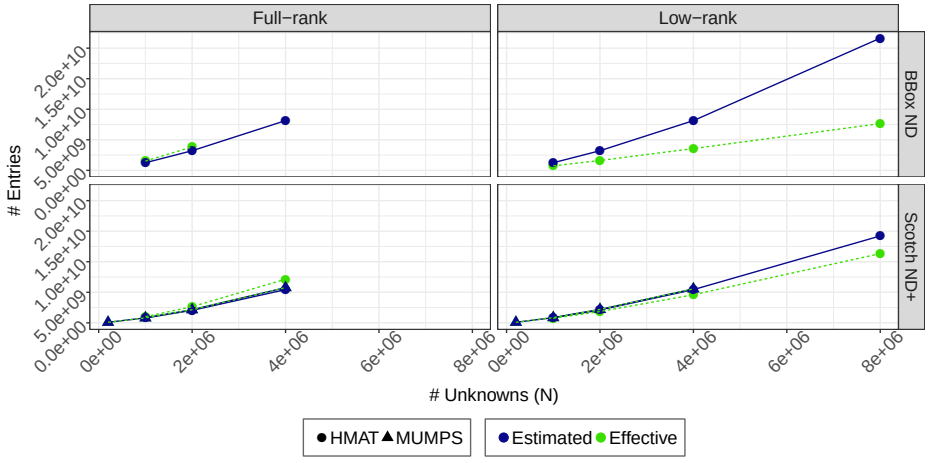
The number of floating-point operations of each method is shown Fig. 3.15a. We may see that the full-rank HMAT solver using Scotch ND+ seems to be able to reach a number of Flop similar to the sparse direct MUMPS solver. This result means that we have successfully implemented a hierarchical solver leading to the same (full-rank) computations as the reference sparse solver. The full-rank BBox ND variant leads to more Flop than the Scotch ND+ variant (for the same reasons as discussed above regarding fill-in). Let us now discuss the results involving low-rank operations. The low-rank variant of Scotch ND+ leads to more Flop than its full-rank counterpart. Yet, the low-rank variant of BBox ND leads to less Flop, and is also able to reach a number of Flop similar to MUMPS. The slope of this latter method is also flatter. This seems to indicate that the



(a) Virtual memory peak.



(b) Real memory peak of the solver.



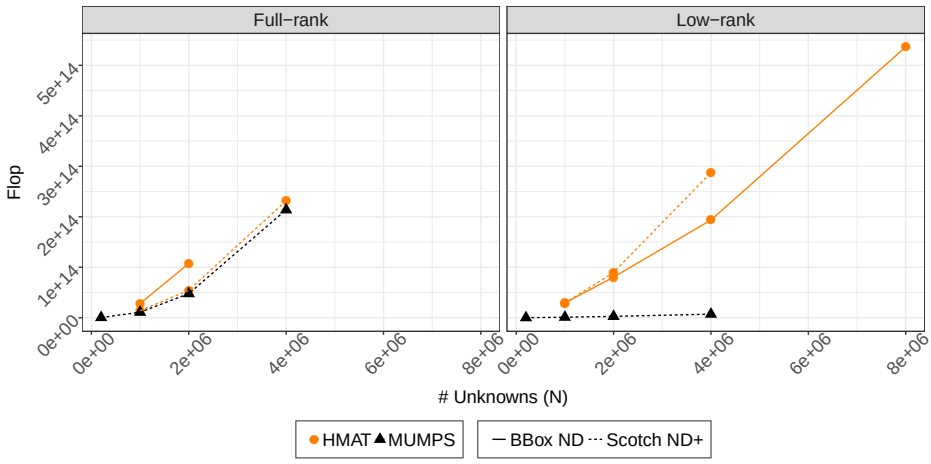
(c) Number of entries in the factorized matrix.

Figure 3.14: Performance for middle-range test cases of full-rank and low-rank MUMPS and HMAT solvers. HMAT is run with the CV-HSF variant. Short pipe test case (Fig. 3.1a). Sequential run on `miriel`.

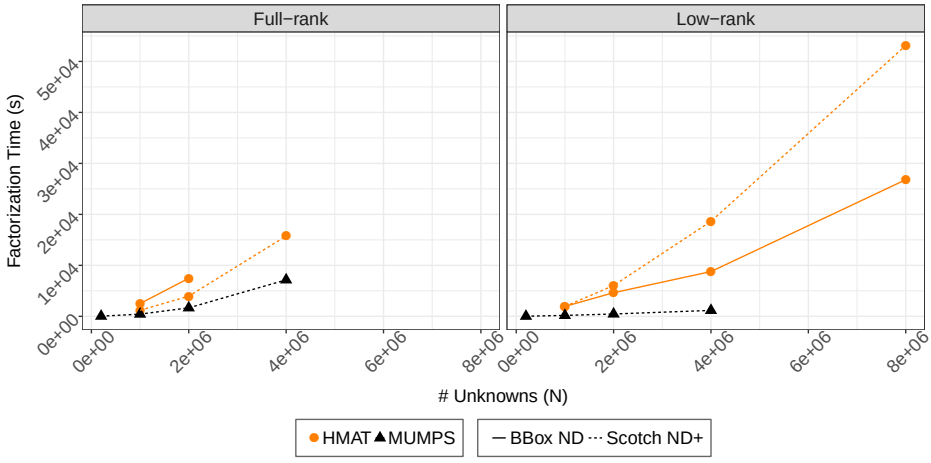
HMAT solver implementation, at this early stage of development towards sparse-oriented structures, is more suited for low-rank operations based on geometric clustering than based on topological information. Fig. 3.15b shows the time required to compute a factorization for each method. The sequential HMAT solver is slower than the reference MUMPS solver, indicating that the HMAT solver is not currently as efficient as MUMPS in terms of Flop/s. The low-rank version of MUMPS is faster than its full-rank counterpart and, consequently, than the HMAT solver.

Fig. 3.15c validates the solution of each method by confirming that the full-rank forward error is close to the machine precision while the low-rank forward error is close to the threshold ϵ, here equal to 10^{-4}.

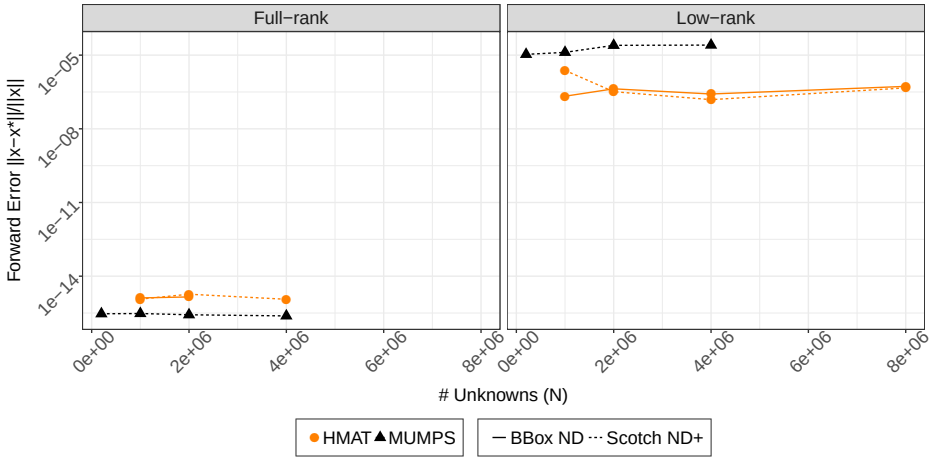
Fig. 3.16 shows the scalability of each solver on `occigen` for a 2M-unknowns problem. First, as we can see, the full-rank version of HMAT using Scotch ND+ computes a factorization faster than its low-rank version, meaning some low-rank operations may be more time-consuming in our implementation than their full-rank counterpart. However, the size of this problem is still quite small and the low-rank solver is expected to fare better on larger problems. As discussed in § 3.5.3, the HMAT solver seems to scale well on 24 cores, with a parallel efficiency around 0.9 on 24 cores for all the methods presented here. The efficiency degrades more rapidly for MUMPS. In a purely multi-thread approach, the HMAT solver still scales on 96 cores, as shown in Fig. 3.17, whereas MUMPS must resort to distributed parallelism. Using 8 MPI processes, MUMPS seems to be able to scale better. The results on 96 cores are also shown for larger problems in Fig. 3.18a. However, when using MPI processes, the tree parallelism of the multifrontal method implies that more fronts may be eliminated at the same time and the memory space dedicated to these fronts is therefore held in the RAM at the same time, as shown in Fig. 3.18b. In fact, the 16M unknowns problem cannot fit into memory using the MPI + threads combination. The version of MUMPS used here does not include memory-aware strategies avoiding this sort of problematics such as discussed in [9]. The memory displayed here is calculated by the solver as we have not been able to retrieve the real memory peak across multiple processors at the time of these experiments. The number of entries stored for each solver is also shown in Fig. 3.18c. The storage of MUMPS is indistinguishable from the HMAT with Scotch ND+ estimated curve (in blue). The factorization using MUMPS fits into the RAM until 16M unknowns on `brise` (for the multi-thread version), though we may speculate how much it will take by following the blue curve of Scotch ND+ method. These results are also shown in Table 3.10 for a problem with 8M unknowns.



(a) Flop.

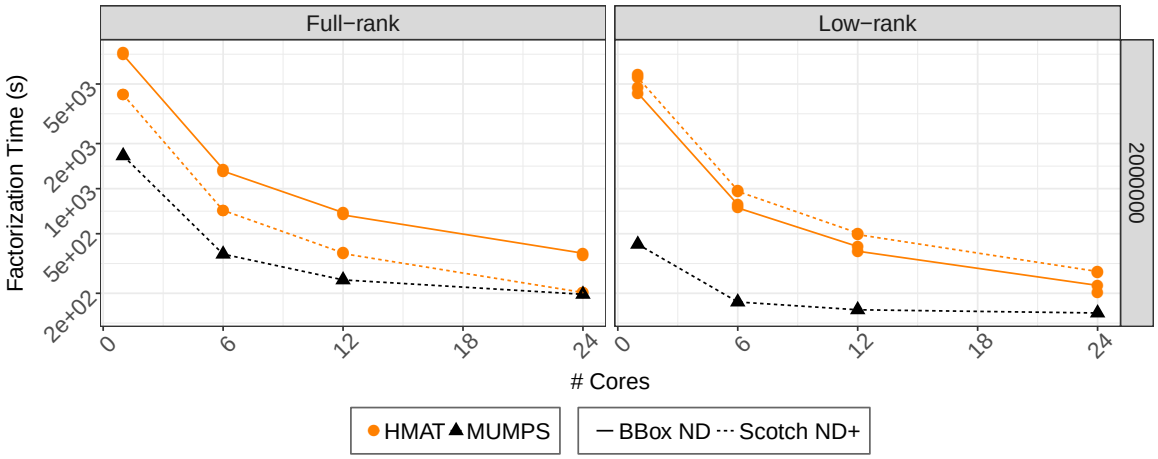


(b) Numerical factorization time.

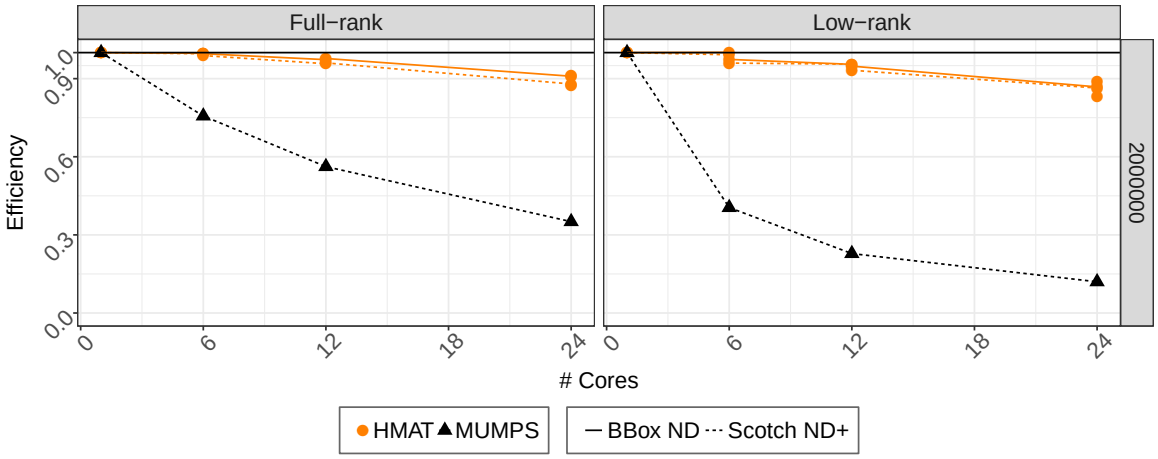


(c) Forward error. The threshold is $\epsilon = 10^{-4}$. MUMPS is used as a full-rank solver here.

Figure 3.15: Performance for middle-range test cases of full-rank and low-rank MUMPS and HMAT solvers. HMAT is run with the CV-HSF variant. Short pipe test case (Fig. 3.1a). Sequential run on `miriel`.

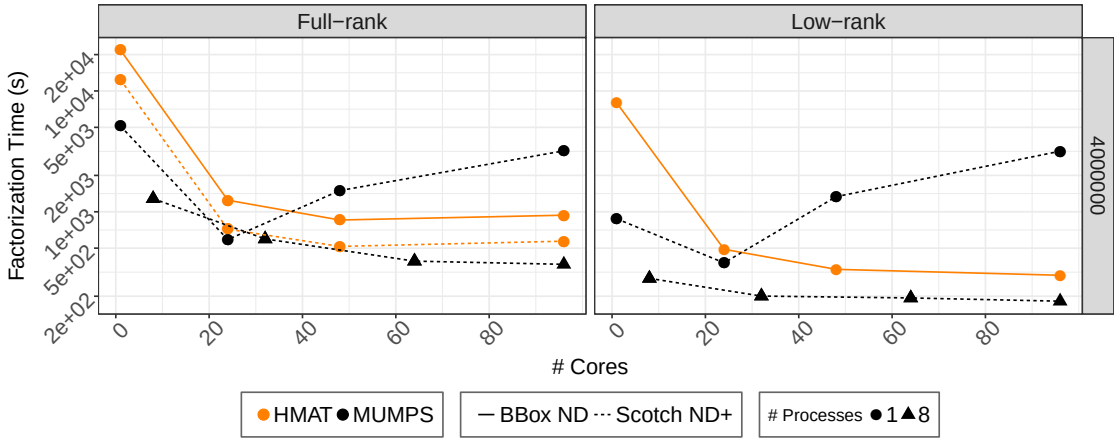


(a) Factorization time.

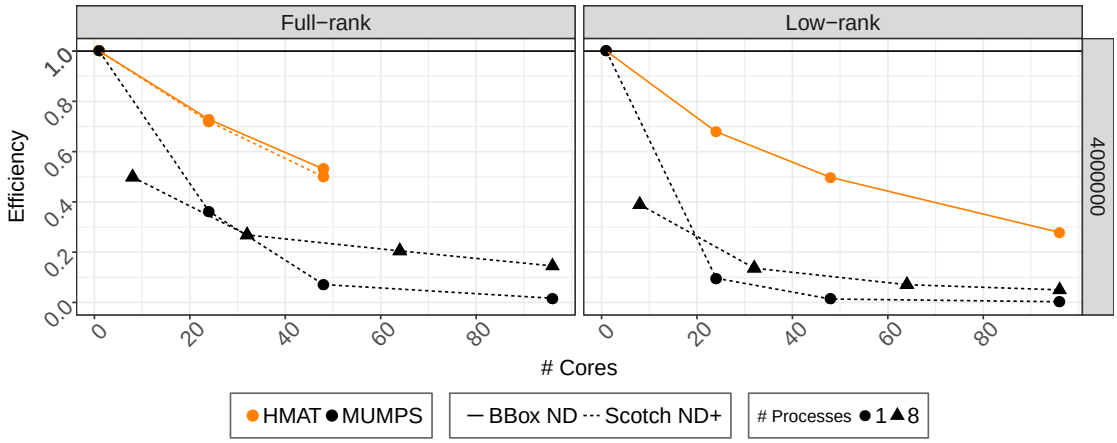


(b) Efficiency.

Figure 3.16: Multi-thread parallel scaling on 24 cores of a problem with 2M unknowns on occigen.

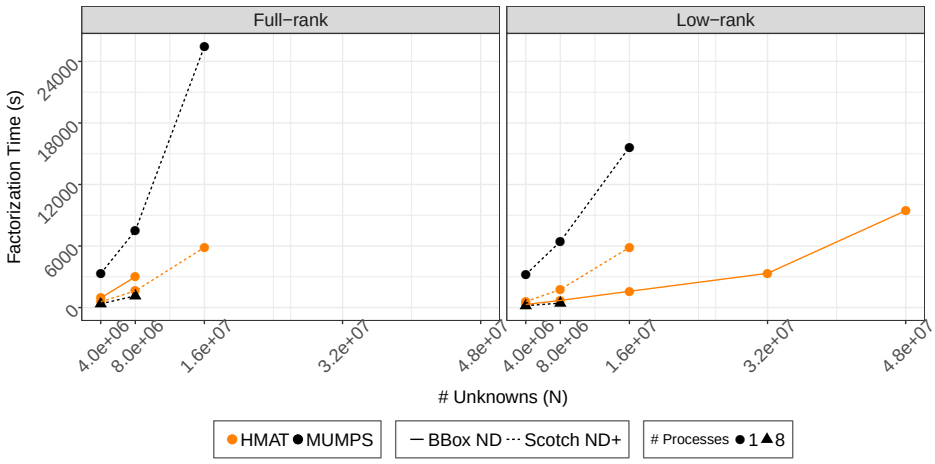


(a) Factorization time.

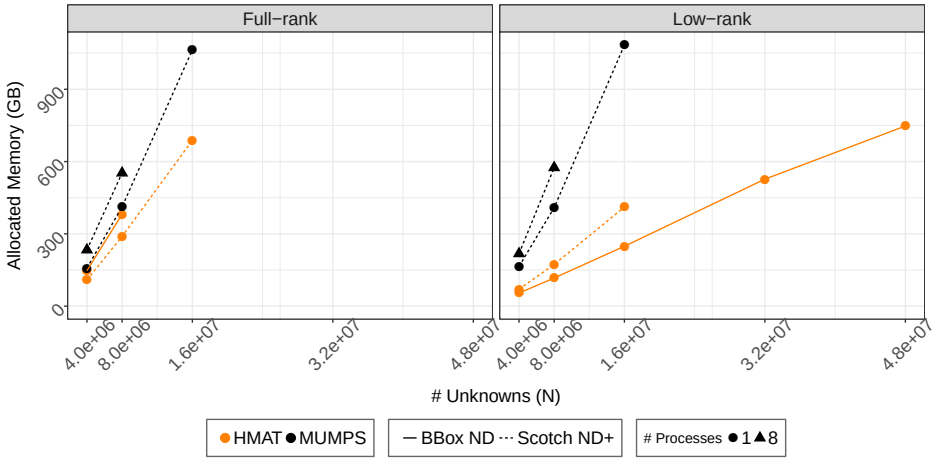


(b) Efficiency.

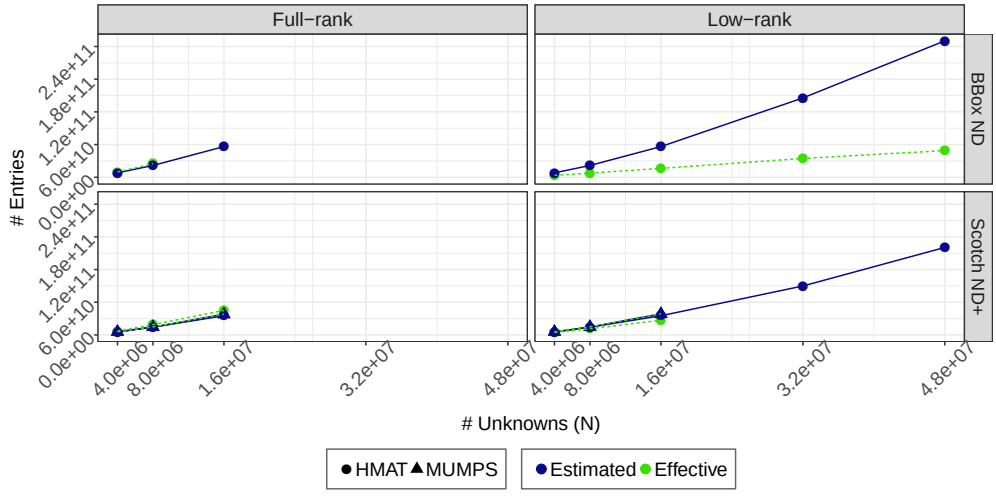
Figure 3.17: Parallel scaling on 96 cores of a problem with 4M unknowns on `brise`. HMAT and MUMPS are run with 1 MPI process with 1 to 96 threads (●). MUMPS is also run with 8 MPI processes, with 1 to 12 threads (▲).



(a) Numerical factorization time.



(b) Allocated memory across all processes.



(c) Number of entries in the factorized matrix.

Figure 3.18: Performance for larger test cases. HMAT is run with CV-HSF. Parallel runs (96 cores) on `brise`.

Solver	Precision	Clustering	#p	#t	Estimated nnz	Effective storage	Factorization time (s)
HMAT	Low-rank	Scotch ND+	1	96	$1.42 \cdot 10^{10}$	$\mathbf{1.13 \cdot 10^{10}}$	1,761.20
HMAT	Low-rank	BBox ND	1	96	$2.16 \cdot 10^{10}$	$\mathbf{7.65 \cdot 10^{9}}$	**696.99**
HMAT	Full-rank	Scotch ND+	1	96	$1.42 \cdot 10^{10}$	$1.90 \cdot 10^{10}$	1,620.39
HMAT	Full-rank	BBox ND	1	96	$2.16 \cdot 10^{10}$	$2.50 \cdot 10^{10}$	2,998.37
MUMPS	Low-rank	Scotch ND+	1	96	$1.46 \cdot 10^{10}$	$1.46 \cdot 10^{10}$	6,431.88
MUMPS	Full-rank	Scotch ND+	1	96	$1.48 \cdot 10^{10}$	$1.48 \cdot 10^{10}$	7,500.00
MUMPS	Low-rank	Scotch ND+	8	12	$1.44 \cdot 10^{10}$	$1.48 \cdot 10^{10}$	**431.86**
MUMPS	Full-rank	Scotch ND+	8	12	$1.43 \cdot 10^{10}$	$1.47 \cdot 10^{10}$	1,136.77

Table 3.10: Number of entries and factorization time for HMAT and MUMPS solvers. HMAT is run with CV-HSF. The "estimated" column corresponds to the storage computed by symbolic factorization. Short pipe (Fig. 3.1a) with 8M unknowns. Parallel runs on 96 cores on `brise`. #p is the number of MPI processes. #t is the number of threads per MPI process.

In this section, we have found that:

- using a clustering based on nested dissection instead of recursive bisection is indeed beneficial for $\mathcal{H}$-Matrices (confirmation of results from the community);
- our geometric nested dissection leads to better performance on the short pipe test case than the topological approach, due to a more efficient compression;
- $\mathcal{H}$-Matrices relying on nested dissection may be improved by the use of symbolic factorization, in particular by the CV-HSF method, which leads to less storage than the CC-HSF method;
- the CV-HSF method leads to a number of non-zeros close to that of a sparse direct solver, and the storage of the $\mathcal{H}$-Matrix without compression is also similar to that of a sparse direct solver, validating the implementation of the symbolic factorization in a hierarchical environment;
- low-rank compression successfully reduces even further the memory consumption of the hierarchical solver;
- however, low-rank operations are as costly as full-rank operations in the Scotch ND+ approach in our implementation, which is not the expected behavior of a low-rank solver;
- the hierarchical solver with or without our methods scales well on many cores in a multithreaded parallel task-based environment.

3.6 FEM/BEM coupling

The larger goal of this thesis is to reduce the (arithmetic and memory) complexity of the solution of the FEM/BEM coupling, such as discussed in Chapter 1. To this end, we lead here studies on the long pipe (Fig. 3.1b, p. 155), closer to real-life applications. Table 3.11 lists the respective sizes of the FEM mesh and the BEM mesh for a given number of unknowns corresponding to the studied test case. The number of FEM unknowns is largely dominant. The efficient usage of the sparsity of the FEM-FEM matrix A_{vv} is therefore critical to the overall solver memory consumption. The displayed length L of

# Unknowns	n_{FEM}	n_{BEM}	L/λ
250,000	218,300	31,700	31.7
500,000	449,600	50,400	40.0
1,000,000	919,864	80,136	50.4
2,000,000	1,873,200	126,800	63.4

Table 3.11: Respective sizes of the FEM mesh and the BEM mesh for a given overall number of unknowns of the long pipe test case (Fig. 3.1b), as well as the length L of the object divided by the wavelength λ.

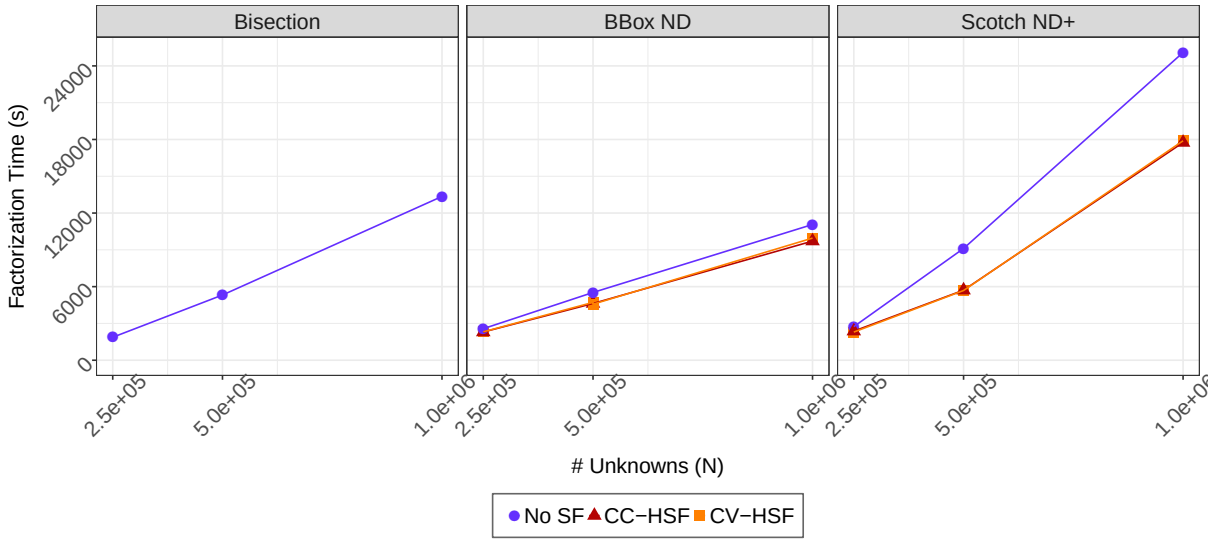
the object is divided by the wavelength λ to give a general idea of the physical dimensions of the considered object. Following § 1.4.4, we study the HMAT approach in § 3.6.1, i.e., the solution of the coupling using $\mathcal{H}$-Matrices only, and compare the MUMPS+SPIDO approaches in § 3.6.2, i.e., the Multi-Solve and Multi-Factorization techniques. We also show preliminary results comparing the best variants of the HMAT approaches and the MUMPS+SPIDO approaches in § 3.6.3.

3.6.1 Comparison of $\mathcal{H}$-Matrix Methods

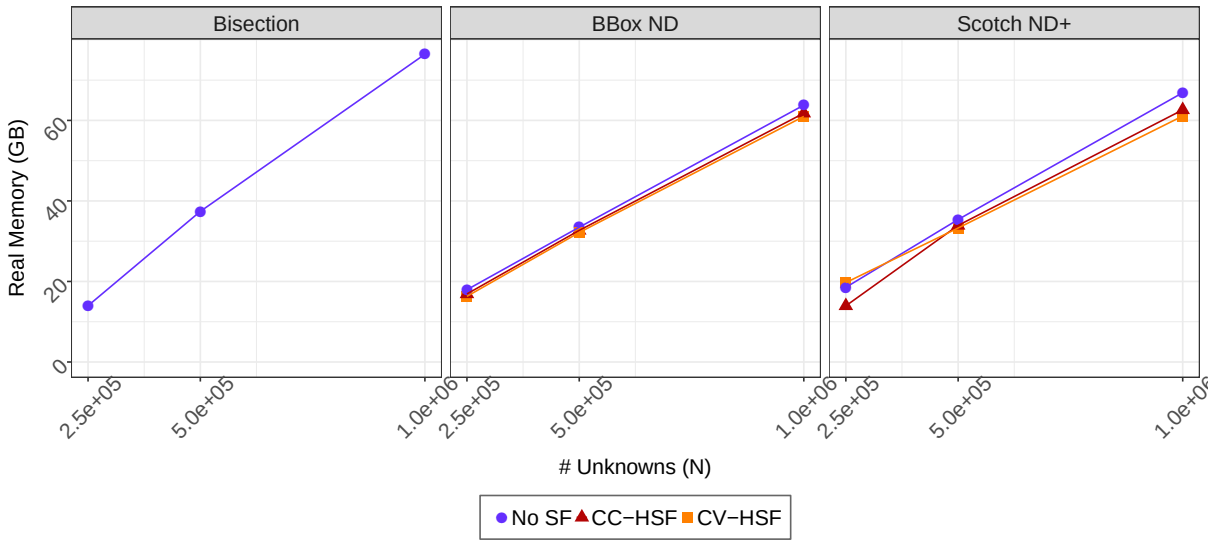
The solution of the FEM/BEM coupling using $\mathcal{H}$-Matrix, as mentioned in § 1.4.4.2 and § 2.5, follows the decomposition of the overall matrix A into:

$$A = \begin{bmatrix} A_{vv} & A_{vs} \\ A_{sv} & A_{ss} \end{bmatrix}. \tag{3.2}$$

To cluster the partition of unknowns corresponding to the FEM-associated mesh v, we either rely on recursive bisection (original $\mathcal{H}$-Matrix clustering) or nested dissection. These global clustering methods (Bisection, BBox ND, Scotch ND+) are given in the columns of the following figures. Furthermore, we also study the use of symbolic factorization on the submatrix A_{vv}. Fig. 3.19a shows the time consumption of the overall hierarchical factorization procedure, therefore including the elimination of FEM unknowns, the computation of the Schur complement (see § 1.4.3, Eq. (1.29) and (1.30)) as well as the factorization of the Schur complement matrix. The BBox ND methods effectively reduce the time required to compute this factorization with respect to the

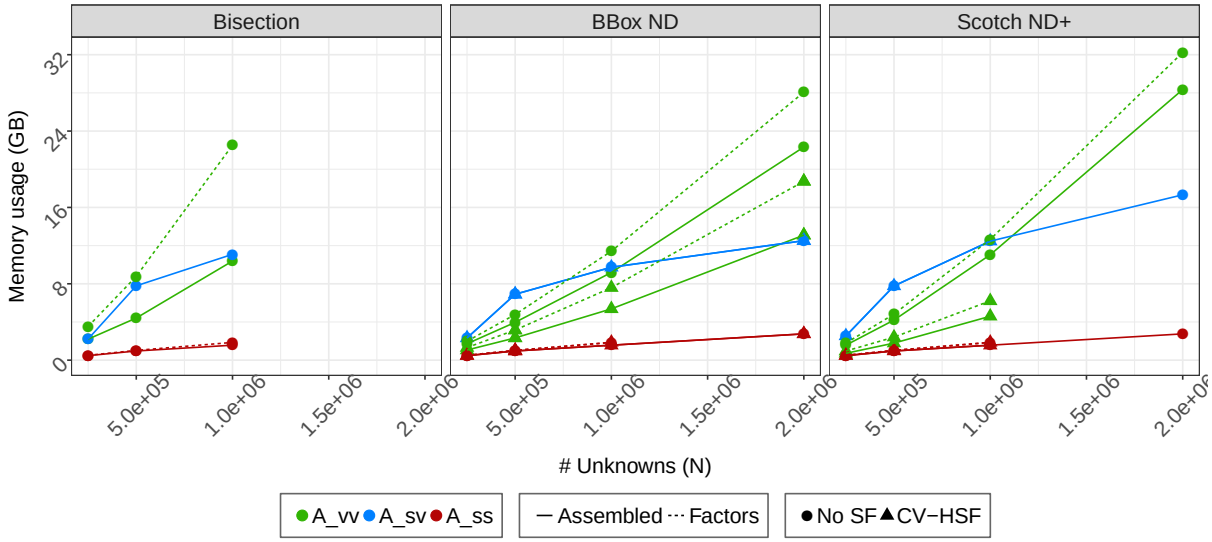


(a) Factorization time.

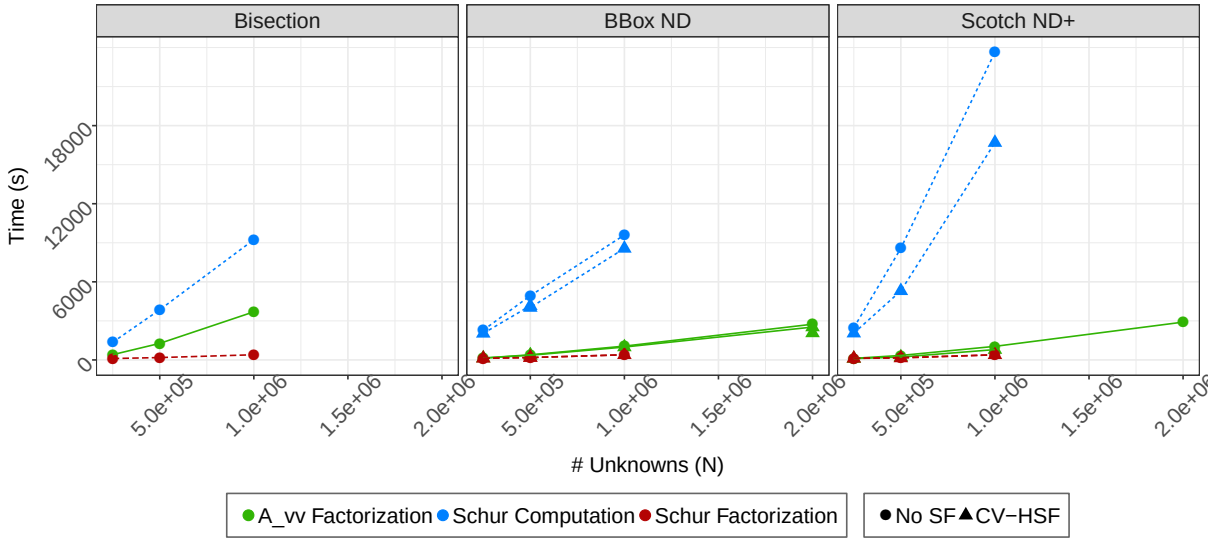


(b) Real memory peak.

Figure 3.19: FEM/BEM coupling solution using $\mathcal{H}$-Matrices. Long pipe test case (Fig. 3.1b). Sequential runs on `miriel`.



(a) Submatrices memory consumption.



(b) Time required to compute factorizations and the Schur complement substeps of the FEM/BEM coupling solution.

Figure 3.20: FEM/BEM coupling solution using $\mathcal{H}$-Matrices. Long pipe test case (Fig. 3.1b). Sequential runs on `miriel`.

recursive bisection, although the Scotch ND+ methods seem not to be able to reduce the overall factorization time. We will give hints concerning the causes of this larger time requirement of Scotch ND+ later in the discussion. The symbolic factorization also helps reducing the factorization time of the nested dissection methods.

Fig. 3.19b show the overall real memory peak reached by the solver during the computations. Both nested dissections are able to lower the memory consumption of the solver compared to the recursive bisection method. Symbolic factorizations are also able to further lower the memory requirements of the solver.

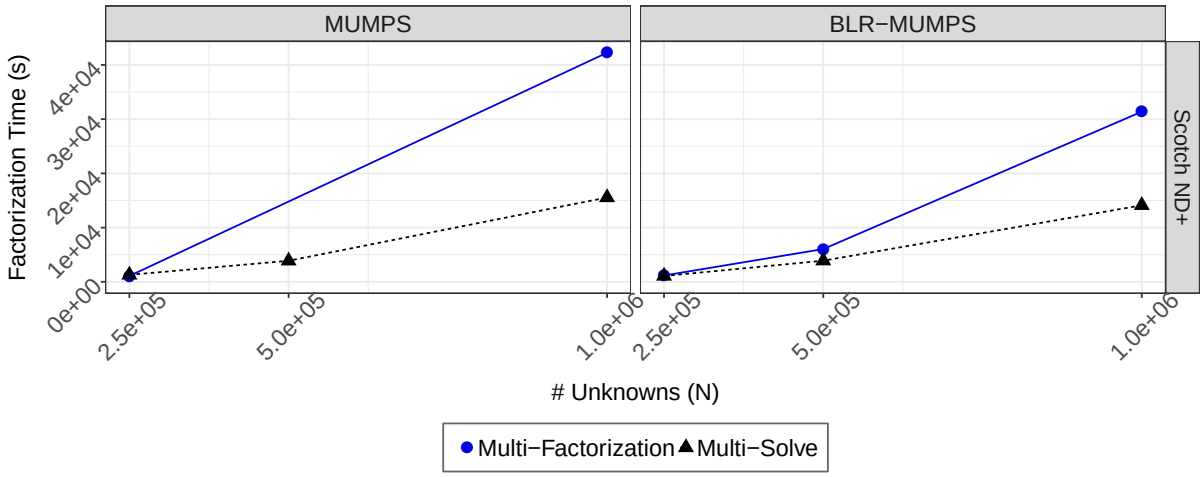
Finally, Fig. 3.20a shows the memory consumption of each submatrix of Eq. (3.2), i.e., A_{vv}, the FEM-FEM subsystem, A_{sv}, representing BEM-FEM interactions, and A_{ss}, corresponding to the BEM-BEM subsystem. We display the information of the matrices once assembled as well as the factorized memory consumption of the matrices A_{vv} and A_{ss}. The matrix A_{ss}, before and after factorization, represents only a small fraction of the overall memory consumption of the solver. While the factorized A_{vv} represents the largest part of the memory consumption of the solver in the recursive bisection strategy, the matrix A_{sv} takes a larger part in the other methods, due to the reduction of the memory consumption of the matrix A_{vv} using nested dissection and symbolic factorization. However, the memory consumption of A_{sv} grows more slowly than A_{vv} with respect to the total number of unknowns. Therefore A_{vv} becomes once more the bottleneck of the solver for larger problems, even for the most efficient of the symbolic factorizations, the CV-HSF method for a problem with 2M unknowns. The matrix associated with the long pipe test case is sparser than with the short pipe. This gives an advantage to the Scotch ND+ method against the BBox ND method in terms of the storage of the (factorized) A_{vv} matrix when a symbolic factorization (CV-HSF) is computed. Note that the computation of the 2M-unknowns problem is only partial and has not finished (reaching the walltime fixed before execution), as one may notice in Fig. 3.20b. In this figure, we can see how the factorization of the sparse matrix A_{vv} requires less time to be computed in the case of nested dissections. However, we have not studied here the reduction of the time required to computed the Schur complement. In particular, if the BBox ND method coupled with symbolic factorization (of the matrix A_{vv}) seems to be able to reduce a little the computation time of this step, the computation of the Schur complement is much longer in the case of Scotch ND+. Multiple factors may explain this problem. Firstly, the Minimum Fill method leads to very small column clusters associated with the FEM mesh, impacting the matrix A_{sv} storage. Smaller clusters lead to less compression and the non-geometrical clustering also leads to more scattered non-zeros. An example of such a matrix A_{sv} in a FEM/BEM coupling is shown in Fig. B.6 in appendix B. The absence of symbolic factorization over this matrix allows for the storage of a large number of zeros. In fact, this may be observed in Fig. 3.20a, which shows the larger amount of memory of the matrix A_{sv} in the case of Scotch ND+. Secondly, the presence of a large number of smaller blocks to handle is not an expected scenario for $\mathcal{H}$-Matrices, leading to a strong decline in performance.

3.6.2 Comparison of Multi-Solve and Multi-Factorization Methods

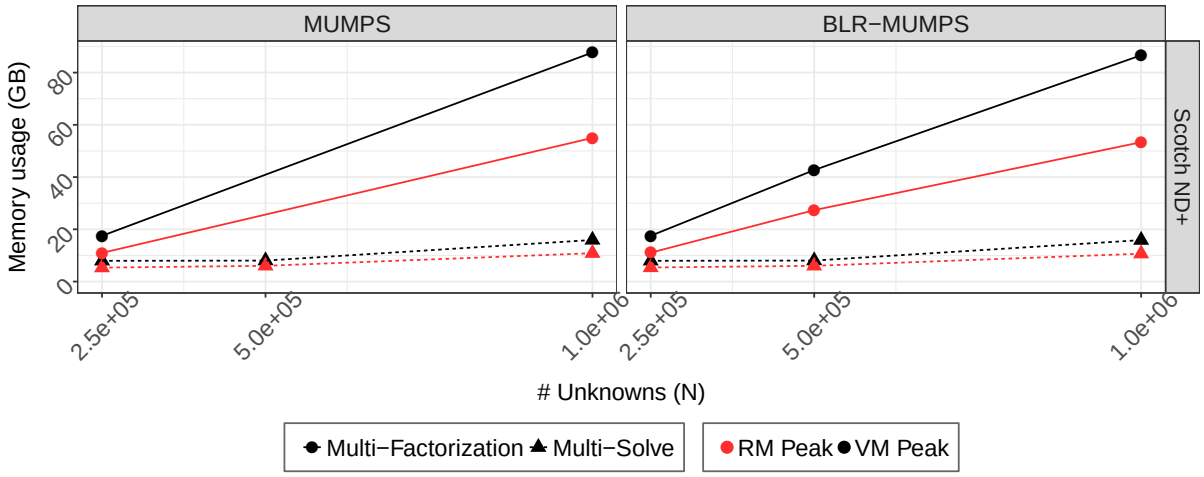
The solution of the FEM/BEM coupling using either the Multi-Solve and Multi-Factorization techniques relies on the decomposition of the overall matrix A into the same 2×2 matrix from Eq. (3.2). On the one hand, the solution of the FEM/BEM coupling using Multi-Solve (introduced in § 1.4.4.1.2) relies on the decomposition of the computation of the Schur complement $\mathcal{S} = A_{ss} - A_{sv}A_{vv}^{-1}A_{vs}$ (see Eq. (1.30), p. 74) into several solves over groups of 32 columns of the matrix A_{vs}. On the other hand, the solution of the FEM/BEM coupling using Multi-Factorization (introduced in § 1.4.4.1.3) directly computes blocks of the Schur complement following the out-of-core division of the A_{ss} submatrix (see § 1.4.4.1.1). The factorization of the sparse matrix A_{vv} is computed by the sparse direct MUMPS with or without low-rank compression, whereas the factorization of the dense Schur complement is computed by the out-of-core dense direct solver SPIDO (without compression). Fig. 3.21 shows a comparison of the Multi-Factorization and Multi-Solve methods. The Multi-Solve technique requires less time and memory for the computation of the overall solution of the FEM/BEM coupling. However, the blocking size n_b of the out-of-core partition has not been changed here and is chosen automatically between 200 and 800 by the solver. This size impacts a lot the Multi-Factorization arithmetic complexity as demonstrated in § 1.4.4.1. On other problems with different characteristics, for example a different ratio between the number of unknowns in the FEM mesh n_{FEM} and the BEM mesh n_{BEM}, the trend could change.

3.6.3 Comparison of $\mathcal{H}$-Matrix and Multi-Solve Methods

We focus on the comparison of the Multi-Solve technique on one side, using MUMPS with or without the BLR feature, and the HMAT solver with variants using recursive bisection, using BBox ND, and using BBox ND and CV-HSF. Due to implementation issues, we are unable to produce results for the HMAT solver with nested dissection (and symbolic factorization) run in parallel for large problems (above 1M unknowns). We also obtain a numerical error (a forward error too large) when running the Multi-Solve technique sequentially. With these considerations in mind, we establish here a parallel comparison between the Multi-Solve technique and the HMAT solver on a small problem of 500000 unknowns in Table 3.12. The total factorization time presented here is the overall time necessary for the computation of the factorized coupling matrix A. As we have already established previously, the purely multi-threaded scalability of $\mathcal{H}$-Matrices is better here than for MUMPS (e.g., with only one MPI process). Therefore, the trends that we observe here may be different in a sequential context or using multiple MPI processes. A first observation we can make is the larger forward error obtained with HMAT compared to the low-rank version of MUMPS, i.e., BLR-MUMPS (with the same value for the compression threshold parameter ϵ). Also, the factorization of the FEM-FEM matrix A_{vv} is faster with MUMPS than with HMAT using nested dissection, with here a factor of about 2, confirming the trends observed in § 3.5.5. However, the computation



(a) Factorization time.



(b) Real and virtual memory peaks.

Figure 3.21: FEM/BEM coupling solution using MUMPS combined with SPIDO. Long pipe test case (Fig. 3.1b). Parallel runs with 24 threads on `miriel`.

of the Schur complement and the factorization of the resulting BEM-BEM matrix are both faster using HMAT (with or without nested dissection) than using the Multi-Solve technique. This leads to an overall lower factorization time for HMAT compared to the Multi-Solve technique. Finally, the solution using $\mathcal{H}$-Matrices with recursive bisection is also slower than the one using nested dissection, mainly due to the reduction of the time required for the factorization of the matrix A_{vv}.

Method	Sparse Technique	Clustering	Total Factor.	A_{vv} Factor.	Schur Comput.	Schur Factor.	Forward Error
HMAT	No SF	Bisection	903.33	284.46	452.29	33.51	$9.33 \cdot 10^{-4}$
HMAT	CV-HSF	BBox ND	542.11	52.42	410.88	33.15	$1.12 \cdot 10^{-3}$
Multi-Solve	MUMPS	Scotch ND+	3,879.28	20.81	2,937.73	919.09	$3.26 \cdot 10^{-14}$
Multi-Solve	BLR-MUMPS	Scotch ND+	3,894.23	21.20	2,954.11	917.41	$1.83 \cdot 10^{-6}$

Table 3.12: Time (s) required for the computation of major steps in the solution of a FEM/BEM coupling using HMAT and Multi-Solve methods and forward error as a control of the solution. The compression threshold is set to $\epsilon = 10^{-4}$. The Multi-Solve technique relies on the out-of-core solver SPIDO (§ 1.4.4.1.1) for the A_{ss} factorization. Long pipe (Fig. 3.1b) with 500000 unknowns. Parallel runs using 24 threads on `miriel`.

We also compare the sequential versions of the HMAT solver against parallel executions with 24 threads of the Multi-Solve technique in Table 3.13 for a problem with 1 million unknowns. At this scale, the HMAT solver still largely benefits from the usage

Method	Sparse Technique	Clustering	#t	Total Factor.	A_{vv} Factor.	Schur Comput.	Schur Factor.	Forward Error
HMAT	No SF	Bisection	1	13,354.11	3,694.44	9,235.46	395.02	$7.40 \cdot 10^{-4}$
HMAT	No SF	BBox ND	1	11,067.36	1,059.70	9,599.45	394.01	$7.40 \cdot 10^{-4}$
HMAT	CV-HSF	BBox ND	1	9,959.82	988.74	8,564.37	392.62	$4.73 \cdot 10^{-3}$
Multi-Solve	MUMPS	Scotch ND+	24	15,553.47	61.21	11,203.32	4,285.24	$3.61 \cdot 10^{-14}$
Multi-Solve	BLR-MUMPS	Scotch ND+	24	14,110.15	56.10	9,599.62	4,451.03	$2.09 \cdot 10^{-8}$

Table 3.13: Time (s) required for the computation of major steps in the solution of a FEM/BEM coupling using HMAT and Multi-Solve methods and forward error as a control of the solution. The compression threshold is set to $\epsilon = 10^{-4}$. The Multi-Solve technique relies on the out-of-core solver SPIDO (§ 1.4.4.1.1) for the A_{ss} factorization. Long pipe (Fig. 3.1b) with 1M unknowns. Multi-thread and sequential runs on 24 cores on `miriel`. #t is the number of threads.

of compression for the factorization of the BEM-BEM matrix A_{ss} compared to the factorization computed by SPIDO in the Multi-Solve technique, by a factor of 10. The computation of the Schur complement seems to require a time fairly equivalent among all the methods, though the Multi-Solve technique using MUMPS without compression

takes a little more time than the other methods, and the HMAT solution using CV-HSF takes a little less time. The computation of the factorization of the sparse matrix A_{vv} is a lot faster using MUMPS though we have to keep in mind that the HMAT solver is here sequential. Among the HMAT variants, we can see that the usage of nested dissection reduces the time required for the factorization of A_{vv} by a factor of 3, while the usage of symbolic factorization reduces a bit more this factorization time. Overall, the HMAT solver is less costly than the Multi-Solve variant. Among the HMAT variants, the usage of nested dissection leads to a lower time requirement for the global factorization of A and symbolic factorization reduces furthermore this requirement.

Concerning comparisons of the memory of the HMAT solver with the Multi-Solve technique, SPIDO relying on out-of-core techniques, we cannot equitably measure memory consumption of the RAM.

The accuracy of the HMAT solver is sufficient for our application and no iterative refinement is therefore used, as mentioned earlier, though the use of $\mathcal{H}$-Matrices as preconditioners may be investigated in the future.

There are multiple conclusions to this section.

- We confirm here that the solution using $\mathcal{H}$-Matrix (based on recursive bisection) drastically reduces the factorization time of the dense submatrix A_{ss} compared to a dense direct solver.
- The hierarchical factorization of the sparse matrix A_{vv} has been proven to be faster using nested dissection and symbolic factorization (compared to recursive bisection and no symbolic factorization), though it does not quite reach the performance of a sparse solver.
- The usage of nested dissection in the $\mathcal{H}$-Matrix solver lowers only the factorization time of the sparse subsystem involved in the coupling, whereas the symbolic factorization reduces the time required to compute the sparse factorization as well as the computation of the Schur complement.
- Finally, the overall solution using $\mathcal{H}$-Matrices is faster than using the Multi-Solve technique (relying on sparse and dense direct solvers MUMPS and SPIDO).

Conclusion

In this thesis, we have studied the combination of hierarchical methods of compression ($\mathcal{H}$-Matrices) with reordering techniques (nested dissection, AMF) and symbolic analysis for the computation of sparse linear systems arising in the coupling between the Finite Element Method (FEM) and the Boundary Element Method (BEM). To this end, we have studied the introduction of the nested dissection for the clustering of unknowns used in the construction of $\mathcal{H}$-Matrices, either using geometric or topological information, and confirmed previous results from the $\mathcal{H}$-Matrix community. We have also proposed new ways of preventing the tall & skinny blocks arising from the use of nested dissection in a $\mathcal{H}$-Matrix. The main contributions of this thesis are the introduction of new algorithms based on the use of symbolic analysis in a hierarchical framework. We group these methods into two classes. The first class of methods performs a Hierarchical Symbolic Factorization with the aim of providing a way to use the sparsity of the structure of non-zeros in a sparse matrix to avoid the storage of zeros in a $\mathcal{H}$-Matrix. The second class of methods aims at clustering the separators computed by the nested dissection following an Interactions-Aware strategy relying on the symbolic information of the interactions between the unknowns.

We have first provided a background of this thesis in Chapter 1 by detailing the industrial context in which it took place as well as the linear systems arising from this context. We have then discussed the solution of these linear systems following their dense or sparse characteristics and proposed to study the solution of the FEM/BEM coupling using two methods, one relying on direct solvers MUMPS and SPIDO and the other relying on $\mathcal{H}$-Matrices. We have primarily focused on the optimization of the second method (based on $\mathcal{H}$-Matrices) and used the first one (based on MUMPS and SPIDO) mainly as a reference.

The first method is based on the solution of the FEM/BEM coupling using a Schur complement and relying on the MUMPS and SPIDO solvers to handle sparse and dense factorizations, respectively. We rely on the Multi-Solve technique to compute the Schur complement (performing a Gaussian elimination of the unknowns associated with the FEM mesh and updating the corresponding unknowns associated with the BEM) by computing multiple solves to construct the Schur complement by block columns.

The other method is based on $\mathcal{H}$-Matrices. We have studied variants of this method relying on geometric nested dissection or topological nested dissection, as well as variants using symbolic factorization. The nested dissection is a heuristic reordering procedure aiming at the reduction of fill-in. It is essential for the solution of sparse matrices by sparse

direct full-rank solvers. On the other hand, the hierarchical compression of $\mathcal{H}$-Matrices also reduces the impact of fill-in of a sparse matrix ordered with recursive bisection. We have presented in § 2.3.1 different clusterings based on the nested dissection method and have confirmed results of previous studies led on the subject [102, 105, 107, 126, 138, 193] concluding that $\mathcal{H}$-Matrices relying on efficient reordering techniques (such as the nested dissection) require less time and memory for the computation of a factorization of a sparse matrix. In the test cases used in this thesis, our geometric implementation of a nested dissection seems to lead to a better compression than our topological approach when using the standard strong admissibility condition used in the literature, with the possible drawback of leading to an unbalanced cluster tree. Also, nested dissection leads to the formation of tall & skinny blocks of which we have presented multiple ways of prevention in § 2.3.2.

Once the nested dissection has been validated in the hierarchical framework, we have focused on the efficiency of the compression of the submatrices of a sparse matrix. We have therefore presented in § 2.3.4 another admissibility condition for the factorization of sparse matrices as well as an oracle determining the optimal compression to use on a pre-computed hierarchical structure. Results show that other admissibility conditions better suited for sparse problems should be investigated in the future.

The search for a new admissibility condition also revealed that a lot of zeros were still present in the hierarchical structure of the solver and that we could simply avoid the storage of null submatrices (filled with only zeros) by performing a symbolic analysis of the matrix before numerical factorization. We have thus presented in § 2.4.2 ways to compute a symbolic factorization in a hierarchical context, i.e., performing a Hierarchical Symbolic Factorization, and shown in § 3.5.2 the beneficial effect of its computation on the numerical factorization of a sparse matrix using $\mathcal{H}$-Matrices relying on a nested dissection clustering.

In a parallel development of the hierarchical symbolic factorization, we have also investigated the clustering of separators arising from the nested dissection, in an effort to maximize the effectiveness of the symbolic factorization and correctly identify the location of the non-zeros and the fill-in and be able to separate zeros from non-zeros properly. To this end, we have provided in § 2.4.3 multiple leads to the computation of an appropriate separator clustering. We have shown in § 3.5.1 that an Interactions-Aware separator clustering may benefit to topological nested dissections. However we have also shown in this section that the geometric separator clustering does not need to rely on such a Interactions-Aware technique. Indeed, its geometric clustering matches the clustering of the (geometric) nested dissection and already provide an efficient block structure in the resulting $\mathcal{H}$-Matrix.

In addition the techniques discussed above, another method has been developed that was meant to take advantage of both the hierarchical symbolic factorization and the interactions-aware techniques. We have indeed introduced a way to use vertex-wise symbolic information on column clusters to subdivide the leaves of a $\mathcal{H}$-Matrix matching the underlying non-zero pattern of the sparse matrix. Experiments show promising results leading to a storage closer to the one of a sparse direct solver, such as presented in § 3.5.5.

$\mathcal{H}$-Matrices have a complexity that should become more and more beneficial as the size of problem gets larger. To be able to observe large-scale behavior of this hierarchical solver, we have discussed in §§ 3.5.3 and 3.5.4 the parallelization of the solver and its scalability on larger sparse problems.

Finally, we have presented methods to efficiently solve a FEM/BEM coupling in § 1.4, which is the broader (and long-term) goal of this work. Preliminary results of the solution of the FEM/BEM coupling are shown in § 3.6. The effects of reordering techniques and symbolic factorization also impact the solution of the FEM/BEM coupling though the benefits are less perceptible due to the lower cost of the sparse factorization compared to other parts of the computations (mainly the computation of the Schur complement). To position the hierarchical solver to a solution using direct solvers MUMPS (sparse) and SPIDO (dense), we have compared the $\mathcal{H}$-Matrices and the Multi-Solve technique and shown that $\mathcal{H}$-Matrices require less time for the computation of the solution of the FEM/BEM coupling. More specifically, the solution using $\mathcal{H}$-Matrices is faster when relying on both the nested dissection and the symbolic factorization.

Perspectives

In this manuscript, we have studied many leads on the solution of sparse linear systems and the FEM/BEM coupling, mainly relying on $\mathcal{H}$-Matrices and sparse techniques such as the nested dissection and symbolic factorization. However, many other challenges are yet to be tackled and we now discuss some developments that may be investigated in the future.

Symbolic factorization on a unique $\mathcal{H}$-Matrix for the FEM/BEM coupling Due to the current implementation of the $\mathcal{H}$-Matrix solver, the overall matrix involved in the FEM/BEM coupling has a 2×2 block structure and is therefore subdivided into four independent submatrices. To be able to use the whole graph associated with the coupling, one could think of constructing a unique $\mathcal{H}$-Matrix of which the first stage of clustering would separate the unknowns associated with the FEM and the unknowns associated with the BEM. This would lead to the possibility of using symbolic information associated with the off-diagonal BEM-FEM matrix (the (2,1) block) in the coupling and use the same methods used on the sparse FEM-FEM matrix (the (1,1) block) to exploit the underlying sparsity pattern of the matrix to reduce memory and time consumption.

Optimization of the ordering of off-diagonal blocks The local clustering based on vertex-wise symbolic information leads to the same kind of problems studied by the Sparse-Direct community. The reduction of off-diagonal blocks via other orderings should therefore be investigated. The current implementation also does not yet support the usage of vertex-wise symbolic information on separator-separator blocks. Implementing and optimizing the corresponding operations would lead to a reduction of the memory consumption of the solver (better matching the underlying non-zero pattern of the sparse

matrix) but should also lead to a reduction of the number of operations computed by the solver.

Investigation of the Flop and time performances The hierarchical solver currently has a moderated efficiency in terms of the number of computed floating-point operations per second, i.e., the flop rate (Flop/s). It is lower than the usual flop rate of a sparse direct solver in our experiments. This may come from a cache inefficiency due to the order of the performed operations. The written and read data constantly changes positions, following a z-curve (Morton order). To avoid this issue, accumulating updates [44] may be a solution, though the hierarchical and low-rank framework, as well as the task-based implementation, could complicate the situations that need to be taken into account.

Admissibility condition suited for sparse linear systems As we have mentioned in the second chapter and earlier in this conclusion, our experiments have shown that the current strong admissibility condition we have relied upon in this thesis leads to more memory and time consumption of the hierarchical factorization compared to the optimal compression computed by an oracle. Other criteria should be studied to determine what are the characteristics specific to sparse linear systems that may be used for low-rank compression.

Other clustering approaches Finally, we have not investigated hybrid clusterings relying on topological information at a global level to reduce the fill-in generated by the factorization of a sparse matrix, and geometric information in separators to take advantage of a better compression efficiency. The computation of an efficient nested dissection based on geometric information, leading to a reduced fill-in equivalent to that generated in a topological nested dissection, should also be studied. Regarding the solution of the FEM/BEM coupling, one could also investigate a clustering for the BEM aware of the (nested dissection) clustering of the FEM.

Appendix A

SCOTCH Ordering Strategies

The strategy used in this thesis for computing a nested dissection only (meaning no local ordering, neither in separators nor local subdomains) is defined in Algorithm A.1.

Regarding the specifics of this strategy, we detail here the main parameters useful to the comprehension of the strategy and refer the interested reader to SCOTCH User's Guide [162]. Following this manual, the function `n` computes a nested dissection, here using the separation strategy defined by the function `sep` for leaves with more than 120 vertices, as defined by the (first) `vert` parameter. It will try to coarsen ("multilevel" method `m`) the two subdomains of the nested dissection until they reach a size of 100 vertices, as defined by the (other) `vert` parameter.

Under this threshold, a local ordering may be used. The strategy used on each separator is defined in the parameter `ose`. In Algorithm A.1, it is defined as the method `s`, short for 'simple', which orders the unknowns in their natural order. In Algorithm A.2, it is defined as the method `g`, short for the Gibbs-Poole-Stockmeyer method [96]. This method is a greedy bipartitioning approach that aims at reducing the number of off-diagonal blocks. The strategy used on local subdomains smaller than the threshold is defined in `ole` as `f`, designating the Block Halo Approximate Minimum Fill method. Other parameters include `cmin` and `cmax` respectively setting a minimum or maximum number of columns for each column block/supernode.

Algorithm A.1: SCOTCH Nested Dissection strategy

```
n{sep=/((vert)>120)?m{rat=0.7,
                      vert=100,
                      low=h{pass=10},
                      asc=b{width=3,
                            bnd=f{bal=0.01},
                            org=(|h{pass=10})f{bal=0.01}}
                      };,
  ole=s,ose=s}
```

Algorithm A.2: SCOTCH Nested Dissection + AMF strategy

```
c{rat=0.7,
cpr=n{sep=/(vert>120)?m{rat=0.8,"
                        "vert=100,"
                        "low=h{pass=10},"
                        "asc=f{bal=0.2}}|"
                      "m{rat=0.8,"
                        "vert=100,"
                        "low=h{pass=10},"
                        "asc=f{bal=0.2}};,"
      "ole=f{cmin=20,cmax=100000,frat=0.08},"
      "ose=g},"
unc=n{sep=/(vert>120)?(m{rat=0.8,"
                         "vert=100,"
                         "low=h{pass=10},"
                         "asc=f{bal=0.2}})|"
                       "m{rat=0.8,"
                         "vert=100,"
                         "low=h{pass=10},"
                         "asc=f{bal=0.2}};,"
      "ole=f{cmin=20,cmax=100000,frat=0.08},"
      "ose=g}}"
```

Algorithm A.3: SCOTCH Nested Dissection strategy used in MUMPS

```
"n{sep=m{asc=b{width=3,strat=q{strat=f}},"
        "low=q{strat=h},vert=1000,dvert=100,dlevl=0,"
        "proc=1,seq=q{strat=m{type=h,vert=100,"
        "low=h{pass=10},asc=b{width=3,bnd=f{bal=0.2},"
        "org=h{pass=10}f{bal=0.2}}}},ole=s,ose=s,osq=s}"
```

The strategy used in the MUMPS solver is defined Algorithm A.3.

Finally, the transformation of SCOTCH's output elimination tree (describing the dependencies between clusters) into a cluster tree may be computed by copying each node i into one of its children (ordered last, i.e., the separator) and extending i as the union of its children (separator included). This is further discussed in Chapter 2, p. 84, supported by Fig. 2.1, p. 85.

Appendix B

Examples of Matrices using Different Orderings

We give here examples of symmetric $\mathcal{H}$-Matrices obtained on a 2000-unknowns problem relying on the CV-HSF strategy to subdivide off-diagonal blocks resulting from non-separator subdomains with separators. The clusterings presented in § 2.3.1 are used. An example with the geometric nested dissection (the BBox ND method) is shown in Fig. B.1. Examples with the two topological nested dissections, i.e., the Scotch ND and Scotch ND+ methods, are shown in Fig. B.3 and B.5, respectively. Note that the Minimum Fill method (Scotch ND+) subdivides diagonal blocks in multiple smaller blocks. We have not investigated the usage of a smaller threshold value for N_{ND} for the BBox ND method though this should be investigated in the future. In terms of compression, BBox ND is already exhibiting signs of compression at this relatively small scale, while Scotch ND+ shows less compression and Scotch ND seems to not compress at all. Fig. B.2, B.4 and B.6 also show the impact of the ordering on the assembled matrix A_{sv} in a FEM/BEM coupling with 8000 unknowns (6482 FEM unknowns and 1518 BEM unknowns). The sizes used here are different for visibility considerations.

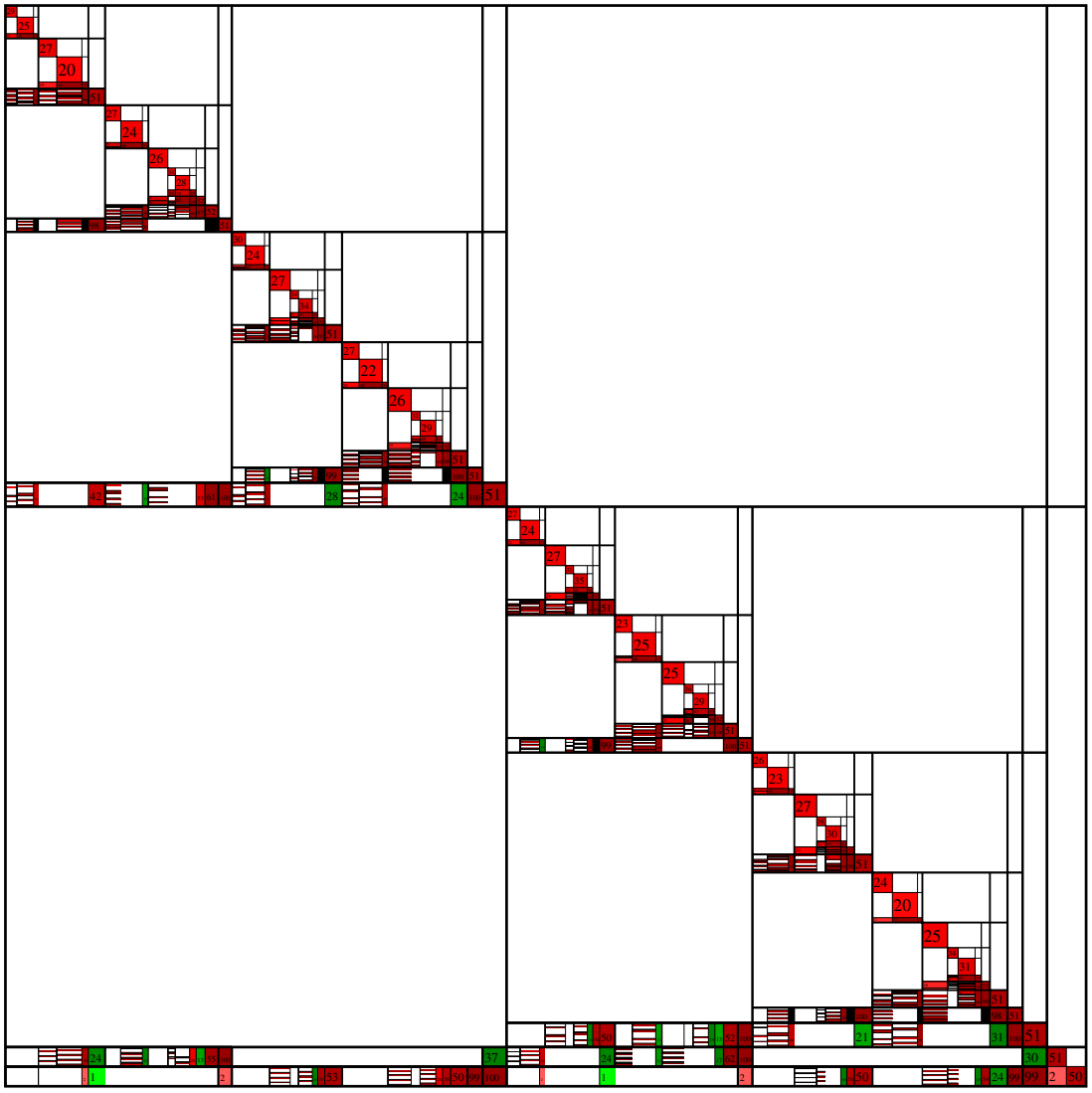


Figure B.1: Example of CV-HSF applied on BBox ND for a purely FEM problem with 2000 unknowns.

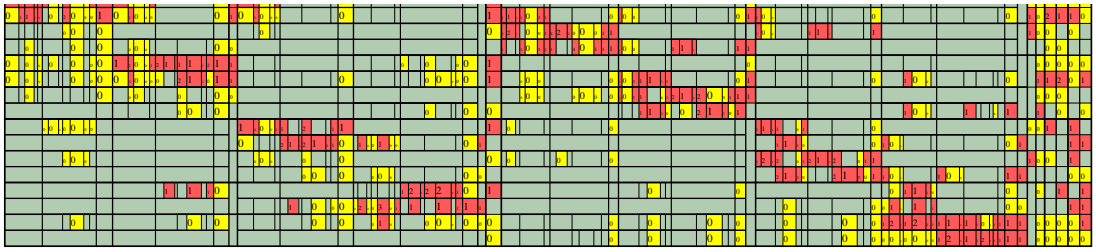


Figure B.2: Example of BEM-FEM matrix A_{sv} storage using BBox ND for the FEM mesh in a FEM/BEM coupling with 8000 unknowns.

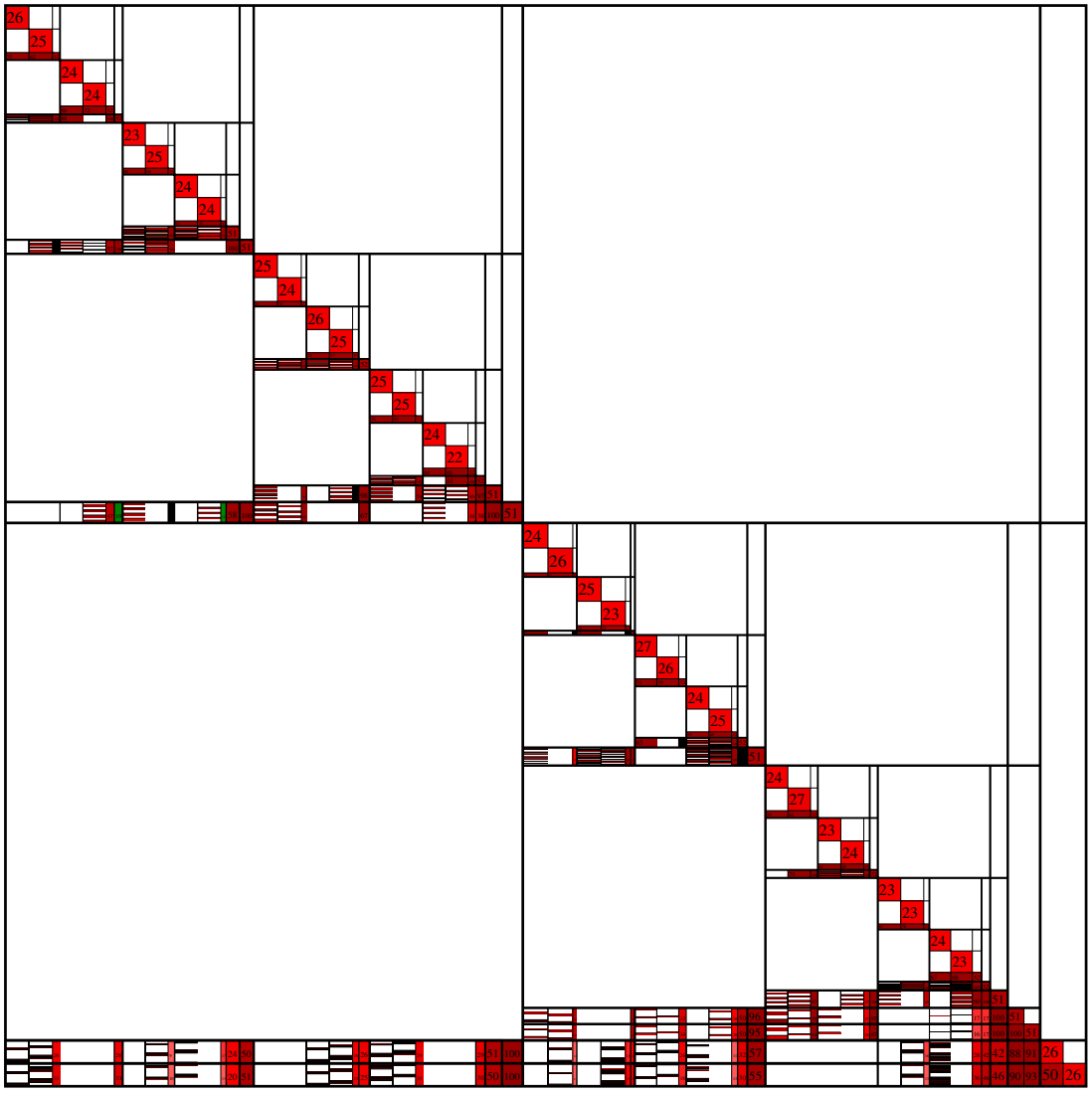


Figure B.3: Example of CV-HSF applied on Scotch ND for a purely FEM problem with 2000 unknowns.

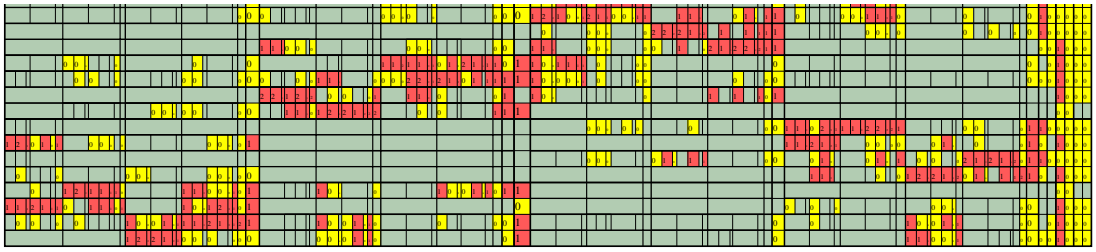


Figure B.4: Example of BEM-FEM matrix A_{sv} storage using Scotch ND for the FEM mesh in a FEM/BEM coupling with 8000 unknowns.

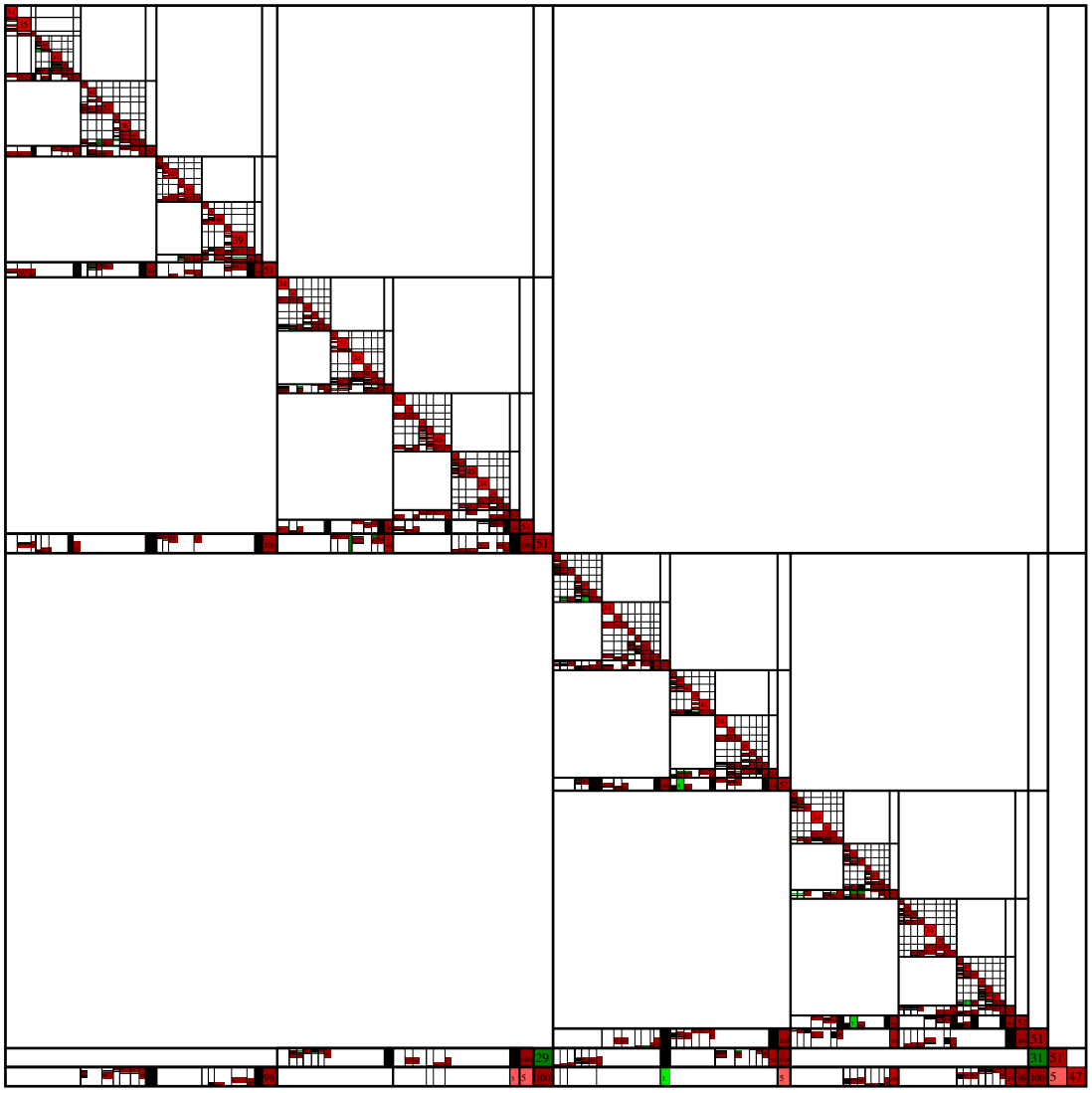


Figure B.5: Example of CV-HSF applied on Scotch ND+ for a purely FEM problem with 2000 unknowns.

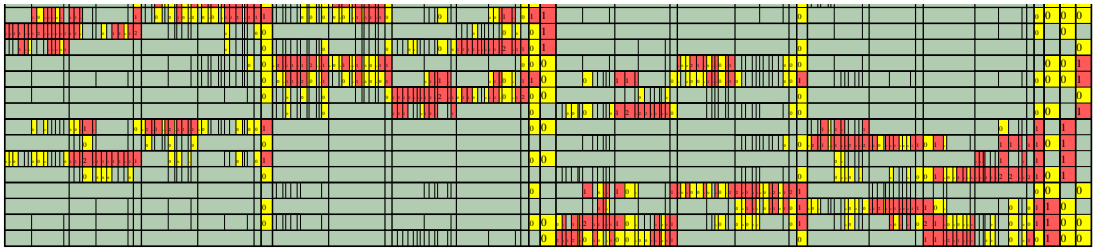


Figure B.6: Example of BEM-FEM matrix A_{sv} storage using Scotch ND+ for the FEM mesh in a FEM/BEM coupling with 8000 unknowns.

Bibliography

[1] Another software library on hierarchical matrices for elliptic differential equations (AHMED). http://bebendorf.ins.uni-bonn.de/AHMED.html. (Pages 21).

[2] GENCI. http://www.genci.fr/fr/content/calculateurs-et-centres-de-calcul. (Pages 157).

[3] hmat-oss. https://github.com/jeromerobert/hmat-oss. (Pages 21).

[4] $\mathcal{H}$-Lib Pro. http://www.hlibpro.com/. (Pages 21).

[5] $\mathcal{H}^2$-Lib. http://h2lib.org. (Pages 21).

[6] PlaFRIM. https://plafrim.bordeaux.inria.fr/. (Pages 157).

[7] ToyRT. https://github.com/BenoitLBen/runtime. (Pages 73).

[8] Rachid Ababou, Amvrossios C. Bagtzoglou, and Eric F. Wood. On the condition number of covariance matrices in kriging, estimation, and simulation of random fields. *Mathematical Geology*, 26(1):99–133, 1994. (Pages 12).

[9] Emmanuel Agullo, Patrick R. Amestoy, Alfredo Buttari, Abdou Guermouche, Jean-Yves L'Excellent, and François-Henry Rouet. Robust memory-aware mappings for parallel multifrontal factorizations. *SIAM Journal on Scientific Computing*, 38(3):C256–C279, 2016. (Pages 185).

[10] Emmanuel Agullo, Eric Darve, Luc Giraud, and Yuval Harness. Low-Rank Factorizations in Data Sparse Hierarchical Algorithms for Preconditioning Symmetric Positive Definite Matrices. *SIAM Journal on Matrix Analysis and Applications*, 39(4):1701–1725, October 2018. (Pages 42).

[11] Emmanuel Agullo, Aurélien Falco, Luc Giraud, and Guillaume Sylvand. Vers une factorisation symbolique hiérarchique de rang faible pour des matrices creuses. In *Conférence d'informatique en Parallélisme, Architecture et Système (ComPAS'17)*, Sophia Antipolis, France, June 2017. (Pages 96).

[12] Airbus Group Innovations. *Actipole User Manual*, 2015. (Pages 15 and 74).

[13] G. Alléon, M. Benzi, and L. Giraud. Sparse Approximate Inverse Preconditioning for Dense Linear Systems Arising in Computational Electromagnetics. *Numerical Algorithms*, 16:1–15, 1997. (Pages 15 and 71).

[14] Sivaram Ambikasaran. *Fast algorithms for dense numerical linear algebra and applications*. PhD thesis, Stanford University Stanford, 2013. (Pages 41 and 42).

[15] Sivaram Ambikasaran and Eric Darve. An $\mathcal{O}(N \log N)$ Fast Direct Solver for Partial Hierarchically Semi-Separable Matrices. *Journal of Scientific Computing*, 57(3):477–501, 2013. (Pages 42).

[16] Sivaram Ambikasaran and Eric Darve. The inverse fast multipole method. *arXiv preprint arXiv:1407.1572*, 2014. (Pages 41 and 42).

[17] Patrick R. Amestoy, Cleve Ashcraft, Olivier Boiteau, Alfredo Buttari, Jean-Yves L'Excellent, and Clément Weisbecker. Improving multifrontal methods by means of block low-rank representations. *SIAM Journal on Scientific Computing*, 37(3):A1451–A1474, 2015. (Pages 42, 84, and 94).

[18] Patrick R. Amestoy, Alfredo Buttari, Jean-Yves L'Excellent, and Theo Mary. On the Complexity of the Block Low-Rank Multifrontal Factorization. *SIAM Journal on Scientific Computing*, 39(4):A1710–A1740, 2017. (Pages 42, 84, and 94).

[19] Patrick R. Amestoy, Alfredo Buttari, Jean-Yves L'Excellent, and Theo Mary. Bridging the gap between flat and hierarchical low-rank matrix formats: the multilevel BLR format. 2018. (Pages 42).

[20] Patrick R. Amestoy, Timothy A. Davis, and Iain S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, 1996. (Pages 50).

[21] Patrick R. Amestoy and Iain S. Duff. Vectorization of a multiprocessor multifrontal code. *The International Journal of Supercomputing Applications*, 3(3):41–59, 1989. (Pages 71).

[22] Patrick R. Amestoy, Iain S. Duff, and J-Y L'Excellent. MUMPS multifrontal massively parallel solver version 2.0. 1998. (Pages 159).

[23] Patrick R. Amestoy, Iain S. Duff, and J-Y L'excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Computer methods in applied mechanics and engineering*, 184(2):501–520, 2000. (Pages 44 and 71).

[24] Patrick R. Amestoy, Iain S. Duff, Jean-Yves L'Excellent, and Jacko Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001. (Pages 71).

[25] Amirhossein Aminfar, Sivaram Ambikasaran, and Eric Darve. A fast block low-rank dense solver with applications to finite-element matrices. *Journal of Computational Physics*, 304:170–188, 2016. (Pages 42, 84, and 94).

[26] Edward Anderson, Zhaojun Bai, Christian Bischof, Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, S. Hammerling, Alan McKenney, et al. *LAPACK Users' guide*, volume 9. SIAM, 1999. (Pages 13, 15, and 16).

[27] C. Ashcraft Anton, J. and C. Weisbecker. A Block Low-Rank multithreaded factorization for dense BEM operators. In *SIAM Conference on Parallel Processing (SIAM PP16). Paris, France*, 2016. (Pages 42).

[28] Walter Edwin Arnoldi. The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Quarterly of applied mathematics*, 9(1):17–29, 1951. (Pages 14).

[29] Cleve Ashcraft and Roger Grimes. The influence of relaxed supernode partitions on the multifrontal method. *ACM Transactions on Mathematical Software (TOMS)*, 15(4):291–309, 1989. (Pages 56).

[30] Cédric Augonnet. *Scheduling Tasks over Multicore machines enhanced with Accelerators: a Runtime System's Perspective*. PhD thesis, Université Bordeaux 1, 351 cours de la Libération — 33405 TALENCE cedex, December 2011. (Pages 73).

[31] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. Rapport de recherche RR-7240, INRIA, March 2010. (Pages 73).

[32] Guillaume Bal, Joseph B. Keller, George Papanicolaou, and Leonid Ryzhik. Transport theory for acoustic waves with reflection and transmission at interfaces. *Wave Motion*, 30(4):303 – 327, 1999. (Pages 5).

[33] Prasanta Kumar Banerjee and Roy Butterfield. *Boundary element methods in engineering science*, volume 17. McGraw-Hill London, 1981. (Pages 4 and 9).

[34] L. Banjai and W. Hackbusch. Hierarchical matrix techniques for low-and high-frequency Helmholtz problems. *IMA journal of numerical analysis*, 28(1):46–79, 2008. (Pages 21 and 156).

[35] M. Bebendorf. Hierarchical LU Decomposition-based Preconditioners for BEM. *Computing*, 74:225–247, 2005. (Pages 15 and 21).

[36] M. Bebendorf. Adaptive Cross Approximation of Multivariate Functions. *Constructive Approximation*, 34:149–179, 2011. (Pages 23 and 24).

[37] M. Bebendorf and S. Rjasanow. Adaptive Low-Rank Approximation of Collocation Matrices. *Computing*, 70:1–24, 2003. (Pages 23 and 24).

[38] Mario Bebendorf. Approximation of boundary element matrices. *Numerische Mathematik*, 86(4):565–589, 2000. (Pages 23, 24, and 35).

[39] Mario Bebendorf. *Hierarchical matrices*. Springer, 2008. (Pages 21 and 156).

[40] I. Benedetti, MH Aliabadi, and G. Davi. A fast 3D dual boundary element method based on hierarchical matrices. *International journal of solids and structures*, 45(7):2355–2376, 2008. (Pages 21 and 156).

[41] Susan Blackford and Jack Dongarra. Installation guide for LAPACK. Technical Report 41, LAPACK Working Note, June 1999. originally released March 1992. (Pages 15).

[42] Steffen Börm. Directional $\mathcal{H}^2$-matrix compression for high-frequency problems. *Numerical Linear Algebra with Applications*, 24(6):e2112. e2112 nla.2112. (Pages 21).

[43] Steffen Börm. Data-sparse approximation of non-local operators by $\mathcal{H}^2$-matrices. *Linear algebra and its applications*, 422(2):380–403, 2007. (Pages 21 and 41).

[44] Steffen Börm. Hierarchical matrix arithmetic with accumulated updates. *ArXiv e-prints*, March 2017. (Pages 134 and 203).

[45] Steffen Börm and Lars Grasedyck. Low-rank approximation of integral operators by interpolation. *Computing*, 72(3):325–332, 2004. (Pages 35).

[46] Steffen Börm and Lars Grasedyck. Hybrid cross approximation of integral operators. *Numerische Mathematik*, 101:221–249, 2005. (Pages 24).

[47] Steffen Börm, Lars Grasedyck, and Wolfgang Hackbusch. Introduction to hierarchical matrices with applications. *Engineering Analysis with Boundary Elements*, 27(5):405 – 422, 2003. Large scale problems using BEM. (Pages 21, 28, and 156).

[48] Steffen Börm, Lars Grasedyck, and Wolfgang Hackbusch. Hierarchical Matrices. Technical report, Max Planck Institut für Mathematik, 2004. (Pages 21 and 156).

[49] A. Le Bot. A vibroacoustic model for high frequency analysis. *Journal of Sound and Vibration*, 211(4):537 – 554, 1998. (Pages 5).

[50] Susanne Brenner and Ridgway Scott. *The mathematical theory of finite element methods*, volume 15. Springer Science & Business Media, 2007. (Pages 4 and 9).

[51] Arline L. Bronzaft and Dennis P. McCarthy. The effect of elevated train noise on reading ability. *Environment and behavior*, 7(4):517–528, 1975. (Pages 6).

[52] James R. Bunch and Linda Kaufman. Some stable methods for calculating inertia and solving symmetric linear systems. *Mathematics of computation*, pages 163–179, 1977. (Pages 16).

[53] Léopold Cambier, Chao Chen, Erik G. Boman, Sivasankaran Rajamanickam, Raymond S. Tuminaro, and Eric Darve. An Algebraic Sparsified Nested Dissection Algorithm Using Low-Rank Approximations. *arXiv preprint arXiv:1901.02971*, 2019. (Pages 91 and 144).

[54] B. Carpentieri. *Sparse preconditioners for dense linear systems from electromagnetic applications*. PhD thesis, 2002. (Pages 15 and 71).

[55] B. Carpentieri, I. S. Duff, and L. Giraud. Sparse pattern selection strategies for robust Frobenius-norm minimization preconditioners in electromagnetism. *Numerical Linear Algebra with Applications*, 7(7-8):667–685, 2000. (Pages 15 and 71).

[56] Fabien Casenave. Aéroacoustique, couplage BEM-FEM 3D pour la simulation du bruit rayonné par un turboréacteur. Technical report, 2010. (Pages 9).

[57] Fabien Casenave. *Méthodes de réduction de modèles appliquées à des problèmes d'aéroacoustique résolus par équations intégrales*. PhD thesis, Université Paris-Est, 2013. (Pages 10).

[58] Fabien Casenave, Alexandre Ern, and Guillaume Sylvand. Coupled BEM–FEM for the convected Helmholtz equation with non-uniform flow in a bounded domain. *Journal of Computational Physics*, 257:627–644, 2014. (Pages 8 and 9).

[59] Jeffrey N Chadwick and David S Bindel. An efficient solver for sparse linear systems based on rank-structured Cholesky factorization. *arXiv preprint arXiv:1507.05593*, 2015. (Pages 84 and 94).

[60] Shiv Chandrasekaran, Ming Gu, and Timothy Pals. A fast ULV decomposition solver for hierarchically semiseparable representations. *SIAM Journal on Matrix Analysis and Applications*, 28(3):603–622, 2006. (Pages 21 and 41).

[61] P. Charrier and J. Roman. Algorithmique et calculs de complexité pour un solveur de type dissections emboîtées. *Numerische Mathematik*, 55(4):463–476, 1989. (Pages 50, 56, 62, 64, 65, 66, 117, and 129).

[62] Cédric Chevalier and François Pellegrini. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel Computing*, 34(6):318–331, 2008. (Pages 32, 51, 97, and 98).

[63] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 1–6. ACM, 1987. (Pages 16).

[64] Eric Darve. *Méthodes multipôles rapides: Résolution des équations de Maxwell par formulations intégrales*. PhD thesis, 1999. (Pages 20).

[65] Eric Darve. The fast multipole method I: Error analysis and asymptotic complexity. *SIAM Journal on Numerical Analysis*, 38(1):98–128, 2000. (Pages 20).

[66] Eric Darve. The fast multipole method: numerical implementation. *Journal of Computational Physics*, 160(1):195–240, 2000. (Pages 20).

[67] Timothy A. Davis. Algorithm 832: UMFPACK V4. 3—an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software (TOMS)*, 30(2):196–199, 2004. (Pages 71).

[68] Timothy A. Davis. *Direct methods for sparse linear systems*, volume 2. SIAM, 2006. (Pages 43, 44, 57, and 58).

[69] Timothy A. Davis, Sivasankaran Rajamanickam, and Wissam Sid-Lakhdar. A survey of direct methods for sparse linear systems. Technical report, Texas A&M University, Department of Computer Science and Engineering, 2016. (Pages 43, 58, and 70).

[70] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph WH Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Analysis and Applications*, 20(3):720–755, 1999. (Pages 53, 55, 56, 61, 63, 64, 70, 121, and 132).

[71] James W. Demmel, Nicholas J. Higham, and Robert S. Schreiber. Block LU Factorization. Technical Report 40, Lapack Working Notes, February 1992. (Pages 19).

[72] James W. Demmel, Nicholas J. Higham, and Robert S. Schreiber. Stability of block LU factorization. *Numerical Linear Algebra with Applications*, 2(2):173 – 190, 1995. (Pages 19).

[73] Jack Dongarra. Freely available software for linear algebra on the web. 2016. (Pages 70).

[74] Jack Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990. (Pages 15).

[75] Jack Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. Van der Vorst. *Numerical linear algebra for high-performance computers*, volume 7. SIAM, 1998. (Pages 43).

[76] Iain S. Duff, Albert Maurice Erisman, and John K. Reid. *Direct methods for sparse matrices*. Oxford University Press, 2017. (Pages 43, 57, and 70).

[77] Iain S. Duff and John K. Reid. The multifrontal solution of indefinite sparse symmetric linear. *ACM Transactions on Mathematical Software (TOMS)*, 9(3):302–325, 1983. (Pages 70).

[78] Iain S. Duff and John K. Reid. The multifrontal solution of unsymmetric sets of linear equations. *SIAM Journal on Scientific and Statistical Computing*, 5(3):633–641, 1984. (Pages 70).

[79] Carl Eckart and Gale Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218, 1936. (Pages 22).

[80] Stanley C. Eisenstat, Howard C. Elman, and Martin H. Schultz. Variational iterative methods for nonsymmetric systems of linear equations. *SIAM Journal on Numerical Analysis*, 20(2):345–357, 1983. (Pages 14).

[81] Stanley C. Eisenstat and Joseph WH Liu. The theory of elimination trees for sparse unsymmetric matrices. *SIAM Journal on Matrix Analysis and Applications*, 26(3):686–705, 2005. (Pages 44).

[82] Alexandre Ern and Jean-Luc Guermond. *Theory and practice of finite elements*, volume 159. Springer Science & Business Media, 2013. (Pages 4 and 9).

[83] Gary W. Evans, Staffan Hygge, and Monika Bullinger. Chronic noise and psychological stress. *Psychological Science*, 6(6):333–338, 1995. (Pages 6).

[84] Aurélien Falco, Emmanuel Agullo, Luc Giraud, and Guillaume Sylvand. Hierarchical Symbolic Factorization for Sparse Matrices. In *Sparse Days 2018*, Toulouse, France, September 2018. (Pages 96).

[85] Yuwei Fan, Lin Lin, Lexing Ying, and Leonardo Zepeda-Núnez. A multiscale neural network based on hierarchical matrices. *arXiv preprint arXiv:1807.01883*, 2018. (Pages 21).

[86] Mathieu Faverge. *Static-Dynamic Hybrid Scheduling in sparse linear algebra for large clusters of NUMA and multi-cores architectures*. Theses, Université Sciences et Technologies - Bordeaux I, December 2009. (Pages 56, 62, and 64).

[87] I. D. Fernando, S. Jayasena, M. Fernando, and H. Sundar. A Scalable Hierarchical Semi-Separable Library for Heterogeneous Clusters. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 513–522, Aug 2017. (Pages 21 and 41).

[88] Roland W Freund, Gene H Golub, and Noël M Nachtigal. Iterative solution of linear systems. *Acta numerica*, 1:57–100, 1992. (Pages 14).

[89] Roland W Freund, Martin H Gutknecht, and Noël M Nachtigal. An implementation of the look-ahead Lanczos algorithm for non-Hermitian matrices. *SIAM journal on scientific computing*, 14(1):137–158, 1993. (Pages 14).

[90] Alan George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973. (Pages iv, 2, and 50).

[91] Alan George and Joseph WH Liu. Computer solution of large sparse positive definite. 1981. (Pages 44).

[92] Alan George and Joseph WH Liu. The evolution of the minimum degree ordering algorithm. *Siam review*, 31(1):1–19, 1989. (Pages 50).

[93] Alan George, Joseph WH Liu, and Esmond Ng. Computer solution of sparse linear systems. *Academic, Orlando*, 1994. (Pages 49, 56, 58, 66, and 117).

[94] Alan George and Esmond Ng. Symbolic factorization for sparse Gaussian elimination with partial pivoting. *SIAM Journal on Scientific and Statistical Computing*, 8(6):877–898, 1987. (Pages 57).

[95] Pieter Ghysels, Xiaoye S. Li, François-Henry Rouet, Samuel Williams, and Artem Napov. An efficient multicore implementation of a novel HSS-structured multifrontal solver using randomized sampling. *SIAM Journal on Scientific Computing*, 38(5):S358–S384, 2016. (Pages 84, 94, and 110).

[96] Norman E. Gibbs, William G. Poole Jr, and Paul K. Stockmeyer. A comparison of several bandwidth and profile reduction algorithms. Technical report, College of William and Mary Williamsburg VA, 1975. (Pages 131 and 204).

[97] John R. Gilbert and Joseph WH Liu. Elimination structures for unsymmetric sparse lu factors. *SIAM Journal on Matrix Analysis and Applications*, 14(2):334–352, 1993. (Pages 44, 58, and 66).

[98] John R. Gilbert and Robert Endre Tarjan. The analysis of a nested dissection algorithm. *Numerische mathematik*, 50(4):377–404, 1986. (Pages 50).

[99] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48, 1991. (Pages 13).

[100] Gene H Golub and Charles F Van Loan. *Matrix computations*, volume 3. JHU Press, 2012. (Pages 17, 22, 26, and 27).

[101] Sergei A Goreinov, Eugene E Tyrtyshnikov, and Nickolai L Zamarashkin. A theory of pseudoskeleton approximations. *Linear algebra and its applications*, 261(1-3):1–21, 1997. (Pages 22 and 23).

[102] L. Grasedyck, R. Kriemann, and S. Le Borne. Domain decomposition based $\mathcal{H}$-LU preconditioning. *Numerische Mathematik*, 112:565–600, 2009. (Pages 84, 86, 88, 110, 156, and 201).

[103] Lars Grasedyck. Adaptive Recompression of $\mathcal{H}$-Matrices for BEM. *Computing*, 74:205–223, 2005. (Pages 23).

[104] Lars Grasedyck and Wolfgang Hackbusch. Construction and Arithmetics of $\mathcal{H}$-Matrices. *Computing*, 70:295–334, 2003. 10.1007/s00607-003-0019-1. (Pages 37).

[105] Lars Grasedyck, Wolfgang Hackbusch, and Ronald Kriemann. Performance of $\mathcal{H}$-LU preconditioning for sparse matrices. *Computational Methods in Applied Mathematics Comput. Methods Appl. Math.*, 8(4):336–349, 2008. (Pages 84, 88, 180, and 201).

[106] Lars Grasedyck, Wolfgang Hackbusch, and Sabine Le Borne. Adaptive geometrically balanced clustering of $\mathcal{H}$-matrices. *Computing*, 73(1):1–23, 2004. (Pages 28).

[107] Lars Grasedyck, Ronald Kriemann, and Sabine Le Borne. Parallel black box $\mathcal{H}$-LU preconditioning for elliptic boundary value problems. *Computing and Visualization in Science*, 11:273–291, 2008. 10.1007/s00791-008-0098-9. (Pages 15, 21, 32, 84, 88, 95, 110, 156, and 201).

[108] Joseph F Grcar. Mathematicians of Gaussian elimination. *Notices of the AMS*, 58(6):782–792, 2011. (Pages 16).

[109] Leslie Greengard and Vladimir Rokhlin. A fast algorithm for particle simulations. *Journal of computational physics*, 73(2):325–348, 1987. (Pages 20).

[110] Laura Grigori, James W. Demmel, and Xiaoye S. Li. Parallel symbolic factorization for sparse LU with static pivoting. *SIAM Journal on Scientific Computing*, 29(3):1289–1314, 2007. (Pages 58 and 66).

[111] Abdou Guermouche, Jean-Yves L'Excellent, and Gil Utard. Impact of reordering on the memory of a multifrontal solver. *Parallel computing*, 29(9):1191–1218, 2003. (Pages 50).

[112] W. Hackbusch. A sparse matrix arithmetic based on $\mathcal{H}$-matrices. part I: Introduction to $\mathcal{H}$-matrices. *Computing*, 62:89–108, 1999. 10.1007/s006070050015. (Pages iv, 2, 5, 13, 21, 41, 84, and 156).

[113] W. Hackbusch and B. N. Khoromskij. A Sparse $\mathcal{H}$-Matrices Arithmetic. *Computing*, 64:21–47, 2000. (Pages 35).

[114] Wolfgang Hackbusch. *Hierarchical matrices: Algorithms and analysis*, volume 49. Springer, 2015. (Pages 21, 29, 35, 36, 37, 39, 41, 83, 86, 92, 103, 106, and 156).

[115] Wolfgang Hackbusch. Survey on the technique of hierarchical matrices. *Vietnam Journal of Mathematics*, 44(1):71–101, 2016. (Pages 88).

[116] Wolfgang Hackbusch and Steffen Börm. $\mathcal{H}^2$-matrix approximation of integral operators by interpolation. *Applied Numerical Mathematics*, 43(1):129–143, 2002. (Pages 21 and 41).

[117] Wolfgang Hackbusch, Boris Khoromskij, and Stefan A. Sauter. On $\mathcal{H}^2$-Matrices. In *Lectures on Applied Mathematics: Proceedings of the Symposium Organized by the Sonderforschungsbereich 438 on the Occasion of Karl-Heinz Hoffmann's 60th Birthday, Munich, June 30–July 1, 1999*, page 9. Springer Science & Business Media, 2000. (Pages 41).

[118] Anna L. Hansell, Marta Blangiardo, Lea Fortunato, Sarah Floud, Kees de Hoogh, Daniela Fecht, Rebecca E. Ghosh, Helga E. Laszlo, Clare Pearson, Linda Beale, et al. Aircraft noise and cardiovascular disease near Heathrow airport in London: small area study. *BMJ*, 347:f5432, 2013. (Pages 6).

[119] P. Hénon, P. Ramet, and J. Roman. PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. *Parallel Computing*, 28(2):301–321, January 2002. (Pages 51, 61, 62, 70, 114, 145, and 156).

[120] Riccardo Hertel, Sven Christophersen, and Steffen Börm. Large-scale magnetostatic field calculation in finite element micromagnetics with $\mathcal{H}^2$-matrices. *arXiv preprint arXiv:1811.05731*, 2018. (Pages 74).

[121] Magnus R. Hestenes and Eduard Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, 46(6):409–436, December 1952. (Pages 14).

[122] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Number 48. SIAM, 2002. (Pages 13).

[123] Peter Hollingsworth and David Sulitzer. Investigating the Potential of Using Quota Count as a Design Metric. *Journal of Aircraft*, 48(6):1894–1902, 2011. (Pages 6).

[124] Alexander Hrennikoff. Solution of problems of elasticity by the framework method. *Journal of applied mechanics*, 8(4):169–175, 1941. (Pages 9).

[125] Pascal Hénon and Yousef Saad. A parallel multistage ILU factorization based on a hierarchical graph decomposition. *SIAM Journal on Scientific Computing*, 28(6):2266–2293, 2006. (Pages 114).

[126] Ilgis Ibragimov, Sergej Rjasanow, and Katharina Straube. Hierarchical Cholesky decomposition of sparse matrices arising from curl–curl-equation. *Journal of Numerical Mathematics JNMA*, 15(1):31–57, 2007. (Pages 84, 88, 91, 156, and 201).

[127] Mathias Jacquelin, Yili Zheng, Esmond Ng, and Katherine Yelick. An asynchronous task-based fan-both sparse Cholesky solver. *arXiv preprint arXiv:1608.00044*, 2016. (Pages 70).

[128] P. H. Jia, J. Hu, Y. Chen, R. Zhang, L. Lei, and Z. Nie. $\mathcal{H}$-Matrices Compressed Multiplicative Schwarz Preconditioner for Nonconformal FEM-BEM-DDM. *IEEE Transactions on Antennas and Propagation*, PP(99):1–1, 2018. (Pages 73).

[129] Jian-Ming Jin. *The finite element method in electromagnetics*. John Wiley & Sons, 2015. (Pages 9).

[130] Claes Johnson and J. Claude Nedelec. On the coupling of boundary integral and finite element methods. *Mathematics of computation*, pages 1063–1079, 1980. (Pages 73).

[131] Douglas Samuel Jones. Acoustic and electromagnetic waves. *Oxford/New York, Clarendon Press/Oxford University Press, 1986, 764 p.*, 1986. (Pages 8).

[132] George Karypis and Vipin Kumar. METIS-unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995. (Pages 32, 51, and 98).

[133] George Karypis and Vipin Kumar. Multilevel k-way Partitioning Scheme for Irregular Graphs. *Journal of Parallel and Distributed computing*, 48(1):96–129, 1998. (Pages 131).

[134] N. Kishore Kumar and Jan Schneider. Literature survey on low rank approximation of matrices. *Linear and Multilinear Algebra*, 65(11):2212–2244, 2017. (Pages 22 and 23).

[135] Wai Yip Kong, James Bremer, and Vladimir Rokhlin. An adaptive fast direct solver for boundary integral equations in two dimensions. *Applied and Computational Harmonic Analysis*, 31(3):346–369, 2011. (Pages 21 and 42).

[136] Ronald Kriemann. *Parallele Algorithmen für $\mathcal{H}$-Matrizen*. Dissertation, Universität Kiel, 2004. (Pages 21).

[137] Ronald Kriemann. Parallel $\mathcal{H}$-Matrices Arithmetics on Shared Memory Systems. *Computing*, 74:273–297, 2005. 10.1007/s00607-004-0102-2. (Pages 21 and 41).

[138] Ronald Kriemann. $\mathcal{H}$-LU factorization on many-core systems. *Computing and Visualization in Science*, 16(3):105–117, 2013. (Pages 28, 84, 88, 89, 91, 95, 110, 130, 156, and 201).

[139] D.-H Kwon., R.J. Burkholder, and P.H. Pathak. Ray analysis of electromagnetic field build-up and quality factor of electrically large shielded enclosures. *IEEE Transactions on electromagnetic compatibility*, 40(1):19 – 26, 1998. (Pages 5).

[140] Xavier Lacoste. *Scheduling and memory optimizations for sparse direct solver on multi-core/multi-GPU duster systems*. PhD thesis, Bordeaux, 2015. (Pages 43, 56, 57, 62, and 64).

[141] Jun Lai, Sivaram Ambikasaran, and Leslie F. Greengard. A fast direct solver for high frequency scattering from a large cavity in two dimensions. *SIAM Journal on Scientific Computing*, 36(6):B887–B903, 2014. (Pages 21 and 42).

[142] Cornelius Lanczos. *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*. United States Governm. Press Office Los Angeles, CA, 1950. (Pages 14).

[143] Sabine Le Borne, Lars Grasedyck, and Ronald Kriemann. Domain-decomposition Based $\mathcal{H}$-LU Preconditioners. In *Domain decomposition methods in science and engineering XVI*, pages 667–674. Springer, 2007. (Pages iv, 2, 84, 88, and 156).

[144] T.H. Lehman. A statistical theory of electromagnetic fields in complex cavities. Technical report, Mai 1993. (Pages 5).

[145] John G. Lewis and Horst D. Simon. The impact of hardware gather/scatter on sparse Gaussian elimination. *SIAM Journal on Scientific and Statistical Computing*, 9(2):304–311, 1988. (Pages 70).

[146] Jean-Yves L'Excellent. *Multifrontal Methods: Parallelism, Memory Usage and Numerical Aspects*. Habilitation à diriger des recherches, Ecole normale supérieure de Lyon - ENS LYON, September 2012. (Pages 5, 44, 57, 58, and 70).

[147] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, 1989. (Pages 73).

[148] Yingzhou Li, Haizhao Yang, Eileen R Martin, Kenneth L Ho, and Lexing Ying. Butterfly factorization. *Multiscale Modeling & Simulation*, 13(2):714–732, 2015. (Pages 21).

[149] M. Lintner. The eigenvalue problem for the 2D Laplacian in $\mathcal{H}$-matrix arithmetic and application to the heat and wave equation. *Computing*, 72(3):293–323, 2004. (Pages 86).

[150] Richard J Lipton, Donald J Rose, and Robert Endre Tarjan. Generalized nested dissection. *SIAM journal on numerical analysis*, 16(2):346–358, 1979. (Pages 61).

[151] Joseph WH Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software (TOMS)*, 11(2):141–153, 1985. (Pages 50).

[152] Joseph WH Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11(1):134–172, 1990. (Pages 47).

[153] Joseph WH Liu. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM review*, 34(1):82–109, 1992. (Pages 70).

[154] Benoît Lizé. Solveur direct haute performance. Technical report, EADS IW/École centrale Paris, 2009. (Pages 75).

[155] Benoît Lizé. *Résolution Directe Rapide pour les Éléments Finis de Frontière en Électromagnétisme et Acoustique : $\mathcal{H}$-Matrices. Parallélisme et Applications Industrielles.* PhD thesis, Université Paris 13, 2014. (Pages 9, 21, 24, 26, 29, 37, 39, 41, and 178).

[156] Thomas A Manteuffel. An incomplete factorization technique for positive definite linear systems. *Mathematics of computation*, 34(150):473–497, 1980. (Pages 71).

[157] Theo Mary. *Block Low-Rank multifrontal solvers: complexity, performance, and scalability.* PhD thesis, Université de Toulouse, 2017. (Pages 42, 84, 94, 110, 156, and 159).

[158] Bruce H McDonald and Alvin Wexler. Finite-element solution of unbounded field problems. *IEEE Transactions on microwave theory and techniques*, 20(12):841–847, 1972. (Pages 73).

[159] Esmond Ng and Padma Raghavan. Performance of greedy ordering heuristics for sparse Cholesky factorization. *SIAM Journal on Matrix Analysis and Applications*, 20(4):902–914, 1999. (Pages 50).

[160] Youness Noumir. *Une analyse haute fréquence des équations de l'aéroacoustique : étude mathématique et simulations numériques.* PhD thesis, Université Paris 13, 2011. (Pages 5).

[161] Seymour Parter. The use of linear graphs in Gauss elimination. *SIAM review*, 3(2):119–130, 1961. (Pages 45).

[162] François Pellegrini. Scotch and libscotch 6.0 user's guide. *Bacchus team, INRIA Bordeaux Sud-Ouest Technical Report: CNRS*, 5800, 2012. (Pages 95 and 204).

[163] Francois Pellegrini, Jean Roman, and Patrick Amestoy. Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering. *Concurrency: Practice and Experience*, 12(2-3):69–84, 2000. (Pages 51).

[164] Grégoire Pichon. *On the use of low-rank arithmetic to reduce the complexity of parallel sparse linear solvers based on direct factorization techniques.* PhD thesis, Université de Bordeaux, November 2018. (Pages 64, 84, 94, 95, 131, 144, and 147).

[165] Grégoire Pichon, Eric Darve, Mathieu Faverge, Pierre Ramet, and Jean Roman. Sparse supernodal solver using block low-rank compression: Design, performance and analysis. *Journal of computational science*, 27:255–270, 2018. (Pages 84, 94, and 110).

[166] Grégoire Pichon, Mathieu Faverge, Pierre Ramet, and Jean Roman. Reordering strategy for blocking optimization in sparse linear solvers. *SIAM Journal on Matrix Analysis and Applications*, 38(1):226–248, 2017. (Pages 95).

[167] Naji Qatanani and Monika Schulz. Analytical and numerical investigation of the fredholm integral equation for the heat radiation problem. *Applied Mathematics and Computation*, 175(1):149 – 170, 2006. (Pages 4 and 9).

[168] P.A. Raviart and J.M. Thomas. A mixed finite element method for 2-nd order elliptic problems. In Ilio Galligani and Enrico Magenes, editors, *Mathematical Aspects of Finite Element Methods*, volume 606 of *Lecture Notes in Mathematics*, pages 292–315. Springer Berlin Heidelberg, 1977. (Pages 4 and 9).

[169] Vladimir Rokhlin. Diagonal forms of translation operators for the Helmholtz equation in three dimensions. *Applied and Computational Harmonic Analysis*, 1(1):82–93, 1993. (Pages 20).

[170] J. Roman. Calculs de complexité relatifs à une méthode de dissection emboîtée. *Numerische Mathematik*, 47(2):175–190, Jun 1985. (Pages 84 and 85).

[171] Edward Rothberg and Stanley C Eisenstat. Node selection strategies for bottom-up sparse matrix ordering. *SIAM Journal on Matrix Analysis and Applications*, 19(3):682–695, 1998. (Pages 50).

[172] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003. (Pages 13, 14, and 72).

[173] Yousef Saad and Martin H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing*, 7(3):856–869, 1986. (Pages 14 and 75).

[174] Yousef Saad and Henk A. Van Der Vorst. Iterative solution of linear systems in the 20th century. *Journal of Computational and Applied Mathematics*, 123(1):1–33, 2000. (Pages 14).

[175] Stefan A. Sauter and Christoph Schwab. *Boundary Element Methods*, pages 183–287. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. (Pages 4 and 9).

[176] Olaf Schenk and Klaus Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Generation Computer Systems*, 20(3):475–487, 2004. (Pages 70).

[177] Robert Schreiber. A new implementation of sparse Gaussian elimination. *ACM Transactions on Mathematical Software (TOMS)*, 8(3):256–276, 1982. (Pages 47 and 70).

[178] A. Schwarzenberg-Czerny. On matrix factorization and efficient least squares solution. *Astronomy and Astrophysics Supplement Series*, 110:405, 1995. (Pages 16).

[179] Marc Sergent, David Goudin, Samuel Thibault, and Olivier Aumage. Controlling the Memory Subscription of Distributed Applications with a Task-Based Runtime System. In *21st International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Chicago, United States, May 2016. (Pages 42).

[180] Mohamed Wissam Sid Lakhdar. *Scaling the solution of large sparse linear systems using multifrontal methods on hybrid shared-distributed memory architectures*. PhD thesis, Lyon, École normale supérieure, 2014. (Pages 95).

[181] Jiming Song, Cai-Cheng Lu, and Weng Cho Chew. Multilevel fast multipole algorithm for electromagnetic scattering by large complex objects. *IEEE Transactions on Antennas and Propagation*, 45(10):1488–1493, 1997. (Pages 20).

[182] Stephen A. Stansfeld and Mark P. Matheson. Noise pollution: non-auditory effects on health. *British medical bulletin*, 68(1):243–257, 2003. (Pages 6).

[183] Volker Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969. (Pages 16).

[184] Guillaume Sylvand. *La Méthode Multipôle Rapide en Électromagnétisme. Performances, Parallélisation, Applications*. PhD thesis, ENPC / CERMICS, 2002. (Pages 9, 15, 21, and 72).

[185] Christopher KW Tarn. Computational aeroacoustics: issues and methods. *AIAA journal*, 33(10):1788–1796, 1995. (Pages 5).

[186] Sivan Toledo, Doron Chen, and Vladimir Rotkin. TAUCS: A library of sparse linear solvers, 2003. (Pages 71).

[187] A. Wang, N. Vlahopoulos, and K. Wu. Development of an energy boundary element formulation for computing high-frequency sound radiation from incoherent intensity boundary conditions. *Journal of Sound and Vibration*, 278(1-2):413 – 436, 2004. (Pages 4 and 9).

[188] Clément Weisbecker. *Improving multifrontal solvers by means of algebraic Block Low-Rank representations*. PhD thesis, Institut National Polytechnique de Toulouse (INP Toulouse), 2013. (Pages 42, 84, 94, and 95).

[189] James Hardy Wilkinson. *Rounding errors in algebraic processes*. Courier Corporation, 1994. (Pages 13).

[190] Jianlin Xia. Randomized sparse direct solvers. *SIAM Journal on Matrix Analysis and Applications*, 34(1):197–227, 2013. (Pages 84 and 94).

[191] Jianlin Xia, Shivkumar Chandrasekaran, Ming Gu, and Xiaoye S. Li. Superfast multifrontal method for large structured linear systems of equations. *SIAM Journal on Matrix Analysis and Applications*, 31(3):1382–1411, 2009. (Pages iv, 2, 84, and 94).

[192] Jianlin Xia, Shivkumar Chandrasekaran, Ming Gu, and Xiaoye S. Li. Fast algorithms for hierarchically semiseparable matrices. *Numerical Linear Algebra with Applications*, 17(6):953–976, 2010. (Pages 42).

[193] M. Yang, R. Liu, H. Gao, and X. Sheng. On the$\mathcal{H}$-LU-Based Fast Finite Element Direct Solver for 3-D Scattering Problems. *IEEE Transactions on Antennas and Propagation*, 66(7):3792–3797, July 2018. (Pages 84, 88, 180, and 201).

[194] Mihalis Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic Discrete Methods*, 2(1):77–79, 1981. (Pages 50).

[195] Lexing Ying. Fast algorithms for boundary integral equations. In *Multiscale Modeling and Simulation in Science*, pages 139–193. Springer, 2009. (Pages 21).

[196] Vladimir Yotov, Marcello Remedia, Guglielmo Aglietti, and Guy Richardson. Efficient matrix randomisation methodology for reduced spacecraft models in stochastic fem-bem vibroacoustc problems. *ecssmet 2018 proceedings*, 2018. (Pages 73).

[197] David Young. Iterative methods for solving partial difference equations of elliptic type. *Transactions of the American Mathematical Society*, 76(1):92–111, 1954. (Pages 14).

[198] Fuzhen Zhang. *The Schur complement and its applications*, volume 4. Springer Science & Business Media, 2006. (Pages 74).

[199] Y. Zhao and J. Mao. Equivalent Surface Impedance-Based Mixed Potential Integral Equation Accelerated by Optimized $\mathcal{H}$-Matrix for 3-D Interconnects. *IEEE Transactions on Microwave Theory and Techniques*, PP(99):1–13, 2017. (Pages 36 and 107).

[200] OC Zienkiewicz, DW Kelly, and P. Bettess. The coupling of the finite element method and boundary solution procedures. *International journal for numerical methods in engineering*, 11(2):355–375, 1977. (Pages 73).

[201] OC Zienkiewicz and RL Taylor. *The finite element method*, volume 3. McGraw-hill London, 1977. (Pages 4 and 9).