On the use of low-rank arithmetic to reduce the complexity of parallel sparse linear solvers based on direct factorization techniques

Grégoire Pichon

université de BORDEAUX

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Grégoire Pichon**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

On the use of low-rank arithmetic to reduce the complexity of parallel sparse linear solvers based on direct factorization techniques

---

Date de soutenance : 29 Novembre 2018

Après avis des rapporteurs :

Esmond Ng	Directeur de recherche, L. Berkeley Nat. Lab.
Yousef Saad	Professeur, Université du Minnesota ....

Devant la commission d'examen composée de :

Alfredo Buttari ..	Chargé de recherche, CNRS	Examinateur
Mathieu Faverge .	Maître de conférence, Bordeaux INP	Co-directeur de thèse
David Goudin ...	Ingénieur chercheur, CEA DAM - CESTA .	Invité
Gildas Kubické ..	Docteur, Responsable du lab. EMC, DGA .	Invité
Stéphane Lanteri .	Directeur de recherche, Inria	Président du jury
Esmond Ng	Directeur de recherche, L. Berkeley Nat. Lab.	Rapporteur
François Pellegrini	Professeur, Université de Bordeaux	Examinateur
Pierre Ramet ...	Maître de conférence, Université de Bordeaux	Directeur de thèse

– 2018 –

Utilisation de la compression low-rank pour réduire la complexité des solveurs creux parallèles basés sur des techniques de factorisation directes.

Résumé

La résolution de systèmes linéaires creux est un problème qui apparaît dans de nombreuses applications scientifiques, et les solveurs creux sont une étape coûteuse pour ces applications ainsi que pour des solveurs plus avancés comme les solveurs hybrides direct-itératif. Pour ces raisons, optimiser la performance de ces solveurs pour les architectures modernes est un problème critique. Cependant, les contraintes mémoire et le temps de résolution limitent l'utilisation de ce type de solveur pour des problèmes de très grande taille. Pour les approches concurrentes, par exemple les méthodes itératives, des préconditionneurs garantissant une bonne convergence pour un large ensemble de problèmes sont toujours inexistants. Dans la première partie de cette thèse, nous présentons deux approches exploitant la compression Block Low-Rank (BLR) pour réduire la consommation mémoire et/ou le temps de résolution d'un solveur creux. Ce format de compression à plat, sans hiérarchie, permet de tirer profit du caractère low-rank des blocs apparaissant dans la factorisation de systèmes linéaires creux. La solution proposée peut être utilisée soit en tant que solveur direct avec une précision réduite, soit comme un préconditionneur très robuste. La première approche, appelée **Minimal Memory**, illustre le meilleur gain mémoire atteignable avec la compression BLR, alors que la seconde approche, appelée **Just-In-Time**, est dédiée à la réduction du nombre d'opérations, et donc du temps de résolution. Dans la seconde partie, nous présentons une stratégie de reordering qui augmente la granularité des blocs pour tirer davantage profit de la localité dans l'utilisation d'architectures multi-coeurs et pour fournir de tâches plus volumineuses aux GPUs. Cette stratégie s'appuie sur la factorisation symbolique par blocs pour raffiner la numérotation produite par des outils de partitionnement comme Metis ou Scotch, et ne modifie pas le nombre d'opérations nécessaires à la résolution du problème. A partir de cette approche, nous proposons dans la troisième partie de ce manuscrit une technique de clustering low-rank qui a pour objectif de former des clusters d'inconnues au sein d'un séparateur. Nous démontrons notamment les intérêts d'une telle approche par rapport aux techniques de clustering classiquement utilisées. Ces deux stratégies ont été développées pour le format à plat BLR, mais sont également une première étape pour le passage à un format hiérarchique. Dans la dernière partie de cette thèse, nous nous intéressons à une modification de la technique de dissection emboîtée afin d'aligner les séparateurs par rapport à leur père pour obtenir des structures de données plus régulières.

Mots-clés: Solveur linéaire creux direct, compression block low-rank, parallélisme, numérotation

Discipline: Informatique

Laboratoire: Equipe-projet HiePACS, Inria Bordeaux - Sud-Ouest, 33405 Talence

On the use of low-rank arithmetic to reduce the complexity of parallel sparse linear solvers based on direct factorization techniques

Abstract

Solving sparse linear systems is a problem that arises in many scientific applications, and sparse direct solvers are a time consuming and key kernel for those applications and for more advanced solvers such as hybrid direct-iterative solvers. For those reasons, optimizing their performance on modern architectures is critical. However, memory requirements and time-to-solution limit the use of direct methods for very large matrices. For other approaches, such as iterative methods, general black-box preconditioners that can ensure fast convergence for a wide range of problems are still missing. In the first part of this thesis, we present two approaches using a Block Low-Rank (BLR) compression technique to reduce the memory footprint and/or the time-to-solution of a supernodal sparse direct solver. This flat, non-hierarchical, compression method allows to take advantage of the low-rank property of the blocks appearing during the factorization of sparse linear systems. The proposed solver can be used either as a direct solver at a lower precision or as a very robust preconditioner. The first approach, called **Minimal Memory**, illustrates the maximum memory gain that can be obtained with the BLR compression method, while the second approach, called **Just-In-Time**, mainly focuses on reducing the computational complexity and thus the time-to-solution. In the second part, we present a reordering strategy that increases the block granularity to better take advantage of the locality for multicores and provide larger tasks to GPUs. This strategy relies on the block-symbolic factorization to refine the ordering produced by tools such as Metis or Scotch, but it does not impact the number of operations required to solve the problem. From this approach, we propose in the third part of this manuscript a new low-rank clustering technique that is designed to cluster unknowns within a separator to obtain the BLR partition, and demonstrate its assets with respect to widely used clustering strategies. Both reordering and clustering where designed for the flat BLR representation but are also a first step to move to hierarchical formats. We investigate in the last part of this thesis a modified nested dissection strategy that aligns separators with respect to their father to obtain more regular data structure.

Keywords: Linear sparse direct solver, block low-rank compression, parallelism, ordering

Discipline: Computer Science

Laboratory: Equipe-projet HiePACS, Inria Bordeaux - Sud-Ouest, 33405 Talence

Acknowledgments

Tout d'abord, je souhaiterais remercier Esmond Ng et Yousef Saad, qui ont accepté de rapporter ma thèse, pour leurs retours sur le manuscrit et les questions abordées durant la thèse. Je tiens également à remercier l'ensemble des participants du jury de thèse. Je remercie les trois examinateurs: Alfredo Buttari, François Pellegrini et Stéphane Lanteri pour leurs questions et les points abordés à l'issue de la soutenance. Je remercie David Goudin pour sa participation au jury et les interactions que nous avons eues durant la thèse. Je remercie également Gildas Kubické pour sa participation à la soutenance en tant que représentant de la DGA, qui a financé ces travaux dans le contexte d'une bourse DGA/Inria.

Je tiens à remercier mes encadrants pour leur aide durant ces trois années de thèse. Je remercie Mathieu Faverge et Pierre Ramet pour m'avoir accueilli dans l'équipe et m'avoir donné l'occasion de découvrir et apprécier le monde de la recherche depuis mon premier stage en 2013. Je remercie également Jean Roman, qui a co-encadré cette thèse, notamment pour m'avoir fait découvrir les méthodes directes en dernière année d'école d'ingénieur et pour toutes ces discussions passionnantes autour de la complexité des solveurs creux. Je remercie Eric Darve qui a participé à l'encadrement de cette thèse avec un œil externe et des remarques constructives. Travailler avec vous quatre a été un vrai plaisir, et la liberté que vous m'avez accordée m'a permis d'explorer au mieux diverses pistes de recherche. J'espère continuer à interagir avec vous dans la suite de mon parcours académique.

Les travaux présentés dans la thèse ont également tiré profit de diverses collaborations. Je remercie Aurélien Esnard pour m'avoir présenté son projet StarPart et avoir participé à l'implémentation d'ALIGnATOR. Je remercie également Stéphane Lanteri pour nous avoir proposé d'intégrer la nouvelle version de PaStiX au sein de son logiciel Horse. Cela a permis de valider les développements réalisés durant la thèse sur un plus grand problème issu d'une application industrielle.

Au-delà de ces collaborations scientifiques, l'environnement de travail à l'Inria a participé au bon fonctionnement de la thèse. Je remercie l'équipe Plafrim pour leur support dans l'utilisation du supercalculateur de Bordeaux. Je remercie également notre assistante d'équipe, Chrystel Plumejeau, pour sa réactivité dans les demandes de missions et les diverses démarches administratives.

Je tiens a remercier tous les chercheurs avec lesquels j'ai pu interagir depuis mes débuts dans le monde de la recherche. Les rencontres lors de diverses visites de laboratoires ou de conférences ont été riches, aussi bien d'un point de vue scientifique qu'humain. Je remercie en particulier Azzam Haidar et Jakub Kurzak qui m'ont encadré en 2014 lors d'un stage à l'Innovative Computing Laboratory. Je re-

mercie également Hatem Ltaeif qui m'a permis de passer un mois à Kaust en 2015. J'espère continuer à faire vivre ces relations dans le futur, en construisant de solides collaborations.

Je remercie mes collègues de l'équipe HiePACS, ainsi que les anciens depuis mes débuts en 2013, qui m'ont permis de passer de bons moments. Les discussions (scientifiques ou non), les pauses café et le baby-foot ont été un bon moyen de se changer les idées. Je remercie également les équipes course à pied et natation d'HiePACS qui m'auront poussé à me remettre plus sérieusement (ou pas) au sport.

Je tiens à remercier l'ensemble de mes amis, en particulier les anciens de l'Enseirb-Matmeca, qui font de Bordeaux un bel endroit pour vivre. Je remercie également mes amis de l'Inria et ceux du Labri.

Je remercie mes parents pour leur soutien depuis mon plus jeune âge, qui m'ont permis de suivre les études que je voulais. Cette thèse est un bel exemple de leur réussite, je la leur dédie. Je remercie également mon petit frère Quentin, et lui souhaite le meilleur pour la fin de ses études. Pour finir, je remercie Léa pour son soutien au quotidien, qui a été particulièrement important durant la période de rédaction de la thèse et de préparation de la soutenance.

Résumé en français

La simulation numérique est utilisée dans de nombreux domaines scientifiques afin de simuler le comportement de systèmes complexes au lieu de réaliser des expériences coûteuses et parfois interdites, par exemple dans le domaine du nucléaire. Les machines utilisées pour réaliser ces simulations ont grandement évolué, jusqu'à pouvoir effectuer des milliards d'opérations par seconde. Cependant, les processeurs ont rapidement atteint une limite dans l'augmentation de leur fréquence à cause de la dissipation de chaleur associée. Afin de continuer à accroître la puissance des machines de calcul, des architectures multi-cœurs ont vu le jour, pour combiner la puissance de plusieurs processeurs. Programmer correctement ces architectures est un problème important, car il faut exprimer suffisamment de parallélisme pour un grand nombre d'unités de calcul tout en limitant la consommation mémoire.

De nombreuses applications scientifiques utilisent des modèles qui nécessitent de résoudre des systèmes linéaires de la forme $Ax = b$. La matrice A peut alors être considérée comme dense ou comme creuse, dans le cas où la plupart des entrées sont nulles. Les matrices creuses apparaissent notamment lors de la discrétisation d'équations aux dérivées partielles, où les interactions à longue distance sont négligées. La résolution de systèmes linéaires creux est une étape cruciale dans de nombreuses applications à cause de son coût en temps et en mémoire.

Plusieurs solutions ont été proposées afin de résoudre des systèmes linéaires creux. Dans cette thèse, nous nous intéressons aux méthodes directes qui permettent de factoriser une matrice en un produit de matrices triangulaires avant de résoudre ces systèmes triangulaires. Par rapport à d'autres approches (itératives, hybrides par exemple), l'utilisation des méthodes directes permet généralement plus de stabilité numérique dans la résolution. Cela permet notamment de résoudre des problèmes plus complexes numériquement, que les autres méthodes ne peuvent pas appréhender. Cependant, les complexités en temps et en mémoire des méthodes directes limitent leur passage à l'échelle et notamment la résolution de très grands problèmes.

Présentation du problème

Dans le contexte des solveurs directs, des travaux récents ont étudié l'impact de la compression des matrices denses ou des blocs denses apparaissant durant la factorisation de matrices creuses. L'objectif de la compression de rang faible consiste à représenter une matrice dense sous une forme plus compacte, avec ou sans perte d'information. En pratique, de nombreuses applications ne nécessitent pas une solution correcte à la précision machine, notamment à cause de l'incertitude des mesures

sur A et b. De nombreux formats de compression **low-rank** ont été proposés pour représenter une matrice dense. Dans cette thèse, on s'intéresse au format **Block Low-Rank** (BLR), qui réalise une compression à plat, contrairement aux formats hiérarchiques ($\mathcal{H}$, $\mathcal{H}^2$, HSS, HODLR...). Plus particulièrement, le format BLR découpe (**clustering**) la matrice en un ensemble de blocs plus petits. Les blocs diagonaux sont conservés denses et les blocs extra-diagonaux sont compressés sous une forme **low-rank** uv^t, obtenue avec des techniques de compression comme la décomposition en valeurs singulières (SVD).

L'objectif principal de cette thèse est d'intégrer des noyaux de compression au sein du solveur supernodal PaStiX. Ce solveur, développé depuis une vingtaine d'années, permet de résoudre des systèmes composés de centaines de millions d'inconnues sur des architectures distribuées, faites de nœuds hétérogènes. L'intérêt principal de ce solveur est qu'il se comporte comme une "boîte noire", permettant la résolution de systèmes issus de diverses applications sans avoir la connaissance de la géométrie du problème ou des équations sous-jacentes. La problématique principale de la thèse est donc d'introduire de la compression **low-rank** tout en conservant le même niveau de parallélisme afin de tirer profit des fonctionnalités développées depuis de nombreuses années et en garantissant le comportement "boîte noire" du solveur, approprié pour de nombreuses applications.

Un solveur creux est généralement divisé en quatre étapes principales:

1. Renumérotation des inconnues afin de minimiser le remplissage – les éléments nuls devenant non nuls durant la factorisation – et de maximiser le parallélisme;

2. Construction de la factorisation symbolique de la matrice, qui permet de prédire le remplissage afin d'allouer en mémoire la structure de la matrice factorisée avant toute opération numérique;

3. Factorisation de la matrice en utilisation des algorithmes par blocs;

4. Résolution de deux systèmes triangulaires.

Au terme de la factorisation symbolique, la matrice est divisée en un ensemble de supernœuds. Chaque supernœud volumineux est découpé en un ensemble de plus petits supernœuds afin d'augmenter le niveau de parallélisme.

Solveur utilisant le format Block Low-Rank

L'approche présentée dans le Chapitre 3 de la thèse consiste à remplacer les blocs extra-diagonaux suffisamment gros par des blocs **low-rank** dans la factorisation symbolique raffinée. Adapter le solveur original à cette nouvelle structure revient à remplacer les noyaux opérant sur des blocs denses par des noyaux opérant sur des blocs au format compressé. Cependant, différentes variantes de l'algorithme peuvent être obtenues en changeant le moment où les blocs sont compressés.

La première stratégie, appelée **Minimal Memory**, consiste à compresser la matrice creuse A et réalise la factorisation avec des blocs **low-rank**. Le coût d'une

partie des opérations est réduit, mais l'addition et la mise à jour de matrices **low-rank** peuvent entraîner un surcoût en temps pour les petites matrices. Avec cette stratégie, des gains en mémoire sont possibles car les blocs de la structure symbolique ne sont jamais alloués sous leur forme dense. La seconde stratégie, appelée **Just-In-Time**, compresse les blocs au plus tard pour éviter le surcoût de l'addition **low-rank**. Ainsi, lorsqu'un bloc à reçu toutes ses mises à jour, il est compressé. De cette manière l'opération de mise à jour des matrices **low-rank** vers des matrices denses est accélérée. Cependant, en compressant les blocs au plus tard, le gain mémoire sera plus faible qu'avec la stratégie précédente, voire nul si tous les blocs de la matrice ont été initialement alloués de manière dense.

Sur un ensemble de matrices comprenant environ un million d'inconnues et pour une tolérance de 10^{-8}, la stratégie **Minimal Memory** permet en moyenne de réduire la consommation mémoire d'un facteur 1.7 avec un surcoût en temps de l'ordre de 1.9. La stratégie **Just-In-Time** permet de réduire le temps de factorisation par 2. L'avantage principal de la stratégie **Minimal Memory** est le gain induit en mémoire, qui a permis de résoudre un Laplacien 3D de taille 330^3 sur un nœud avec 128 Go de RAM alors que la version originale du solveur était limitée à un problème de taille 200^3.

Renumérotation des inconnues

Une des problématiques principales dans la résolution de systèmes linéaires est la faible granularité des blocs. Dans le contexte du solveur PaStiX utilisant le format BLR, cette faible granularité augmente le nombre de mises à jour **low-rank**. Par ailleurs, la faible taille des données limite l'utilisation de machines hétérogènes, par exemple celles composées de cartes graphiques. Afin de pallier ce problème, le Chapitre 4 de cette thèse propose une technique de rénumérotation des inconnues de chaque supernœud. Une telle renumérotation peut être réalisée sans modifier le remplissage, et donc en gardant intacts le nombre d'opérations et la consommation mémoire.

La stratégie mise en place consiste à calculer des distances entre les inconnues, distance qui représente le nombre de blocs extra-diagonaux induits dans la factorisation symbolique. Par la suite, les inconnues sont renumérotées grâce un algorithme de voyageur de commerce qui minimise la distance entre les inconnues et donc le nombre de blocs extra-diagonaux associés.

Sur un grand ensemble de matrices, cette stratégie permet en moyenne de réduire d'un facteur de 40% le nombre de blocs extra-diagonaux. Sur une architecture moderne utilisant des GPUs Kepler, les temps de factorisation sont également réduits, jusqu'à un facteur de 20%.

Regroupement des inconnues

Afin de compresser correctement les supernœuds, l'utilisation d'un bon découpage, appelé **clustering**, des inconnues en blocs est nécessaire pour optimiser les taux de compression. La plupart des approches existantes dans la littérature ne considèrent que les propriétés des séparateurs issus de la dissection emboîtée, qui correspondent

aux blocs diagonaux dans la factorisation symbolique. L'approche classique consiste à regrouper les inconnues en fonction du graphe associé, dans l'objectif de créer des **clusters** de faible diamètre et ayant peu de voisins.

Dans le Chapitre 5 de cette thèse, une nouvelle stratégie de **clustering** est proposée. Son objectif est de considérer à la fois les propriétés des séparateurs (les blocs diagonaux qui seront découpés) et les interactions entre séparateurs (les blocs extra-diagonaux). Pour la stratégie **Minimal Memory**, pour laquelle le **clustering** est particulièrement important, la nouvelle stratégie permet de réduire en moyenne le temps de factorisation d'un facteur de 10%, avec un faible surcoût mémoire, de l'ordre de 5%.

Alignement des séparateurs

Les stratégies développées dans les Chapitres 4 et 5 sont nécessaires à cause du traitement algébrique du problème. En effet, la géométrie n'est pas connue et ne peut pas être exploitée par l'outil externe (Metis ou Scotch) qui calcule le partitionnement des inconnues. A cause du découpage non symétrique des inconnues et de son caractère irrégulier, il est difficile d'obtenir une bonne granularité pour les blocs. Les solveurs ayant connaissance de la géométrie du problème peuvent tirer profit de ces propriétés pour réduire les temps de calcul et améliorer les taux de compression, il est donc naturel d'essayer de se ramener dans une telle situation dans un cadre algébrique.

Le Chapitre 6 propose une nouvelle version de la dissection emboîtée, où les séparateurs sont alignés par rapport à leur séparateur père. De cette manière, les interactions sont plus régulières et limitent les problèmes vus précédemment.

Conclusions et perspectives

Au delà des contributions principales de la thèse, nous avons également étudié l'impact de l'utilisation des supports d'exécution avec des noyaux **low-rank**. PaStiX a récemment été étendu pour exprimer les algorithmes sous forme de graphes acycliques de tâches où les tâches représentent les noyaux de calcul et les arêtes les dépendances entre les calculs. Cela permet de déléguer, à un outil externe comme Parsec ou StarPU, l'ordonnancement des tâches pour maximiser le parallélisme. Étant donné que l'implémentation utilisant le format de compression BLR ne modifie pas le niveau de parallélisme du solveur, ces implémentations ont continué a fonctionner sans modification majeure. On observe, notamment avec Parsec, des gains en temps intéressants grâce à un meilleur équilibrage de charge.

Par ailleurs, la version BLR de PaStiX a été intégrée au sein de l'application HORSE, développée par l'équipe Nachos à l'Inria Sophia-Antipolis, et qui permet la résolution de problèmes d'électromagnétique avec une approche décomposition de domaine. Cela a permis de valider les résultats sur une application réelle avec des gains importants.

En combinant les différentes contributions de ce travail, la stratégie **Minimal Memory** permet de réduire la consommation mémoire d'un facteur 2 en moyenne lors de la factorisation de matrices représentatives de diverses applications, avec un

temps de factorisation proche de celui obtenu avec la version originale du solveur. Pour des problèmes de plus grande taille, la réduction du nombre d'opérations est plus importante et on observe des gains sur le temps de factorisation. La principale contribution est la résolution de systèmes qui étaient trop grands pour être traités avec la version originale de PaStiX. La principale perspective de recherche issue de ces travaux concernera le passage à un format de compression hiérarchique, qui devrait apporter des gains algorithmes supplémentaires.

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

Scientists in various fields used to conduct experiments to validate numerical models that can predict the behavior of complex systems. With the emergence of computers that perform billions of operations per second, computational science has emerged, to simulate experiments. The objective is to lower the cost of real-life experiments (wind tunnel experiments for aeronautic for instance) or to simulate experiments that are dangerous or unauthorized (nuclear experiments for instance).

Nowadays, processors have reached the limit performance for a single computational core. Indeed, the heat dissipation is now too important to keep increasing the frequency of the CPUs. In the last decades, multicore chips have been developed to increase the computational power of machines by combining several cores together. With this evolution of the number of cores, the ratio between the computational capabilities and the memory available per core keeps increasing. Programming over those parallel architectures is a challenging problem, since one has to express a sufficient level of parallelism to take advantage of many computational units, without increasing too much the memory requirements.

Many scientific applications, such as electromagnetism, astrophysics, and computational fluid dynamics, use numerical models that require solving linear systems of the form $Ax = b$. In those problems, the matrix A can either be considered as dense (almost no zero entries) or sparse (mostly zero entries). Sparse matrices appear mostly when discretizing Partial Differential Equations (PDEs) on 2D and 3D finite element or finite volume meshes. For instance, when modeling the heat equation, the continuous function is discretized into points and each point corresponds to an entry in the matrix A. In the modelization, remote interactions are often neglected, leading to a sparse system. It is the case for the heat equation modelization, where only the interactions between the closest points are considered in the model. It is important to study the resolution of sparse systems since it can enhance the simulation of the problem by 1) increasing the number of points and thus the quality of the discretization and 2) solving systems more quickly and thus allowing the simulation to perform more complex analysis, for instance when predicting the climate.

Due to the multiple structural and numerical differences that appear in sparse systems, many solutions exist to solve them in a parallel context. In this thesis, we focus on problems leading to sparse systems with a symmetric pattern and more specifically on direct methods which factorize the matrix A in either LL^t, LDL^t or LU, with L, D and U respectively unit lower triangular, diagonal, and upper triangular according to the problem numerical properties. The use of direct methods allows solving most systems due to its numerical stability, making it more reliable

than other methods. Thus, it is used in many applications or in hybrid solvers, that use internally a direct method to solve subdomains. It is often a critical kernel for those applications due to time and memory requirements. There are still limitations to solve larger and larger systems in a black-box approach without any knowledge of the geometry of the underlying problem. However, for other methods such as iterative solvers, general black-box preconditioners that can ensure fast convergence for a wide range of problems are still missing. Thus, the size or the accuracy of the problems that can be solved is always limited by the memory available on modern platforms.

In the context of direct solvers, some recent works have investigated the low-rank representations of dense matrices or dense blocks appearing during the sparse matrix factorizations. Those blocks are represented through many possible compression formats such as Block Low-Rank (BLR) or Hierarchical formats ($\mathcal{H}$, $\mathcal{H}^2$, HSS, HODLR...). The objective of those approaches is to approximate the data at a given tolerance to compute approximate solutions. Then, the solver provides an approximate solution which is good enough for many real-life applications. Indeed, for many simulations, the data used on entry (A and b) is issued from measures, such as the temperature. Its accuracy can be lower than the machine precision and an approximate solution is often sufficient. These different approaches reduce the memory requirement and/or the time-to-solution of the solvers. Depending on the compression strategy, these solvers require knowledge of the underlying geometry to tackle the problem or can do it in a purely algebraic fashion.

Two approaches may be used to perform direct factorizations: the multifrontal method and the supernodal method, leading to different schemes of contributions. In this thesis, we focus on the supernodal method, which increases the level of parallelism. The contributions presented in this thesis were developed in the PaStiX solver to take advantage of available features to manage sparsity and efficiently exploit parallelism. This library solves sparse linear systems with a direct, supernodal approach, on top of modern architectures made of distributed heterogeneous nodes. It has been developed since two decades and is able to solve systems with tens of millions of unknowns. The main objective of this thesis is to enhance the solver to reduce time-to-solution and/or memory requirements by using low-rank compression, while keeping control on the quality of the solution.

In Chapter 1, we present the background that will be used all throughout this thesis. We describe sparse direct solvers, before presenting low-rank compression techniques. Finally, we present the PaStiX solver, in which developments were realized.

In Chapter 2, we detail existing solvers that make use of low-rank compression to introduce the contributions of this thesis with respect to the literature. We also present several ordering techniques that aim at enhancing the sparse patterns or the compressibility of low-rank blocks.

In Chapter 3, we introduce the first and main contribution of this thesis. It presents how BLR compression has been introduced in the PaStiX solver. The objective of this study is to compress, at a given accuracy, data blocks appearing during the factorization, and to study how it is possible to maintain a correct accuracy in a supernodal context. It reduces both time-to-solution and memory requirements.

In Chapter 4, we study the off-diagonal blocks structure. Indeed, in a sparse context, there are many off-diagonal blocks with variable size. Increasing the granularity of those blocks should enhance the compressibility of low-rank blocks. Thus, we modify the sparse structures to increase the average blocking size, without modifying the memory requirement or the number of operations. It also reduces time-to-solution when using accelerators, which can only perform efficient operations on large data blocks.

In Chapter 5, we study strategies that perform low-rank clustering. Indeed, building set of unknowns that are well separated increases compression rates. For instance, for the heat equation, unknowns that are far-away have only few interactions. The objective is to isolate sets of unknowns such that non-compressible interactions occur between vertices of a same cluster and the interactions between clusters are low-rank. We analyze the assets and drawbacks of existing strategies and propose a new heuristic to perform algebraically the clustering for sparse matrices.

In Chapter 6, we propose a preliminary framework to enhance regularity of sparse patterns. In Chapter 4 and Chapter 5, the objective is to work on the partition provided by partitioning tools to increase compressibility. However, directly introducing some constraints on the partitioning step may impact the compression rates. Indeed, many solvers take advantage of the geometry of the problem to compute a suitable clustering and enhance compressibility. The objective of this study is to partition algebraically a sparse matrix by introducing some constraints to obtain aligned structures. Thus, one can expect to compute a better clustering of the unknowns to increase the compressibility with more regular structures.

Finally, we conclude this thesis and present some future works. In appendices, we present complementary studies around this thesis. We study the use of the new low-rank factorizations on top of runtimes systems in Appendix B. In Appendix C, we compare the full-rank version of PaStiX with the low-rank factorizations inside the domain decomposition solver Horse, to analyze the behavior of low-rank strategies in a large real-life test-case.

This material is based upon work supported by the DGA under a DGA/Inria grant. It was advised by Mathieu Faverge (Bordeaux INP) and Pierre Ramet (Université de Bordeaux), and co-advised by Eric Darve (Stanford University) and Jean Roman (Inria). Most of experiments presented in this thesis were carried out using the PlaFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine (see https://www.plafrim.fr/). Some experiments were carried out using the Occigen supercomputer (see https://www.cines.fr/calcul/materiels/occigen).

Chapter 1

Background

Designing efficient sparse direct solvers for parallel architectures has raised much interest over the past decades. More recently, the use of low-rank compression in those solvers has brought new challenges. In this chapter, we present the background that will be used later in this thesis. We start with a general presentation of sparse direct solvers in Section 1.1, before presenting low-rank compression in Section 1.2. In Section 1.3, we describe the solver used in this thesis and its specificities. Finally, we discuss the positioning of this thesis with respect to this background in Section 1.4.

1.1 Sparse direct solvers

The scientific community has developed multiple methods to efficiently solve sparse linear systems of the form $Ax = b$, where A is a sparse matrix of size n. In the sparse case, A contains few non-zero terms, because some interactions are non-existent or neglected in the modelization of the problem. For instance, a widely known problem is the heat diffusion: a continuous function is discretized with Taylor-Young approximation into points and each entry in the matrix A represents the diffusion from a point to another. Frequently, interactions between remote points are neglected in the discretization of the problem, because the influence on the overall system is small. Finally, x and b are two vectors of size n, x being the unknown of the system and b the right-hand-side (in the example it corresponds to the initial temperature on each point).

x can then be computed with the formula $x = A^{-1}b$. However, the computation of the matrix A^{-1} is generally avoided, since we have no a priori knowledge of the structure of A^{-1}, and it might require being stored in an expensive dense matrix. This computation would require too much memory and computational time, and is not numerically stable. The methods developed by the scientific community are often adapted to the characteristics and the constraints of the simulation code. The need for accuracy, the structural and numerical stability of the matrix, and its numerical complexity depend on the studied problem and the method used for the simulation.

Several methods such as direct, iterative, or hybrid methods were developed to solve sparse linear systems. The purpose of direct methods is to factorize the matrix in a product of two triangular matrices thanks to Gauss elimination. Then, solving

the system is possible by solving both triangular systems. Its main advantage is the computation of the exact solution in exact arithmetic. Direct solvers are widely used for the quality of the resulting solution, and are often used as a basic kernel in hybrid solvers. For a complete survey on direct methods, one can refer to [37, 44, 45]. Iterative methods [102] build a solution from a starting point, or guess, and refine the solution thanks to stationary or Krylov methods. While it is generally less robust than direct methods, they offer a consequent gain on memory and computational costs. Alternative methods may be used (hybrid [5, 42, 49, 97, 114], ILU [60],...) to provide a solution between direct and iterative solvers. In many cases, a direct or a hybrid method is used as a preconditioner to provide an approximate solution that will be refined by an iterative method. A recent survey by Davis et al. has been published [34] with a list of available softwares in the last section.

In this thesis, we will focus only on the use of direct methods. Depending on the matrix properties, there are at least three possible factorizations:

- $A = LU$ in the general case,
- $A = LDL^t$ if the matrix is symmetric (LDL^H for hermitian case),
- $A = LL^t$ if the matrix is symmetric positive definite (LL^H for complex case)

L is a lower triangular matrix, U an upper triangular matrix and D a diagonal matrix. For memory reasons, the factorization is always computed in place and crushes the original values of A.

After the matrix factorization, triangular solves can compute the solution, with a forward and a backward step. For instance, if the matrix is general, we perform an LU factorization, and the resulting system $LUx = b$ is solved through $Ly = b$, followed by $Ux = y$.

For the sake of simplicity, we will focus on the general case, together with a symmetric pattern. In this thesis, when a matrix A is unsymmetric, the symmetric pattern associated with $(A + A^t)$ will be used to keep the symmetric property and rely on more efficient operations. It is the case in many direct solvers, but some implementations allow handling unsymmetric structures. Algorithm 1 presents the sequential LU algorithm for a dense matrix of size n.

When the A matrix is dense, the Gauss elimination for an n unknown linear system requires a $\Theta(n^2)$ storage while the resolution requires $\Theta(n^3)$ operations. For the sparse case, the number of elements in the given matrix A is in $\Theta(n)$, which is small with respect to n^2, and the computational cost depends on the fill-in, **i.e.**, zeroes becoming non-zeroes during the factorization process.

In order to efficiently factorize and solve, preprocessing steps are required to minimize the fill-in and reorganize the sparse matrix structure. Since the unknowns can be reordered without modifying the numerical properties of the original matrix, it may help to handle efficient data structures. Those preprocessing stages allow numerical steps (factorization and solve) to be performed more efficiently.

A sparse linear solver will be composed of four steps:

1. Find a suitable ordering to minimize the fill-in, and thus reduce both time and storage requirements. In a parallel context, exhibit independence between unknowns elimination too, to express as much parallelism as possible;

Algorithm 1 Sequential column LU algorithm

```
For k ∈ [1, n] Do
  u_kk = a_kk
  For i ∈ [k + 1, n] Do
    l_ik = a_ik / u_kk
    u_ki = a_ki
  End For
  For i ∈ [k + 1, n] Do
    For j ∈ [k + 1, n] Do
      a_ij = a_ij − l_ik u_kj                    ▷ Fill-in
    End For
  End For
End For
```

2. Compute a symbolic factorization, that allows to build the structure of the factorized matrix to allocate the final factors L and U before any numerical operation;

3. Factorize the matrix in place on L and U structures;

4. Solve the system with forward and backward triangular solves.

1.1.1 Ordering step

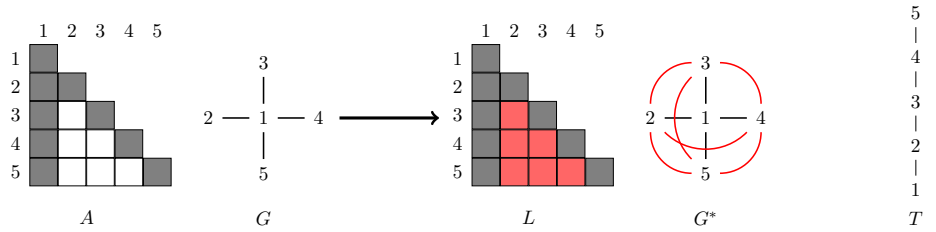
Given a matrix A of size n, we associate to A the graph $G(A) = (V, E, \sigma_p)$. The graph vertices are represented by V. Edges are represented by $E \subseteq V \times V$, and a set (x_i, x_j) belongs to E if $a_{ij} \neq 0$ or $a_{ji} \neq 0$ with $i > j$.

The main concern in sparse matrix factorizations is the fill-in: an element l_{ij} or u_{ij} can become non-zero even if the original element a_{ij} is null. Thus, it is a crucial challenge to exploit as much as possible the original sparse pattern to reduce the complexity of the factorization.

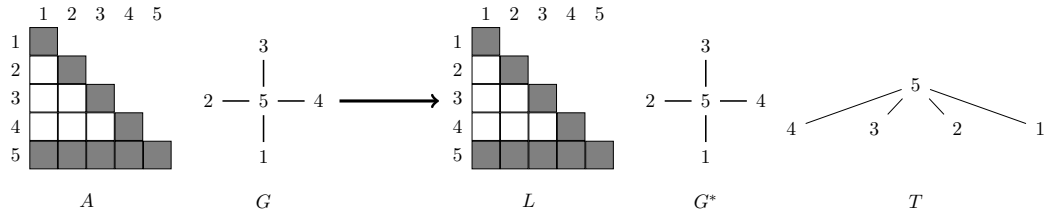
Figure 1.1(a) presents a matrix of size 5 and the original ordering of the unknowns. One can note that during the factorization, there will be fill-in for all elements, leading to a dense matrix. If a smart ordering of the unknowns is used, as presented in Figure 1.1(b), the fill-in is reduced. In this case, if unknowns 1 and 5 are inverted, there is no fill-in.

Then, the first step is to provide an ordering of the unknowns that will minimize the fill-in, **i.e.**, a permutation $\sigma_p : [1, n] \rightarrow [1, n]$ represented by a permutation matrix P. Such a matrix is orthogonal with $P^{-1} = P^t$, thus solving the equation $(PAP^t)Px = Pb$ is equivalent to solve the original equation $Ax = b$.

The graph associated with the factorized matrix L is $G(L) = (V, E^*, \sigma_p)$ with $E^* = E \cup \Re$, where edges include A edges as well as fill-in edges $\Re$. As presented in Algorithm 1, when considering the kth step of the factorization process, there will be fill-in in a_{ij} if both u_{kj} and l_{ik} are non-zeroes, with $k < min(i, j)$. Thus, if vertices i and j are adjacent to vertex k without being adjacent together, a fill-in edge (i, j) will be created in E^*. Theorem 1.1 gives a global characterization of fill-in edges.



(a) With original ordering.



(b) With a dedicated ordering.

Figure 1.1: Two different orderings for a five unknowns system. The original sparse matrix is A, associated with its graph G. The factorized matrix is L and in G^* fill-in edges were added. The elimination tree T gives dependencies between unknowns elimination.

Theorem 1.1. *(From [100]) Given an $n \times n$ sparse matrix A, and its adjacency graph $G = (V, E)$, any entry $a_{i,j} = 0$ from A will become a non-zero entry in the factorized matrix if and only if there is a path in G from vertex i to vertex j that only goes through vertices with a lower index than i and j.*

We then introduce the elimination tree to express in which order unknowns must be eliminated. Such a tree is presented in Figure 1.1(a) and Figure 1.1(b), where dependencies are expressed in the tree from the leaves to the root. One can notice that second ordering not only reduces the fill-in, but also increases the level of parallelism. The columns 1 to 4 from Figure 1.1(b) can be eliminated independently while column 5 will require the contributions from its sons before any computation.

Thus, the σ_p-permutation must meet two criteria to achieve good performances for the factorization: maintain the sparse structure ($E^* \approx E$) and provide a large balanced elimination tree for the sake of parallelism.

Among the ordering techniques known to efficiently reduce the matrix fill-in are the Approximate Minimum Degree (AMD) algorithm [12] and the Minimum Local Fill (MF) algorithm [85, 108]. However, these orderings fail to expose a lot of parallelism in the computation during the factorization. In order to both reduce fill-in and exhibit parallelism, an ordering algorithm named nested dissection has been introduced by George [43] and is now the most widely used in sparse direct solvers. The algorithm builds a partition of V vertices in the form $V = A \cup B \cup C$, where C, so called separator, is disconnecting A and B, meaning that there is no edge connecting any vertex from A to any vertex from B. Each path from a A vertex

to a B vertex goes at least by one vertex from C. In order to take advantage of the characterization theorem, one has to order C vertices with larger numbers than those of A and B vertices. It ensures that there will be no fill-in between A and B vertices.

The nested dissection algorithm is inherently recursive: after building a partition of V and ordering C vertices in a decreasing fashion, starting from n, the same process is applied recursively on A and B. When a subgraph is small enough, another strategy is used such as AMD or MF. As long as the fill-in will occur on C, namely the separator, building a small separator will minimize the fill-in. Given this process, the elimination tree describes the dependencies between separators and subparts elimination. In order to provide a balanced elimination tree, subparts A and B have to be of similar sizes.

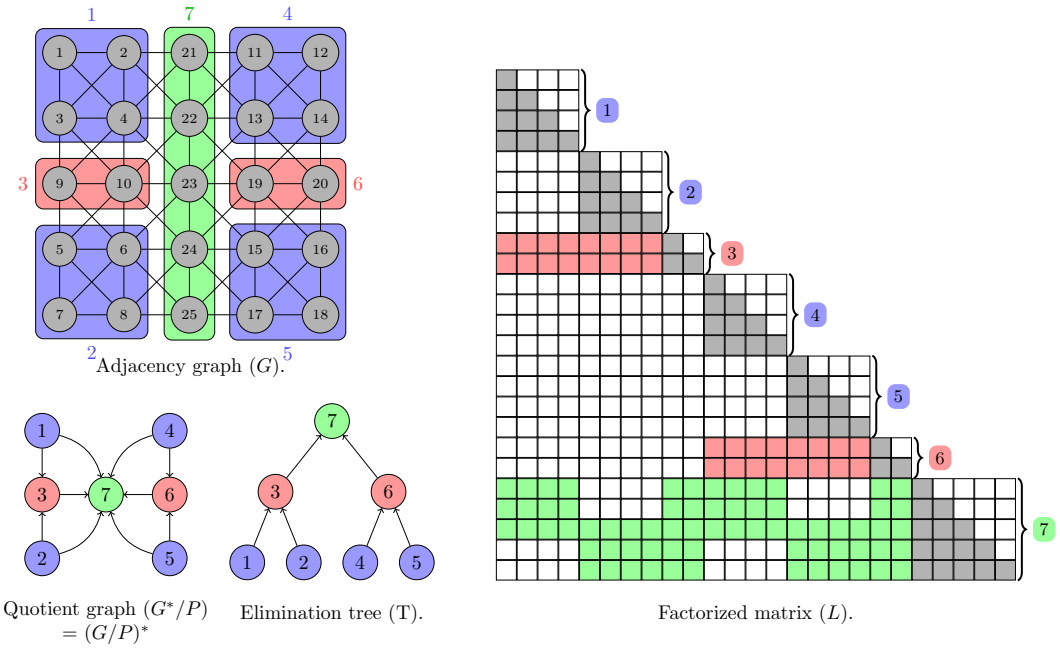


Figure 1.2: Nested dissection and block-data structure for L.

Figure 1.2 presents the nested dissection ordering on the adjacency graph of a 5-by-5 grid. The first separator (in green) divides vertices in two balanced sets, and the second-level separators (in red) partition correctly each subpart. Thus, we obtain a well balanced elimination tree with separators which are as small as possible.

This partitioning and ordering operations are usually performed through an external tool such as Metis [70] or Scotch [92].

1.1.2 Block-symbolic factorization

Now, let us see how is built the block-symbolic factorization. As we will present in Section 1.3, managing large data blocks is necessary to reach good efficiency on modern architectures. Thus, instead of partitioning the original matrix until reaching

subgraphs of size 1, a minimum size is used to obtain a resulting partition were columns are merged together into column blocks. There is a trade-off between 1) optimal memory consumption and number of operations, reached using subgraphs of size 1 and 2) practical time-to-solution, which can be reduced using block operations instead of scalars. In Figure 1.2, each separator represents a column block, as well as each subgraph obtained after stopping nested dissection (size ≤ 4 in this example). In this example, we obtain 7 column blocks, also called supernodes afterward.

In a more general case, a global supernode partition of the unknowns is obtained by merging the set of supernodes from the nested dissection process (all the separators) and the set of supernodes achieved from the reordered non-separated subgraphs (by using the algorithm introduced in [82, 83]).

Given this supernodal partition, one can compute the block-symbolic data structure of the factorized matrix, as presented on the right part of Figure 1.2. The objective is to predict the block-data structure of the final L (and U due to pattern symmetry) matrix for the numerical factorization and to gather information in blocks that will enable the use of efficient kernels as BLAS Level 3 operations [35]. This block-data structure is composed of N column blocks, one for each supernode of the partition, with a dense diagonal block (in gray in the figure) and with several dense off-diagonal blocks (in green and red in the figure) corresponding to interactions between supernodes; some additional fill-in may be accepted to form dense blocks in order to be more CPU-efficient. The block-symbolic factorization computes this block-data structure with $\Theta(N)$ space and time complexities [29]. From this structure, one can deduce the quotient graph which describes all the interactions between supernodes during the factorization, as well as the block elimination tree which describes the dependencies between the elimination of unknowns. For example, supernode 1 will contribute to supernodes 3 and 7. Finally, before distributing the column blocks on the processors, the biggest column blocks corresponding to the top most supernodes in the tree are split in order to exploit the parallelism inside the dense computations [59].

1.1.3 Numerical factorization

Finally, the sequential algorithm (cf. Algorithm 1) is replaced by a blocked algorithm, presented by Algorithm 2, which takes advantage of BLAS and that can be easily parallelized. Figure 1.3 summarizes the tasks when eliminating a sparse column block: 1) Factorize, 2) Solve, and 3) Update. Those blocked operations allow to increase the ratio computation/data, which helps to raise an acceptable efficiency with respect to the theoretical peak of a machine.

Several approaches exist to manage both parallelism and data locality in sparse linear solvers: multifrontal and supernodal. The multifrontal method [38] uses frontal matrices to accumulate updates such that during the elimination of supernodes, children are only updating their parent in the elimination tree. A front is composed of fully-summed variables, representing unknowns which have received all their contributions, and non-fully-summed variables, representing unknowns that are partially updated. Those unknowns will be transferred in upper levels of the elimination tree for being updated. It can be seen as the matrix composed of the colored blocks

Algorithm 2 Sequential blocked LU algorithm

```
For k ∈ [1, nb_blocks] Do
  Factorize block A_kk into L_{k,k}U_{k,k}
  For i ∈ [k+1, nb_blocks] Do
    Solve L_{i,k}U_{k,k} = A_{i,k}
    Solve L_{k,k}U_{k,i} = A_{k,i}
  End For
  For i ∈ [k+1, nb_blocks] Do
    For j ∈ [k+1, nb_blocks] Do
      Update A_{i,j} = A_{i,j} − L_{i,k}U_{k,j}
    End For
  End For
End For
```

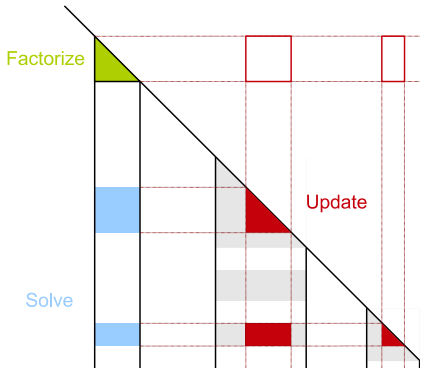


Figure 1.3: Tasks appearing during the elimination of a column block.

in Figure 1.3, where blue and green blocks correspond to fully-summed variables and red blocks to non-fully-summed variables. Another method, the supernodal approach, performs all updates between a node and its ancestors in the elimination tree directly. Thus, there is no need for extra memory to store non-fully-summed variables. In this approach, two solutions may be used to perform updates. The right-looking approach directly performs all updates immediately after eliminating a supernode (reading locality) while the left-looking approach computes all contributions to a same supernode the later, just before this supernode is eliminated (writing locality).

The factorization of the matrix is split among three elemental kernels. The **Factorize** kernel performs the factorization of a dense diagonal blocks and both **Solve** and **Update** stages are described after.

Solve. Usually, the solve task is done through one or multiple calls to BLAS kernels according to the data distribution used by the solver. Several task distributions can be used in sparse direct solvers. Using a 1D distribution, all blocks are stored contiguously in a dense storage to increase efficiency with a single BLAS call. With a 2D distribution, blocks are stored one-by-one; if it can reduce efficiency, it increases the level of parallelism, since there are many more independent updates.

Update. The update task can be done in multiple ways. The first option is to do it similarly to dense factorization with one matrix-matrix multiplication per couple of off-diagonal blocks (red updates in Figure 1.3). However, the granularity of tasks in sparse solvers is often small before reaching the top levels of the elimination tree. It makes this approach inefficient, as the number of elementary tasks generated is too large and their granularity too small. A commonly adopted solution for supernodal methods is to compute a matrix-matrix multiplication for each column block that requires updates, meaning one per blue off-diagonal block (only the first two are represented in Figure 1.3). The temporary result is then scattered and added to the target column block. The last option, similar to what is done in a multifrontal solver, consists in a single matrix-matrix multiplication that is followed by a 2D scatter of the updates. In the last two options, if the updates are too discontinuous and spread all over the updated submatrix, this can lead to memory bound additions while the operation is originally compute bound.

1.1.4 Triangular system solve

Once the factorization is performed, the solution can be obtained by solving successively the following systems:

1. $Ly = b$
2. $Ux = y$.

The solve is decomposed into a first step, called **forward substitution** and a second one, named **backward substitution**. It applies Gauss elimination on triangular matrices. Note that in the case of a single right-hand-side (when b is

a vector), those operations are cheap with respect to the numerical factorization. However, in the case of multiple right-hand-sides, it can become critical for the application.

1.1.5 Complexity

In real-life applications, a large number of graphs are issued from the mesh representation of physical 2D or 3D problems. The majority of those graphs are bounded-density [90] graphs. In this section, we summarize the theoretical complexities of direct sparse solvers using a nested dissection partitioning strategy.

Good separators can be built for classes of graphs occurring in finite element problems based on meshes which are special cases of bounded-density graphs [90] or more generally of overlap graphs [89]. In d-dimension, such n-node graphs have separators whose size grows as $\Theta(n^{(d-1)/d})$.

Lipton et al. [80] have studied the complexity of direct sparse linear solvers for 2D and 3D problems. With $\sigma = n^{(d-1)/d}$, the results for $\frac{1}{2} \leq \sigma < 1$ are summarized in Table 1.1. Computing the block symbolic factorization is in $\Theta(n)$, as shown by Charrier and Roman [29].

Type	σ	Number of elements in factors	Number of operations
2D	$\frac{1}{2}$	$\Theta(n \log(n))$	$\Theta(n^{\frac{3}{2}})$
3D	$\frac{2}{3}$	$\Theta(n^{\frac{4}{3}})$	$\Theta(n^2)$

Table 1.1: Complexities of sparse direct solvers for matrices issued from 2D and 3D meshes.

1.2 Low-rank compression

In practice, many applications do not require to compute a solution at the machine precision. Indeed, initial conditions (b) and discretization (formation of A) are not known exactly, and can have an accuracy far from the machine precision.

It is one of the reasons why incomplete factorizations were developed. With ILU(k) [60] factorization, fill-in paths longer than k are not considered, reducing both memory and time requirements. Another approach, named ILU(τ), drops values lower than a threshold τ during the factorization. Those approaches show significant results on many applications, but they are not fully algebraic as they do not consider all the numerical properties of a given problem. Thus, the use of low-rank approximation in direct solvers has been developed to perform approximate factorizations with a better dropping criterion.

A matrix A of size m-by-n is rank-deficient if it exists a couple of matrices U and V of size m-by-r and n-by-r with $r < min(m, n)$ such that $A = UV^t$. In this thesis, we are interested in computing low-rank approximations with a relative error

lower than ε, such that $||A - UV^t|| \leq \varepsilon ||A||$. The best representation respecting this criterion can be obtained through the Singular Value Decomposition(SVD) [39].

In this section, we will describe the different criteria to represent a matrix in a low-rank form, before detailing several low-rank formats. Then, we will present methods to compute a UV^t form and will give details of the updating stage.

1.2.1 Low-rank formats

1.2.1.1 Block-admissibility criterion

Let us consider a matrix decomposed into a set of blocks, by a block-clustering strategy that split the set of unknowns into clusters as it will be presented further. A block C, noted $\sigma \times \tau$, of this matrix corresponds to the interaction between two sets σ and τ, where σ contains the row indices of the block while τ contains its column indices.

If $\sigma = \tau$, the block C is a diagonal block, representing self-interaction and is full-rank. However, is $\sigma \neq \tau$, C is an off-diagonal block that may be considered low-rank. Determining if the block is suitable (admissible) or not for compression is referred to as the admissibility condition.

A widely used admissibility condition, named strong block-admissibility is defined as follows:

$$\sigma \times \tau \text{ is admissible} \iff \max(\text{diam}(\sigma), \text{diam}(\tau)) \leq \eta \ \text{dist}(\sigma, \tau) \tag{1.1}$$

where η is a fixed parameter, diam() is the geometric diameter of a cluster and dist() the minimum distance between two clusters. It means that, intuitively, the rank of a block is correlated to the distance between σ and τ, and relatively to their respective diameters. The greater the distance, the smaller the rank. This criterion can be relaxed using a large η such that a block is admissible if $\text{dist}(\sigma, \tau) > 0$, **i.e.**, if clusters are not direct neighbors.

Another used admissibility condition named weak admissibility is less strict:

$$\sigma \times \tau \text{ is admissible} \iff \sigma \neq \tau. \tag{1.2}$$

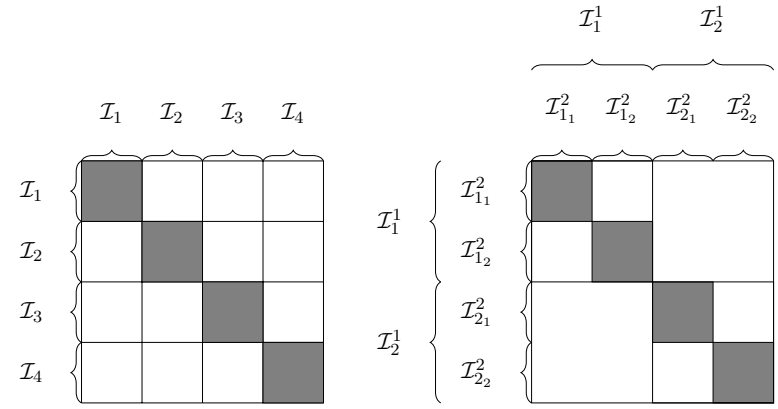
With this last admissibility condition, only diagonal blocks (representing self-interaction) are not admissible.

1.2.1.2 Block clustering

For the sake of simplicity, we consider in this section that rows and columns are clustered in the same fashion, although notations can be extended otherwise.

With a flat clustering as presented in Figure 1.4(a), the set of unknowns is split among k clusters: $\mathcal{I} = (\mathcal{I}_1, \ldots, \mathcal{I}_k)$, and the interaction between two clusters $\mathcal{I}_i$ and $\mathcal{I}_j$ can be considered low-rank following an admissibility criterion as given previously.

With a hierarchical clustering as presented in Figure 1.4(b), the set of unknowns is split recursively. At a given level ℓ of recursion, the i-th block is clustered as $\mathcal{I}_i^\ell = (\mathcal{I}_{i_1}^{\ell+1}, \ldots, \mathcal{I}_{i_k}^{\ell+1})$. The interaction between blocks on the same level and sharing



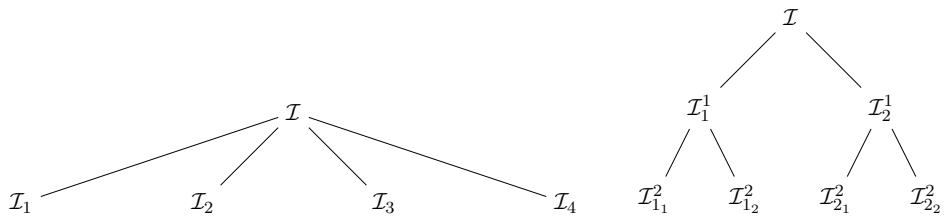
(a) Flat clustering with four clusters.

(b) Hierarchical clustering.

Figure 1.4: Flat and hierarchical clustering.

the same parent is low-rank (accordingly again to some block-admissibility condition) and leaves represent self-interaction and are thus full-rank.

From the block clustering, the cluster tree can be built, as presented in Figure 1.5(a) for the flat clustering and in Figure 1.5(b) for the hierarchical clustering.



(a) Flat cluster tree.

(b) Hierarchical cluster tree.

Figure 1.5: Flat and hierarchical cluster trees.

If building such a cluster tree and identifying admissible blocks can be straightforward in a geometry context, it is a more challenging problem in a fully algebraic context, without knowledge of the underlying geometry of this problem. For this reason, distances and diameters in the graph representing the matrix can be used instead of geometric information. In addition, if a binary tree is often used for the sake of simplicity (as it appears in Figure 1.5(b)), the number of clusters of $\mathcal{I}_i^\ell$ can be arbitrary large in the general case.

1.2.1.3 Nested bases

To present nested bases, we will now rely on low-rank structures of the form UBV^t, where B is a r-by-r matrix, and U and V represent orthogonal bases. Using nested bases, the low-rank bases U and V of a block depend on the basis of its descendants in the cluster tree. For a simple example, let us consider a two-level hierarchy as

presented in Figure 1.5(b). We consider the low-rank representation of the interaction block between clusters $\mathcal{I}_1^1$ and $\mathcal{I}_2^1$. The low-rank base $U_{1\times 2}$ will depends on $U_{1_1\times 1_2}$ (and $U_{1_2\times 1_1}$), which corresponds to the left base for the block representing interaction between clusters $\mathcal{I}_{1_1}^2$ and $\mathcal{I}_{1_2}^2$, as given by the following equation:

$$U_{1\times 2} = \begin{pmatrix} U_{1_1\times 1_2} & 0 \\ 0 & U_{1_2\times 1_1} \end{pmatrix} U_{1\times 2}^{small}. \tag{1.3}$$

Thus, the storage is reduced to the storage of $U_{1\times 2}^{small}$, B and $V_{1\times 2}^{small}$. If more levels are used in the cluster tree, both $U_{1_1\times 1_2}$ and $U_{1_2\times 1_1}$ will also be represented in the same form.

The matrix $U_{1\times 2}^{small}$ is named transfer matrix, since it allows to extend the children bases to their father.

1.2.1.4 Classification

From the block-admissibility, block clustering and low-rank bases criteria, one can classify existing low-rank formats as presented in Table 1.2.

Table 1.2: Classification of existing compression methods.

Block-admissibility	Partitioning		
	Flat	Hierarchical	
		Without nested bases	With nested bases
Weak	BLR [8]	HODLR [13] (binary tree)	HSS [112]
Strong		$\mathcal{H}$ [54]	$\mathcal{H}^2$ [56]

The main difference between hierarchical formats is that for HSS and HODLR matrices, off-diagonal blocks are not refined (weak-admissibility), while for $\mathcal{H}$ and $\mathcal{H}^2$ matrices, the blocks are recursively clustered until reaching suitable admissibility condition. In addition, HODLR matrices require the use of a binary tree, which is not the case in other hierarchical formats. However, practical implementations are often restricted to binary trees.

The use of nested matrices for HSS and $\mathcal{H}^2$ matrices make those formats more restrictive, since they require the existence of transfer matrices to exploit the low-rank basis of the children.

BLR can be seen in both weak and strong block-admissibility categories. Indeed, all blocks can be admissible (weak) while in practice some blocks will be kept in full-rank if their rank is too high (strong). In addition, off-diagonal blocks that are at distance 1 from the diagonal can sometimes be considered as non-admissible as they represent close interactions. In each case, the complexity is kept untouched [10], under some constraints on the number of blocks that are not compressed.

1.2.2 Low-rank compression kernels

The goal of low-rank compression is to represent a general dense matrix A of size m-by-n by its compressed version $\hat{A} = UV^t$, where U, and V, are respectively matrices of size m-by-r, and n-by-r, with r being the rank of $\hat{A}$, supposed to be small with respect to m and n. In order to keep a given numerical accuracy we choose r such that $||A - \hat{A}||_F \leq \tau ||A||_F$, where τ is the prescribed tolerance and $||.||_F$ the frobenius norm.

Note that we refer to as $*$ in indices when all columns (or all rows) are used.

1.2.2.1 Singular Value Decomposition (SVD)

A is decomposed as $Q\Sigma\tilde{Q}^t$, where Q and $\tilde{Q}$ are orthogonal matrices and Σ contains singular values on its diagonal. The low-rank form of A consists of the first r singular values and their associated singular vectors such that: $\sigma_{r+1} \leq \tau ||A||_F$, $U = Q_{*,1:r}$, and $V^t = \Sigma_{1:r}\tilde{Q_{1:r,*}}^t$ with $Q_{*,1:r}$ (respectively $\tilde{Q_{1:r,*}}$) being the first r columns of Q (respectively $\tilde{Q}$) and $\Sigma_{1:r}$ is the vector made of the first r singular values. This solution presents multiple drawbacks. It has a large operation cost varying as $\Theta(m^2n + n^2m + n^3)$ to compress the matrix, and all singular values need to be computed to get the first r ones. On the other hand, as the singular values of the matrix are explicitly computed, it leads to the lowest rank for a given tolerance [39].

1.2.2.2 Rank-Revealing QR (RRQR)

A is decomposed as QRP, where P is a permutation matrix, and QR the QR decomposition of AP^{-1}. The low-rank form of A of rank r is then formed by $U = Q_{*,1:r}$, the first r columns of Q, and by $V^t = R_{1:r,*}$, the first r rows of R. The main advantage of this process is that one can stop the factorization as soon as the norm of the trailing submatrix $||\tilde{A}_{r+1:m,r+1:n}||_F = ||A - Q_{*,1:r}R_{1:r,*}P||_F$ is lower than $\tau ||A||_F$. Thus, the complexity is lowered by $\Theta(mnr)$ operations. However, it returns an evaluation of the rank which might be a little larger than the one returned by the SVD approach and thus induces more flops during the numerical factorization and solve steps.

1.2.2.3 Randomization

Most randomization techniques used to compress a dense matrix are dedicated to make a better use of BLAS Level 3 kernels to reach a higher efficiency. The approach relies on computing a randomized sampling by projecting the matrix to compress onto a small Gaussian random matrix which is expected to be full rank. The main issue with RRQR being the search for column pivoting through the full matrix, the objective of randomized RRQR algorithms is to perform this operation on a smaller basis to be more efficient. Then pivots are applied by block to the original matrix and efficient block operations are performed.

Several algorithms were developed to take advantage of randomization in RRQR. However, all those approaches compute an approximation for a given rank. It is thus impossible to predict the accuracy of the result unless good rank prediction can be obtained through knowledge of the underlying problem. In [113], RRQR is performed

on the smaller subspace obtained after projecting the matrix into the random space. At each step, it requires to update the full matrix, as for a classic QR factorization, which is quite costly. In [36], the authors extend this solution with a left-looking strategy, where updates are only applied when a new column is treated. However, it requires extra workspace to save previous computations. Their approach allows to refine the solution if a good approximation of the rank is known a priori.

In [87], Martinsson proposed two spectrum-revealing factorization methods, in order to obtain values close to singular values on the diagonal of the matrix R. The control of the error depends on the convergence of those values to singular values, and thus the quality of the compression may be degraded. The first one, similar to [113], forms a permutation matrix while the second one relies on rotations to find pivots.

The use of randomization techniques will not be studied in this thesis, since algorithms presented in [36, 87, 113] work with a knowledge of the rank. In addition, it was shown that those methods are efficient for large matrices. In the BLR approach presented in this thesis, the block sizes used may be too small to take advantage of those methods.

1.2.2.4 Summary

In the following chapters of this thesis, we will only consider RRQR and SVD kernels. SVD is much more expensive than RRQR. However, for a given tolerance, SVD returns lower ranks. Put another way, for a given rank, SVD will have better numerical accuracy. Thus, there is a trade-off between time-to-solution (RRQR) versus memory consumption (SVD), and SVD will be used as a reference point for memory consumption.

Some other techniques are based on the graph of the matrix, such as Pseudo-skeletal approximation [57] or Boundary Distance Low-Rank (BDLR) [13]. Adaptive Cross Approximation (ACA) [99] builds the approximation by keeping columns with the largest norms. The main issue with those methods is the control of the error, especially when there is no knowledge of numerical properties of the problem. In addition, those methods often work with the original sparse matrix while there is fill-in during the factorization. If such approaches can be suitable to form a low-rank representation of a sparse matrix, the rank of the representation may grow due to updates during the factorization, and no numerical control can help to increase the rank.

1.2.3 Full-rank or low-rank assembly

Beyond the different formats presented previously, a main characteristic of direct solvers using low-rank compression is the type of assembly, **i.e.**, the sum operation in the update process. In the fully-structured approach, a block is considered low-rank all throughout computations and low-rank assembly is performed. In the non-fully-structured approach, a block is compressed after it received all its contributions, which were applied through full-rank assembly.

From this classification, we refer to Section 2.1 for some existing solvers using

low-rank compression. We present their differences among the representation used, the algebraic or geometric approach, and the type of assembly.

1.3 The PaStiX sparse supernodal solver

In this thesis, every development has been done in the context of the sparse direct solver PaStiX [59]. The ongoing hardware evolution leads to systems with many computational units, exhibiting a huge level of heterogeneity, since a single system may be composed of many cores and some accelerators. To provide more computational capabilities, those systems are coupled together into distributed clusters of SMP machines.

We focus on the PaStiX solver in a heterogeneous context. The development of this solver has started two decades ago, following the advances made on modern architectures, such as the advent of accelerators. PaStiX is able to manage both real and complex precision and can perform any kind of factorization: LU, LL^t, LDL^t, LL^H and LDL^H. Iterative refinement (BICGSTAB, CG, GMRES) can be used to refine the quality of the solution. In this work, we use the static pivoting strategy which was introduced in [79].

In Section 1.3.1, we describe how the granularity of kernels impacts efficiency and number of low-rank updates, before describing how the PaStiX solver was designed to take advantage of modern architectures in Section 1.3.2.

1.3.1 Modern architectures

On modern heterogeneous architectures mixing CPUs and accelerators such as GPUs or Intel Xeon Phi, the number of computational units requires a large number of independent tasks to reach a correct level of efficiency. In addition, to fully take advantage of many architectural features such as caches, vector units or shared memory on accelerators, the granularity of computations has to be correctly managed. It makes the parallelization process a challenging problem: blocking sizes have to remain large enough to maintain efficiency of basic kernels while many tasks have to be expressed to feed a large number of workers.

For instance, a widely used kernel in linear algebra is the GEneral Matrix-Matrix product, referred to as `GEMM`. This operation is computational bound, since the number of operations is much larger than the data accesses. Thus, one can expect to reach a good level of performance with this particular operation. However, as presented in Figure 1.6, the matrix size is an important factor to reach good efficiency on modern architectures. If a size of 300 can be sufficient for a CPU, an even larger size is required to fully take advantage of modern GPUs, and this trend continues with last architectures.

1.3.2 Exploiting modern architectures in the PaStiX solver

The use of BLAS Level 3 kernels together with parallelization techniques has demonstrated significant speedup over multi-threaded and distributed architectures. However, the kernels granularity had to be adapted to target architectures with a larger

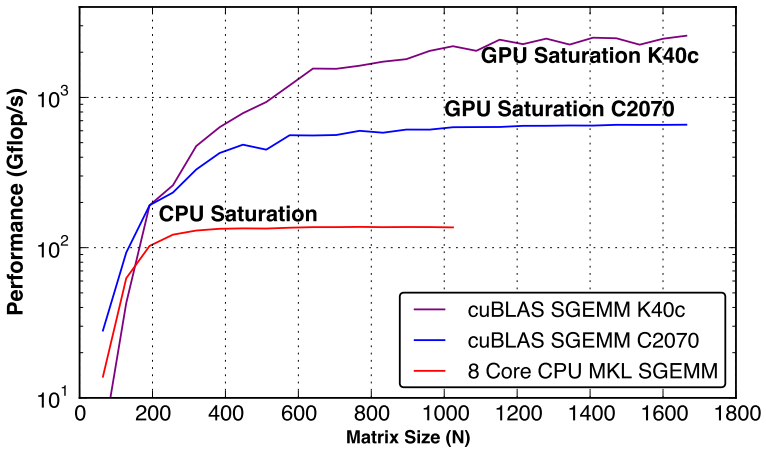


Figure 1.6: GEMM performance on different architectures.

number of computational units. This problem was first studied in the context of dense matrices. From LAPACK [15], using multi-threaded BLAS Level 3 kernels, Plasma [73] was developed, expressing the algorithm in the form of a Directed Acyclic Graph (DAG) of tasks, where each node represents a computational kernel and each edge a data dependency between two tasks. For distributed architectures, such an approach was used to move from the 2D distribution of ScaLAPACK [23] to more advanced approaches. In Dplasma [24] and Chameleon [3], such an approach is used for distributed architectures with task-based implementations. After expressing the algorithm as a DAG, a runtime system such as Parsec [25] or StarPU [19] schedules tasks on the different computational units available respecting dependencies. The use of DAGs allow to separate writing task-based algorithms from developing smart scheduling strategies. In addition, it can help to better manage certain constraints such as memory consumption [105].

PaStiX uses a right-looking approach, which performs all updates when a supernode is eliminated. Thus, it can induce writing locality problems, when an update modifies non contiguous parts of a target supernode. It is then interesting to control the number of off-diagonal blocks to reduce this phenomenon by using larger blocks.

The solver manages several elementary task distributions, as presented in Section 1.1.3. Using a 1D distribution favors data locality, while a 2D distribution increases the number of independent tasks and thus provide more parallelism.

To manage parallelism, several approaches were developed. Note that all elemental kernels are sequential. PaStiX is using only sequential BLAS Level 3 calls and manages all the parallelism internally, contrary to some solvers that perform multi-threaded BLAS Level 3 operations on large blocks. Static scheduling splits the work among workers before any numerical operations, which can degrade load balancing. Dynamic scheduling [40] was introduced to allow work stealing among different workers belonging to a same shared-memory node and then reduce the time-to-solution. Recently [76], the use of Parsec and StarPU runtime systems was introduced in PaStiX, especially to take advantage of accelerators (GPUs, Xeon Phi...) more eas-

ily. In addition, it allows handling more irregular kernels, such as the one that will be developed for low-rank compression, as load balancing is dynamically managed.

1.4 Positioning of the problem

Sparse direct solvers are widely used, despite their memory and computational complexities, because they are able to provide a robust solution to sparse systems. However, many solvers relying on iterative, hybrid or incomplete methods were developed to reduce time-to-solution or to be able to solve larger problems, that would not have fit in memory with the direct approach. The main issue is that many approaches are geometry-based or require a knowledge of the underlying equation. In this thesis, we consider the PaStiX sparse supernodal solver. This solver is working in a fully algebraic context, and proposes new features to reduce the burden on memory consumption and/or time-to-solution, that we will benefit in the context of this thesis.

More recently, the use of low-rank compression has gained a lot of interest, since it allows the application to better control the error made with approximation. It is particularly interesting when the accuracy required by the application is less important than the machine precision. Still, using those methods in a fully algebraic approach is an open problem, and many solvers are used as preconditioners but cannot provide directly a good accuracy such as 10^{-10} or 10^{-12} in double precision. Especially, reducing both time-to-solution and memory consumption asymptotic complexities at a given accuracy is still a challenging problem. In this thesis, we want to consider this problem in a supernodal context.

When using low-rank compression in sparse direct solvers, a widely used solution is to represent separators and off-diagonal blocks with a low-rank structure. In a fully algebraic context, building a suitable cluster tree and identifying which blocks are admissible is not straightforward. Thus, we want to study low-rank clustering techniques in the sparse case, and to see if the widely used nested dissection algorithm can be enhanced to increase the compressibility of blocks. In the same idea, we want to study blocking strategies for several reasons, such as increasing the blocking size for modern architecture performances or to manage larger low-rank structures. In particular, we want to study reordering techniques to reduce the number of off-diagonal blocks and better manage block computation granularity.

In Chapter 2, we will present existing solutions that address parts of those problems. We describe direct solvers using low-rank compression, techniques that are used to enhance data locality in off-diagonal blocks and clustering approaches in the sparse case. From existing solution, we will conclude with a presentation of the contributions of this thesis.

Chapter 2

State of the art

As we have seen previously, low-rank compression techniques allow reducing time-to-solution and/or memory consumption and thus have gained lot or interest in direct solvers. Yet, there are still open problems, especially when applied to sparse matrices, for which many techniques were developed to correctly exploit sparsity. A fundamental aspect of low-rank compression in sparse solvers is to combine sparse constraints with the techniques that were designed to apply compression for dense matrices. In this Chapter, we present direct solvers (for dense and sparse matrices) using low-rank compression in Section 2.1. Reaching high efficiency on top of modern architectures requires handling large blocks of data, to make a good use of BLAS Level 3 kernels. In Section 2.2, we discuss strategies that can enhance the sparsity pattern by reordering the unknowns within separators. Similarly, unknowns can be reordered to exhibit better cluster trees when using low-rank compression techniques. We present existing techniques that are used to cluster unknowns in Section 2.3. We then conclude with a positioning of the contributions that will be presented in this thesis in Section 2.4.

2.1 Solvers using low-rank compression

In the context of direct solvers, some recent works have investigated the low-rank representations of matrices or dense blocks appearing in sparse matrix factorization. These blocks may be compressed using many possible formats such as BLR, $\mathcal{H}$, $\mathcal{H}^2$, HSS, HODLR,... Depending on the compression strategy, these solvers require knowledge of the underlying geometry to tackle the problem or can do it in a purely algebraic fashion.

Hackbusch [54] introduced the $\mathcal{H}$-LU factorization for dense matrices. It compresses the matrix into a hierarchical matrix before applying low-rank operations instead of classic dense operations. In this approach, the full matrix is expressed as an $\mathcal{H}$ matrix and low-rank assembly is performed. This is a fully-structured approach. This work was extended for sparse matrices by Grasedyck et al. in [53]. It takes advantage of the nested dissection to exhibit large structural zeroes as in sparse direct solvers. Indeed, during the partitioning process, the graph is partitioned into $A \cup B \cup C$ where C is a separator, such that there is no interaction between subparts

A and B. However, as opposed to sparse direct solvers, the coupling between A and C (respectively B and C) is considered dense as no symbolic factorization is performed. It means that some blocks that are structurally made of zeroes are considered as rank-0 low-rank matrices instead of directly avoiding the management of those blocks, as it is made in most sparse direct solvers. Those works on $\mathcal{H}$-LU factorizations for dense and sparse matrices are summarized in [55]. In that paper, the existence of zeroes blocks in the coupling is showed, but the size of any off-diagonal block always matches the size of the leaf of the target separator. As blocks are larger than necessary, it can reduce the potential gains on memory consumption since some structural zeroes are stored. In [52], Grasedyck et al. used $\mathcal{H}$-LU factorization in an algebraic context, where the geometry of the problem is unknown. Performance, as well as a comparison of $\mathcal{H}$-LU with PARDISO and UMFPACK sparse direct solvers is presented in [51], and it shows that the approach is competitive for some problems, but can degrade time-to-solution and memory consumption with respect to sparse direct solvers. Kriemann [72], Lizé [84] and Aliaga et al. [7] implemented $\mathcal{H}$-LU factorization for dense matrices using a task-based approach and runtime systems. It allows to better handling load balancing, especially when ranks are unknown before numerical operations, which may degrade static balancing strategies.

The HODLR compression technique was used by Aminfar et al. in [13] to accelerate the elimination of large dense fronts appearing in the multifrontal method, relying on the Woodbury matrix identity. The authors demonstrated the assets of such an approach for dense fronts. However, if it enhances the factorization step, the solve step is costly, which may degrade performance. In addition, there is no asymptotic memory consumption gain with this approach since each front is allocated in a dense fashion before low-rank compression is performed. It was fully integrated in a sparse solver by Aminfar and Darve in [14], using Boundary Distance Low-Rank (BDLR) to allow both time and memory savings by avoiding the formation of dense fronts. The solver allows reducing time-to-solution and memory consumption but is used as a low-accuracy preconditioner.

A supernodal solver using a compression technique similar to HODLR was presented by Chadwick and Bindel in [28], based on the CHOLMOD [30] solver. The proposed approach allows memory savings and can be faster than standard preconditioning techniques. However, it is slower than the full-rank version in the benchmarks and requires an estimation of the rank to use randomized techniques that accelerate the solver.

The use of HSS matrices in sparse direct solvers has been investigated in several solvers. In [112], Xia et al. presented a solver for 2D geometric problems, where all operations are computed algebraically. In [109], Wang et al. developed a geometric solver, but contribution blocks are not compressed, making memory savings limited. Ghysels et al. [47, 48] proposed an algebraic code that uses randomized sampling to manage low-rank blocks and to allow memory savings in the STRUMPACK solver. Randomization techniques have been studied for HSS matrices in [86, 111], and take advantage of nested bases to accelerate the solver. The main issue of those approaches is that they require the existence of these nested bases. In addition, randomization techniques may fail to provide high accuracy solution and those solvers may be used mostly as preconditioners.

Hackbusch and Börm introduced $\mathcal{H}^2$ arithmetic [56], which has been used in several sparse solvers. In [94], Pouransari et al. introduced a fast sparse $\mathcal{H}^2$ solver, called LoRaSp and based on extended sparsification. In [115], Yang et al. presented a variant of LoRaSp, aiming at improving the quality of the solver when used as a preconditioner, as well as a numerical analysis of the convergence with $\mathcal{H}^2$ preconditioning. In particular, this variant was shown to lead to a bounded number of iterations irrespective of problem size and condition number (under certain assumptions). In [61], Ho and Ying introduced a fast sparse solver based on interpolative decomposition and skeletonization. It was optimized for meshes that are perturbations of a structured grid. In [107], Sushnikova and Oseledets described an $\mathcal{H}^2$ sparse algorithm. It is similar in many respects to [94], and extends the work of [61]. All these solvers have a guaranteed linear complexity, for a given error tolerance, and assuming a bounded rank for all well-separated pairs of clusters (the admissibility criterion in Hackbusch et al.'s terminology and presented in Section 1.2.1.1).

Block Low-Rank compression has been investigated for dense matrices in an industrial context by Anton et al. [16] and Lacoste et al. [75]. It has also been studied by Abdulah et al. [1, 6]. It was studied in a sparse context in [8, 88] by Amestoy et al. when using the multifrontal solver MUMPS [9]. Considering that this approach is similar to the current study, a detailed comparison will be given in Section 3.5, with the contributions of this thesis regarding low-rank compression in the PASTIX solver. One of the differences of our approach with [88] is the supernodal context coupled with fully-structured approach (low-rank updates) that leads to different low-rank operations, and possibly increases the memory savings.

Recent works have investigated new low-rank formats between $\mathcal{H}$ and BLR to take advantage of both worlds. Indeed, if $\mathcal{H}$ matrices allow a better theoretical complexity, the regularity of blocks distribution in BLR makes it suitable to reach good performance and better load balance computations. In [65, 66], Ida et al. proposed to express the matrix as a BLR matrix at a coarse grain, to distribute easily blocks in a distributed context, following SCALAPACK 2D distribution. Then, each block is, at a finer grain, seen as an $\mathcal{H}$ matrix to reduce both the memory and computational complexities. In [11], the MUMPS authors proposed a multilevel BLR format. A matrix is expressed in a BLR format and diagonal blocks can be themselves refined using BLR compression, with a constant number of levels. Under certain conditions, both those approaches can reach the complexity of $\mathcal{H}$ matrices, although it is more restrictive.

Most of the solvers presented before are managing matrices with a knowledge or the geometry and/or the underlying equations, or require the existence of nested bases (HSS, $\mathcal{H}^2$) or weak admissibility criterion (HODLR, HSS). In addition, if some approaches can provide high accuracy solutions, many solvers are used as preconditioners for iterative methods. Finally, approaches using low-rank assembly are either working for some well-defined problems [112], or without fully considering sparsity in off-diagonal blocks [28, 55]. In several multifrontal sparse direct solvers [8, 13], low-rank compression is performed on dense fronts, and thus it avoids the problem of low-rank assembly between sparse structures, at the cost of storing fronts in a dense fashion.

In this thesis, we target to build an algebraic solver that does not require nor

nested bases or weak admissibility criterion and that can directly provide a solution at a prescribed precision. In addition, we propose to use a fully-structured approach to maximize memory consumption savings and thus solve problems that would not have fit in memory with other approaches.

2.2 Reordering strategies to enhance blocking

Studying the structure of off-diagonal blocks was used in different contexts. In [46], George and McIntyre proposed to reduce the overhead associated with the management of each single off-diagonal block thanks to a reordering strategy that refines the ordering provided by the minimum degree algorithm. Their experiments were applied to 2D graphs and successfully reduced the number of off-diagonal blocks. However, the authors did not provide a theoretical study of their reordering algorithm and their solution did not apply in the context of 3D graphs.

In [63], Hogg et al. introduced reordering techniques in the context of both supernodal and multifrontal solvers for the HSL solver [104]. The objective is to create larger off-diagonal blocks to enhance data locality and reduce factorization time in the MA87 supernodal solver. Each diagonal block is reordered according to its set of children. Given a child and one of its ancestors, Section 1.1.2 showed that the set of rows from the ancestor connected to the child should be ordered contiguously to create a single off-diagonal block and avoid scattering operations. HSL reordering strategy starts by sorting children according to their contribution size to the studied supernode, in other words with the number of rows that connect the two nodes. Then the rows connected to the larger child are numbered continuously to create a single off-diagonal block coming from this child, and the process is repeated on the remaining rows until all of them are reordered. This strategy has led to performance gains in a multi-threaded context [63]. However, from construction, this algorithm gives priority to the largest branch of the elimination tree, neglecting the other ones. In practice, we observe that leaves from both sides of the elimination tree might be connected to the same unknowns of their ancestors, making this solution sub-optimal.

In [106], Sid-Lakhdar proposed a reordering strategy for the multifrontal solver MUMPS. The objective is to provide a row ordering and the associated mapping on a set of processors to minimize the total volume of communications. This strategy minimizes communication between a parent and its set of children. As opposed to HSL and the solution we propose in Chapter 4, this algorithm, designed for multifrontal solvers only, dynamically reorders the rows at each level of the elimination tree. It considers only interactions from children to direct parents to minimize scattering operations at each level when updating the frontal matrix. This way, rows from a parent node can have different orderings at each level of the elimination tree, or even between different child branches.

2.3 Low-rank clustering strategies

As presented in Section 1.2.1.2, finding a suitable clustering of unknowns is a crucial problem to obtain low-rank structures. A commonly used approach consists in

building clusters to respect a given admissibility condition. For dense matrices, one can require clusters to have a small diameter and a few neighbors to increase the number of interactions that will be considered as compressible. In this section, we present existing techniques for dense and sparse clustering.

In [98], Rebrova et al. built cluster trees from the geometry of specific problems. Such an approach cannot directly be extended to the algebraic case, which is the focus of this thesis.

In [21], Bebendorf presented block-admissibility conditions and a spectral bisection strategy to cluster unknowns of a sparse matrix is introduced. The authors also mention nested dissection to perform the low-rank clustering of a sparse matrix. As in our study we are interested in clustering separators, using such an approach would be similar to use graph partitioning techniques on the subgraph induced by the separator.

In [116], Yu et al. presented a method to cluster unknowns of a dense matrix without the knowledge of the geometry or properties of underlying equations. The authors use the fact that any SPD matrix corresponds to a Gram matrix of vectors in an unknown Gram space [62]. Each entry of the matrix can be seen as an inner product, which allows to define distances among points. The authors use sampling to avoid computing all distances as it would be too costly for a dense matrix, and then split unknowns recursively to obtain a balanced cluster tree using those distances.

In practice, k-way partitioning or recursive bisection are the most commonly used approaches to perform clustering of fronts or separators. It is the strategy adopted in both MUMPS and STRUMPACK solvers. However, those graph methods are dedicated to connected graph, which is not necessarily the case for a front or a separator. In practice, those subgraphs issued for the original larger graph are reconnected using halo vertices at distance 1 or 2 to obtain fully connected graphs. Then, k-way methods available in partitioning tools such as METIS or SCOTCH are used to perform the clustering.

The techniques used to reduce the number of off-diagonal blocks appearing in the block-symbolic structure (as presented in Section 2.2) can also be used to enhance the sparsity pattern and thus reduce the number of low-rank updates. Such approaches allow clustering vertices that will receive similar contributions together, but those vertices are not necessarily close in the subgraph of the separator receiving contributions, which can degrade compressibility within the separator. When such a reordering strategy is used, vertices are clustered after the reordering process. Since vertices receiving similar contributions are ordered consecutively, one can expect that splitting the set of vertices into clusters of equal sizes will provide a suitable ordering. In practice, a smarter split using non fixed sizes up to a given tolerance has been introduced in Lacoste thesis [74] to reduce the number of off-diagonal blocks split when clustering the unknowns of a given separator.

To summarize, clustering techniques can be classified into four classes: 1) use the geometry and/or the kernel function of the underlying equations; 2) introduce algebraically a distance for a dense matrix and use graph partitioning methods; 3) apply graph partitioning techniques for sparse matrices and 4) focus on sparse pattern without considering distances in the graph. Clustering techniques corresponding to 1) and 2) are out-of-scope of this thesis since they concern dense matrices or are not

algebraic.

2.4 Positioning of the thesis and description of contributions

The main objective of this thesis is to enhance the PaStiX sparse direct solver by introducing low-rank compression techniques to reduce time-to-solution and/or memory consumption. We also study ordering and clustering techniques to enhance the blocking sizes and the granularity of low-rank structures in the objective of reaching good efficiency as well as good low-rank compression rates. A fundamental challenge of this work is to introduce low-rank compression in a sparse supernodal solver while keeping a fully algebraic approach. It should lead to higher practical memory consumption savings than low-rank compression in multifrontal solvers, by avoiding the extra cost of storing fronts.

Before presenting each contribution of this thesis, let us illustrate a crucial problem when using low-rank compression inside separators issued from the nested dissection process. For the sake of simplicity, we consider a regular cube and two levels of nested dissection, as presented in Figure 2.1. Note that this example presents a perfect nested dissection since separators are as small as possible and split vertices among perfectly balanced subparts. The gray plane corresponds to the first separator and the green and red planes correspond to second-level separators. On the left, the graph made of vertices of the first separator is drawn. Vertices that are directly connected to vertices of second-level separators are colored in green and red.

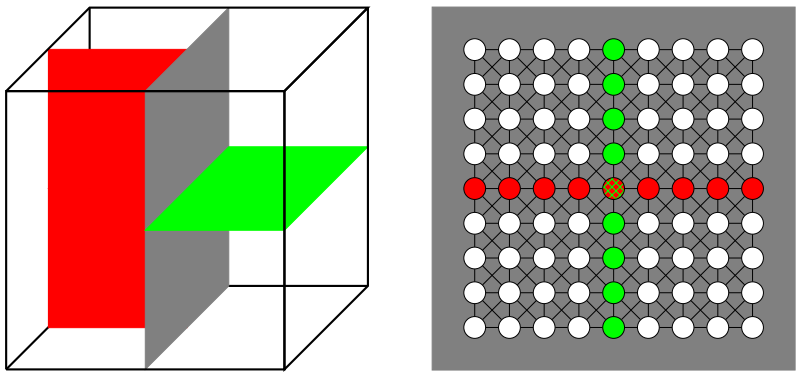


Figure 2.1: Two levels of nested dissection on a regular cube (on the left) and traces (green and red) of second-level separators on the first separator (on the right).

In Section 1.1.2, we presented the block-symbolic factorization which allows to build the block structure of L and U, with one dense diagonal block per supernode and several off-diagonal blocks. A classical approach used by most solvers presented in Section 2.1 consists of compressing diagonal blocks as well as off-diagonal blocks issued from this structure. In a supernodal context, where no fronts are used to accumulate contributions between children and their direct parent, the granularity of

the updates has to be correctly controlled to take advantage of modern architectures (cf. Section 1.3.1) and to limit the number of elementary updates (cf. Section 1.1.3).

A classical approach consists of using k-way partitioning in fronts or separators to obtain the clustering. The main issue is that it correctly considers interactions within a separator but not from outside the separator. In practice, if we consider the sparse matrix corresponding to the cube in Figure 2.1, the approach considers only vertices of the gray separators and orders those vertices without considering upcoming contributions. One can note that a k-way partitioning will divide vertices corresponding to interaction with direct children (red and green traces) among several clusters, while they receive exactly the same contributions.

The challenge of clustering unknowns within a separator is that, contrary to what is made for dense matrices, the clustering does not impact only the separator compressibility, but also the sparsity pattern of contributions that will be applied to the separator. Since existing strategies do not take into account both intra-separator compressibility and inter-separator sparsity pattern, there is room for improvement. For both the multifrontal and supernodal methods, using low-rank assembly requires to exhibit suitable data structures to perform efficiently updates between children and their parent(s). As pointed out in Figure 2.1, there is no optimal solution since separators are issued from generic partitioning tools that may destroy the symmetric structure of the original mesh.

Sparse supernodal solver using Block Low-Rank compression. In Chapter 3, we present how BLR compression was introduced in the PaStiX solver. We developed two strategies: **Minimal Memory**, which focuses on reducing the memory consumption, and **Just-In-Time**, which focuses on reducing the time-to-solution (factorization and solve steps). During the factorization, the first strategy compresses the sparse matrix before factorizing it, **i.e.**, compresses A factors, and exploits dedicated low-rank numerical operations to keep the memory cost of the factorized matrix as low as possible. The second strategy compresses the information as late as possible, **i.e.**, compresses L factors, to avoid the cost of low-rank update operations. The resulting solver can be used either as a direct solver for low accuracy solutions or as a high-accuracy preconditioner for iterative methods, requiring only a few iterations to reach the machine precision.

Reordering strategy to reduce the number of off-diagonal blocks. In Chapter 4, we study blocking strategies to increase the average size of off-diagonal blocks. The objective is to better take advantage of modern architectures with larger blocks of data and to reduce the burden on small granularity of low-rank structures for the strategies presented in Chapter 3. We propose an algorithm that reorders the unknowns of the problem to increase the average size of the off-diagonal blocks in block-symbolic factorization structures. The major feature of this solution is that, based on an existing nested dissection ordering for a given problem, our solution will keep constant the amount of fill-in generated during the factorization. So, the amount of memory and computation to store and compute the factorized matrix is invariant. The consequence of this increased average size is that the number of off-diagonal blocks is largely reduced. It diminishes the memory overhead of the

data structures used by the solver and the number of tasks required to compute the solution in task-based implementations [4, 74], increasing the performance of BLAS kernels.

Block Low-Rank clustering. In Chapter 5, we study the impact of clustering heuristics on the solver, to enhance low-rank strategies presented in Chapter 3. We analyze the widely used k-way partitioning with respect to the reordering strategy presented in Chapter 4 and propose a new heuristic to perform algebraic block low-rank clustering based on both approaches. A good clustering of unknowns has two main criteria to respect: 1) the diameter of clusters has to be small and 2) the number of neighbors in the clustering has to be constant. We investigate how sparse constraints, managing correctly data granularity, and existing clustering strategies for dense matrices can be coupled.

Separators alignment. In Chapter 6, we propose a modified nested dissection to align separators. The nested dissection process was originally designed for reducing both the number of operations and the memory consumption of sparse direct solvers. However, it was not designed to obtain large blocks or suitable low-rank structures. When using nested dissection, a graph is split into $A \cup B \cup C$ where C is the separator and the same process is applied independently to both A and B. Thus, the pattern of sparse contributions on C will not be symmetric for both subparts. For instance, in Figure 2.1, a perfect nested dissection is presented, in the sense that separators are as small as possible and split the graph among equal-size subparts. However, second-level separators are not aligned, which can degrade the sparsity pattern since there are more type of contribution patterns. The objective is to reduce the burden on ordering strategies by exhibiting more symmetric data structures. Such an approach should enhance the reordering strategy presented in Chapter 4 as well as the clustering strategies studied in Chapter 5, since contributions will be more similar. Aligning separators can be seen as a solution to exhibit simultaneously suitable sparse structures and efficient low-rank clustering approaches.

Complementary studies. Finally, additional studies related to this thesis are presented in appendices. In Appendix A, we present the environmental context on which experiments were performed. In Appendix B, we study the use of low-rank compression in PaStiX together with the use of Parsec and StarPU runtime systems. It demonstrates the assets of using runtime systems, which can perform dynamically load balancing. It is especially interesting when using irregular kernels, such as those appearing with low-rank compression. In Appendix C, we integrate the low-rank solver into a domain decomposition solver and study its behavior for a large real-life application. We evaluate how a high accuracy preconditioner behaves with respect to the full-rank version of the solver.

Chapter 3

Sparse supernodal solver using Block Low-Rank Compression

Low-rank compression techniques allows reducing memory consumption and/or time-to-solution while controlling the error made on the approximation. In this chapter, we describe the first contribution of this thesis which is a BLR solver developed within the PaStiX library. The modifications of the PaStiX solver are presented in Section 3.1, with two strategies, a first one that maximizes memory consumption reducing and a second one that reduces time-to-solution. In Section 3.2, we detail the basic kernels that are used to manage compressed blocks in a supernodal solver. Section 3.3 compares the two BLR strategies with the original approach, that uses only dense blocks, in terms of memory consumption, time-to-solution and numerical behavior. We also investigate the efficiency of low-rank kernels, as well as the impact of the BLR solver parameters. Section 3.4 surveys in more detail related works on BLR for dense and/or sparse direct solvers, highlighting the differences with our approach, before discussing how to extend this work to a hierarchical format ($\mathcal{H}$, HSS, HODLR...). In Section 3.5, we discuss the global behavior of low-rank compression in the PaStiX solver and propose some future works.

3.1 Block Low-Rank solver

In this section, we first introduce the notation used for the factorization that will serve to integrate low-rank blocks in the solver. Then, using the newly introduced structure, we describe two different strategies leading to a sparse direct solver that optimizes the memory consumption and/or the time-to-solution.

3.1.1 Notation

Let us consider the symbolic block structure of a factorized matrix L, obtained through the block-symbolic factorization presented in Section 1.1.2. Initially, we allocate this structure initialized with the entries of A and perform an in-place factorization. We denote initial blocks by A and blocks in their final state by L (or U). The matrix is composed of N_{cblk} column blocks, where each column block is

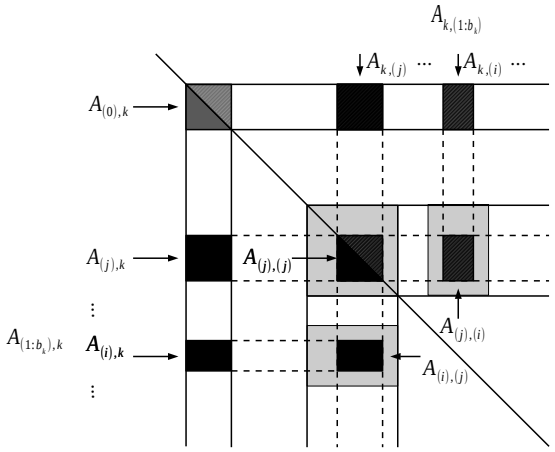


Figure 3.1: Symbolic block structure and notation used for the algorithms for the column block of index k, and its associated blocks. The larger gray boxes highlight the fact that the contributions modify only a part of the target blocks.

associated with a supernode, or to a subset of unknowns in a supernode when the latter is split to create parallelism. Each column block k is composed of b_k+1 blocks, as presented in Figure 3.1 where:

- $A_{(0),k}(= A_{k,(0)})$ is the dense diagonal block;
- $A_{(j),k}$ is the j^{th} off-diagonal block in the column block with $1 \leq j \leq b_k$, (j) being a multi-index describing the row interval of each block, and respectively, $A_{k,(j)}$ is the j^{th} off-diagonal block in the row block;
- $A_{(1:b_k),k}$ represents all the off-diagonal blocks of the column block k, and $A_{k,(1:b_k)}$ all the off-diagonal blocks of the symmetric row block;
- $A_{(i),(j)}$ is the rectangular dense block corresponding to the rows of the multi-index (i) and to the columns of the multi-index (j).

In addition, we denote by $\hat{A}$ the compressed representation of a matrix A.

3.1.2 Sparse direct solver using BLR compression

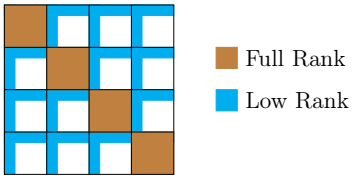


Figure 3.2: Block Low-Rank compression of a dense matrix using four clusters. Diagonal blocks are kept dense while off-diagonal blocks are represented in a low-rank form.

The BLR compression scheme is a flat, non-hierarchical format, unlike others formats mentioned in Section 1.2.1.4. If we consider the example of a dense matrix, the BLR format clusters the matrix into a set of smaller blocks, as presented in Figure 3.2. Diagonal blocks are kept dense and off-diagonal blocks are admissible for compression. Thus, these off-diagonal blocks can be represented through a low-rank form UV^t, obtained with a compression technique such as Singular Value Decomposition (SVD) or Rank-Revealing QR (RRQR) factorization. Compression techniques were detailed in Section 1.2.2

We propose in this chapter to apply this scheme to the symbolic block structure of sparse direct solvers. Firstly, diagonal blocks of the largest supernodes in the block elimination tree can be considered as large dense matrices which are compressible with the BLR approach. In fact, as we have seen in Section 1.1.2, it is common to split these supernodes into a set of smaller column blocks in order to increase the level of parallelism. Thus, the block structure resulting from this operation gives the clusters of the BLR compression format. Secondly, interaction blocks from two large supernodes are by definition long distance interactions, and thus can be represented by a low-rank form. It is then natural to store them as low-rank blocks as long as they are large enough. To summarize, if we take the final symbolic block structure (after splitting) used by the PaStiX solver, all diagonal blocks are considered dense, and all off-diagonal blocks might be stored using a low-rank structure. In practice, we limit this compression to blocks of a minimal size, and all blocks with relatively high ranks are kept dense.

From the original block structure, adapting the solver to block low-rank compression mainly relies on the replacement of the dense operations with the equivalent low-rank operations. Still, different variants of the final algorithm can be obtained by changing **when and how** the low-rank compression is applied. We introduce two scenarios: **Minimal Memory**, which compresses the blocks before any other operations, and **Just-In-Time**, which compresses the blocks after they received all their contributions.

3.1.2.1 Minimal Memory

This scenario, described by Algorithm 3, starts by compressing the original matrix A block by block without allocating the full matrix as it is done in the full-rank and **Just-In-Time** strategies. Thus, all low-rank blocks that are large enough are compressed directly from the original sparse form to the low-rank representation (lines $1-4$). Note that for a matter of conciseness, loops of compression and solve over all off-diagonal blocks are merged into a single operation. In this scenario, compression kernels and later operations could have been performed using a sparse format, such as CSC for instance, until we get some fill-in. However, for the sake of simplicity we use a low-rank form throughout the entire algorithm to rely on blocks and not just on sets of values. Then, each classic dense operation on a low-rank block is replaced by a similar kernel operating on low-rank forms, even for the usual matrix-matrix multiplication (**GEMM**) kernel that is replaced by the equivalent **LR2LR** kernel operating on three low-rank matrices (cf. Section 3.2).

Algorithm 3 Right looking block sequential LU factorization with **Minimal Memory** scenario.

▷ /* Initialize A (L structure) compressed */
1: **For** $k = 1$ to N_{cblk} **Do**
2: $\hat{A}_{(1:b_k),k} = \text{Compress}(\ A_{(1:b_k),k}\ )$
3: $\hat{A}_{k,(1:b_k)} = \text{Compress}(\ A_{k,(1:b_k)}\ )$
4: **End For**
5: **For** $k = 1$ to N_{cblk} **Do**
6: Factorize $A_{(0),k} = L_{(0),k}U_{k,(0)}$
7: Solve $\hat{L}_{(1:b_k),k}\, U_{k,(0)} = \hat{A}_{(1:b_k),k}$
8: Solve $L_{(0),k}\, \hat{U}_{k,(1:b_k)} = \hat{A}_{k,(1:b_k)}$
9: **For** $j = 1$ to b_k **Do**
10: **For** $i = 1$ to b_k **Do**
▷ /* LR to LR updates (extend-add) */
11: $\hat{A}_{(i),(j)} = \hat{A}_{(i),(j)} - \hat{L}_{(i),k}\hat{U}_{k,(j)}$ ▷ **LR2LR**
12: **End For**
13: **End For**
14: **End For**

Algorithm 4 Right looking block sequential LU factorization with **Just-In-Time** scenario.

1: **For** $k = 1$ to N_{cblk} **Do**
2: Factorize $A_{(0),k} = L_{(0),k}U_{k,(0)}$
▷ /* Compress L and U off-diagonal blocks */
3: $\hat{A}_{(1:b_k),k} = \text{Compress}(\ A_{(1:b_k),k}\ )$
4: $\hat{A}_{k,(1:b_k)} = \text{Compress}(\ A_{k,(1:b_k)}\ )$
5: Solve $\hat{L}_{(1:b_k),k}\, U_{k,(0)} = \hat{A}_{(1:b_k),k}$
6: Solve $L_{(0),k}\, \hat{U}_{k,(1:b_k)} = \hat{A}_{k,(1:b_k)}$
7: **For** $j = 1$ to b_k **Do**
8: **For** $i = 1$ to b_k **Do**
▷ /* LR to dense updates */
9: $A_{(i),(j)} = A_{(i),(j)} - \hat{L}_{(i),k}\hat{U}_{k,(j)}$ ▷ **LR2GE**
10: **End For**
11: **End For**
12: **End For**

3.1.2.2 Just-In-Time

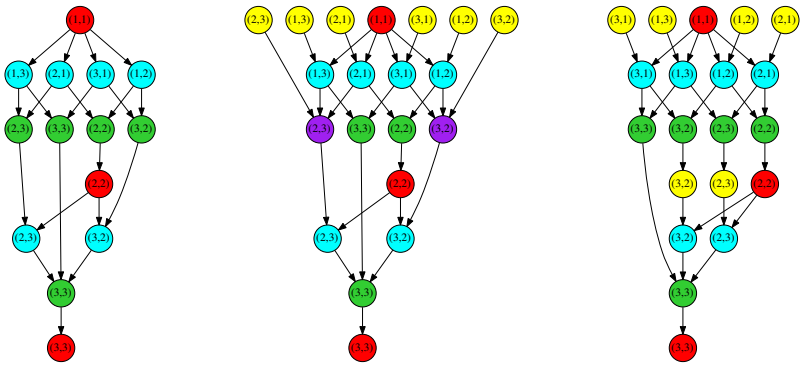
This second scenario, described by Algorithm 4, delays the compression of each supernode after all contributions have been accumulated. The algorithm is thus really close to the previous one with the only difference being in the update kernel, **LR2GE**, at line 9, which accumulates contributions on a dense block, and not on a low-rank form.

This operation, as we describe in Section 3.2.2, is much simpler than the **LR2LR** kernel, and is faster than a classic **GEMM**. However, by compressing the initial

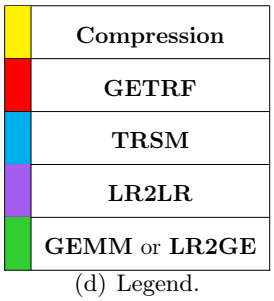
matrix A, and maintaining the low-rank structure throughout the factorization with the **LR2LR** kernel, **Minimal Memory** can reduce more drastically the memory footprint of the solver. Indeed, the full-rank structure of the factorized matrix is never allocated, as opposed to **Just-In-Time** that requires it to accumulate the contributions. The final matrix is compressed with similar sizes in both scenarios.

3.1.2.3 Summary

For the sake of simplicity, we now compare the three strategies in a dense case to illustrate the potential of different approaches. Figure 3.3(a) presents the Directed Acyclic Graph (DAG) of LU operations for the original full-rank version of the solver on a 3-by-3 block matrix. On each node, the couple (i, j) represents respectively the row and column indexes of the block of the matrix being modified.



(a) Full-rank. (b) **Minimal Memory**. (c) **Just-In-Time**.



(d) Legend.

Figure 3.3: DAGs of full-rank and both low-rank strategies for the factorization of a 3-by-3 dense block matrix.

The **Minimal Memory** strategy generates the DAG described in Figure 3.3(b). In this context, the six off-diagonal blocks are compressed at the beginning. In the update process, dense diagonal blocks as well as low-rank off-diagonal blocks are updated. One can notice that off-diagonal blocks are never used in their dense form, leading to memory footprint reduction. However, as we will describe in Section 3.2, the **LR2LR** operation in sparse arithmetic is quite expensive and may lead to an increase of time-to-solution.

The **Just-In-Time** strategy generates the DAG described in Figure 3.3(c). In this case, the six off-diagonal blocks are compressed throughout the factorization. Since those off-diagonal blocks are used in their dense form before being compressed, there is less room for memory improvement. However, as we will see later, the **LR2GE** can still be performed efficiently in the sparse case.

3.2 Low-rank kernels

We introduce in this section the low-rank kernels used to replace the dense operations, and we present a complexity study of these kernels. Two families of operations are studied to reveal the rank of a matrix: Singular Value Decomposition (SVD) which leads to smaller ranks, and Rank-Revealing QR (RRQR) which has shorter time-to-solution.

Note that for the **Minimal Memory** scenario, the first compression (of sparse blocks) may be realized using Lanczos's methods, to take advantage of sparsity. However, both SVD and RRQR algorithms inherently take advantage of these zeroes. In addition, most of the low-rank compression are applied to blocks stored as dense blocks and represent the main part of the computation.

3.2.1 Solve

The solve operation for a generic lower triangular matrix L is applied to blocks in low-rank forms in our two scenarios: $L\hat{x} = \hat{b} \Leftrightarrow LU_xV_x^t = U_bV_b^t$. Then, with $V_x^t = V_b^t$, the operation is equivalent to applying a dense solve only to U_b, and the complexity is only $\Theta(m_L^2 r_x)$, instead of $\Theta(m_L^2 n_L)$ for the full-rank (dense) representation.

In practice, the solve operations that appear in Algorithm 3 and in Algorithm 4 are not exactly the same. As the U factors are stored and compressed in a transposed form, the operations on lines $7-8$ of Algorithm 3 (and lines $5-6$ for Algorithm 4) keep untouched the U bases instead of V bases. Thus, if the original low-rank representation of those blocks has orthogonal U bases (which will be necessary in the following), this property is kept invariant.

3.2.2 Update

Let us consider the generic update operation, $C = C - AB^t$. Note that the PaStiX solver stores L, and U^t if required. Then, the same update is performed for Cholesky and LU factorizations. We break the operation into two steps: the product of two low-rank blocks, and the addition of a low-rank block to either a dense block (**LR2GE**), or a low-rank block (**LR2LR**).

3.2.2.1 Low-rank matrix product

This operation can simply be expressed as two dense matrix products:

$$\hat{A}\hat{B}^t = (U_A(V_A^tV_B))U_B^t = U_A((V_A^tV_B)U_B^t),$$

where U_A is kept unchanged if $r_A \leq r_B$ (U_B^t is kept otherwise) to lower the complexity.

However, it has been shown in [16] that the rank r_{AB} of the product of two low-rank matrices of ranks r_A and r_B is usually smaller than $\min(r_A, r_B)$. As U_A and U_B are both orthogonal, the matrix $E = (V_A^t V_B)$ has the same rank as $\hat{A}\hat{B}^t$. Thus, the complexity can be further reduced by transforming the matrix product to the following series of operations:

$$E = V_A^t V_B \tag{3.1}$$
$$\hat{E} = V_A^t \hat{V_B} = U_E V_E^t \tag{3.2}$$
$$U_{AB} = U_A U_E \tag{3.3}$$
$$V_{AB}^t = V_E^t U_B^t. \tag{3.4}$$

3.2.2.2 Low-rank matrix addition

Let us consider the next generic operation $C' = C - U_{AB}V_{AB}^t$, with $m_{AB} \leq m_C$ and $n_{AB} \leq n_C$ as it generally happens in the supernodal method. This is illustrated for example by the update block $A_{(i),(j)}$ in Figure 3.1.

If C is not compressed, as it happens in the **LR2GE** kernel, C' will be dense too, and the addition of the two matrices is nothing else than a **GEMM** kernel. The complexity of this operation grows as $\Theta(m_{AB}\ n_{AB}\ r_{AB})$.

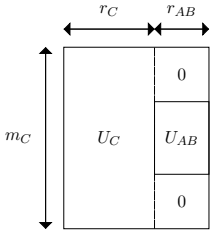


Figure 3.4: Accumulation of two low-rank matrices when sizes do not match. In this example, a large matrix C receives a smaller contribution AB, and their respective U bases are accumulated using zero padding.

If C is compressed as in the **LR2LR** kernel, C' will be compressed too, and

$$\hat{C'} = U_C V_C^t - U_{AB} V_{AB}^t \tag{3.5}$$
$$U_{C'} V_{C'}^t = [U_C, U_{AB}]([V_C, -V_{AB}])^t \tag{3.6}$$

where $[,]$ is the concatenation operator. The update given at Equation (3.5) is the commonly named **extend-add** operation. Without further optimization, this operation costs only two copies. In the case of the supernodal method, adequate padding is also required to align the vectors coming from the AB and C matrices as shown in Figure 3.4 for the U vectors. The operation on V is similar.

One can notice that in this form, the rank of the updated C is now $r_C + r_{AB}$. When accumulating multiple updates, the rank grows quickly and the storage exceeds the full-rank version. In order to maintain a small rank for C, recompression techniques are used. As for the compression kernel, both SVD and RRQR algorithms can be used.

Recompression using SVD We start by computing a QR decomposition for both composed matrices:

$$[U_C, U_{AB}] = Q_1 R_1 \text{ and } [V_C, -V_{AB}] = Q_2 R_2. \tag{3.7}$$

Then, the temporary matrix $T = R_1 R_2^t$ is compressed using the SVD algorithm described previously. This gives the final C' with:

$$U_{C'} = (Q_1 U_T) \text{ and } V_{C'} = (Q_2 V_T). \tag{3.8}$$

The complexity of this operation is decomposed as follows: $\Theta((m_C + n_C)(r_C + r_{AB})^2)$ for the two QR decompositions of Equation (3.7), $\Theta((r_C + r_{AB})^3)$ for the SVD decomposition, and finally $\Theta((m_C + n_C)(r_C + r_{AB})r_{C'})$ for the application of both Q_1 and Q_2.

Recompression using RRQR This solution takes advantage of the fact that U_C is orthonormal to first orthogonalize U_{AB} with respect to U_C. For this operation, we refer to Section 3.2.2.3 to form an orthonormal basis $[U_C, \overline{U_{AB}}]$ such that

$$[U_C, U_{AB}] = [U_C, \overline{U_{AB}}]\ T. \tag{3.9}$$

We follow by applying the RRQR algorithm to:

$$T \left([V_C, -V_{AB}] \right)^t = QRP. \tag{3.10}$$

As for the compression, we keep the $k = r_{C'}$ first columns of Q and rows of R to form the final C':

$$U_{C'} = ([U_C, \overline{U_{AB}}]\ Q_k) \text{ and } V_{C'}^t = R_k P. \tag{3.11}$$

Note that $U_{C'}$ is kept orthonormal for future updates.

When the RRQR algorithm is used, the complexity of the recompression is then composed of: $\Theta(r_C\ r_{AB}\ m_{AB})$ to form the intermediate product $U_C^t\ U_{AB}$, $\Theta(m_C\ r_C\ r_{AB})$ to form the orthonormal basis, $\Theta(n_{AB}\ r_{AB}\ r_C)$ to generate the temporary matrix used in (3.10), $\Theta((r_C + r_{AB})n_C\ r_{C'})$ to apply the RRQR algorithm, and finally again $\Theta((r_C + r_{AB})n_C\ r_{C'})$ to compute the final $U_{C'}$.

3.2.2.3 Orthogonalization

Let us consider the orthogonalization of $[U_C, U_{AB}] = QT$, taking advantage of U_C orthogonality. The following recompression of $T[V_C, -V_{AB}]^t$ will now be referred to as **right recompression**.

A main issue is that $[U_C, U_{AB}]$ may not be full-rank in exact arithmetic if both matrices share a common spectrum. Thus, if some zero columns can be removed, it will reduce the cost of the **right recompression**, by ignoring zero rows. In practice, for each zero column, we permute both $[U_C, U_{AB}]$ and $[V_C, -V_{AB}]$ matrices and reduce the rank involved in next computations.

The objective is to reduce the number of operations depending on r_C, considering that in many cases $r_{AB} < r_C$ as many large off-diagonal blocks receive small

contributions. In order to do so, we perform Gram-Schmidt (GS) [103] projections to take advantage of U_C orthogonality. Two variants of GS are widely used: Classical Gram-Schmidt (CGS) and Modified Gram-Schmidt (MGS). Both may have stability issues, and several iterations may be performed to ensure a correct (at machine precision) orthogonality. In practice, we use CGS for its locality advantages and perform a second iteration if required. To verify that a second iteration is required, we use a widely used criterion [50] and did not experience any orthogonality issue for the set of problems we consider.

We now present several variants for orthogonalization.

Householder QR factorization. This method performs a Householder QR factorization on the full matrix $[U_C, U_{AB}] = Q_1 R_1$. Thus, it does not exploit the existing orthogonality in U_C, and cannot properly extract zero columns from the final solution to reduce the cost of later operations. However, it is probably the most stable approach and is the mostly tuned kernel in linear algebra libraries such as Intel MKL. The complexity of this operation grows as $\Theta(m_C(r_C + r_{AB})^2)$.

PartialQR. It performs a projection of U_{AB} into U_C:

$$\tilde{U_{AB}} = U_{AB} - U_C(U_C^t \, U_{AB}). \tag{3.12}$$

From this projection, we obtain:

$$[U_C, U_{AB}] = [U_C, \tilde{U_{AB}}] \begin{pmatrix} I & U_C^t U_{AB} \\ 0 & I \end{pmatrix}. \tag{3.13}$$

In practice, we perform two projections to ensure the stability. We now have $U_C \perp \tilde{U_{AB}}$ and want to orthogonalize $\tilde{U_{AB}}$ to obtain an orthogonal basis. In order to do so, we either perform a Householder QR factorization $\tilde{U_{AB}} = \overline{U_{AB}} R$ or a rank-revealing QR factorization at the machine precision $\tilde{U_{AB}} = Q_k R_k P$ to remove zero columns. The first approach can make a good use of Level 3 BLAS operations while the second is less efficient but may construct zero columns and then reduce the right recompression.

Classical Gram-Schmidt. In this variant, we orthogonalize one-by-one each vector of U_{AB} with one or two CGS iterations depending on the criterion presented before. The orthogonalization of the i-th column of U_{AB} to obtain the corresponding column of $\overline{U_{AB}}$ is performed with:

$$\overline{U_{AB_i}} = U_{AB_i} - U_{AB_{0:i-1}} U_{AB_{0:i-1}}^t U_{AB_i}. \tag{3.14}$$

Each column is then removed after its orthogonalization if it is a **zero** column. This reduces the final rank of the update and the cost of the following operations.

In the experimental study, we will compare the three approaches in terms of operation count and performance. For orthogonality, we did not observe any numerical issue in our test problems. Note that for both PartialQR and CGS, only last r_{AB} rows of $([V_C, -V_{AB}])^t$ are updated. For those two methods, the orthogonalization complexity grows as $\Theta(m_C \; r_C \; r_{AB})$.

3.2.3 Summary

Table 3.1 summarizes the computational complexity for the two low-rank strategies with respect to the original version of the solver. The main factor of complexity is computed under the assumption that $m_C \geq m_A \geq m_B$, $r_A \geq r_B$, $m_C \geq n_C$, and $r_C \leq r_{C'}$. It does not depend on n_A but on the ranks r_A and r_B: there are fewer operations to be performed. The **Minimal Memory** strategy requires using either the SVD or RRQR recompression, for which the complexity depends on m_C and n_C, the dimensions of the block C. This explains why this strategy may appear of higher complexity than the original solver.

When considering dense matrices, the low-rank matrix (blocks) are updated by contributions of the same size. In that case, the complexity of the low-rank update becomes asymptotically cheaper than the full-rank updates, and leads to performance gain. This is exploited in dense BLR solvers as [16], and in the CUFS (Compress, Update, Factor, Solve) strategy of the BLR-MUMPS solver, which compresses a dense front before applying operations between low-rank blocks of the same size.

In the supernodal approach, blocks belonging to last levels separators receive many small contributions. Thus, $r_{C'}$ is often close to or equal to r_C and lower than $r_C + r_{AB}$: the rank is often invariant when applying a small contribution, which makes RRQR recompression more efficient than SVD recompression, and especially when an orthogonalization method that reduces the basis *on the fly* is used.

In terms of complexity, it is important to notice that the **LR2LR** update depends on the size of the target block C, and not on the contribution size as in full-rank. Thus, low-rank contributions between blocks of similar sizes are asymptotically cheaper than full-rank contributions on these blocks. Dense and multifrontal solvers exploit this property as they work only on regular block sizes. At the opposite, small updates to larger blocks, that regularly appear in supernodal solvers, have a higher cost in low-rank than in full-rank. In summary, we can decompose updates in two groups of contributions:

- the updates that will occur within each supernode at the top of the elimination tree and which are generated by regular split of the supernode to generate parallelism, as well as updates from direct descendants of the sons. These contributions work on regular sizes and will reduce the complexity of the solver as observed in dense and multifrontal solvers;

- the updates from deeper descendants in the elimination tree that generate updates from small blocks to larger nodes of the elimination tree. In these contributions, the complexity is defined by the target block, and thus the cost of the low-rank updates may increase the complexity of the solver with respect to the full-rank solver on small problems.

The experiments showed that this overhead slows down the solver with the **Minimal Memory** strategy on small problems, but when the problem size increases this strategy outperforms the full-rank factorization.

The main advantage of the **Minimal Memory** scenario is that it can drastically reduce the memory footprint of the solver, since it compresses the matrix before the factorization. Thus, the structure of the full-rank factorized matrix is never

Table 3.1: Summary of the operation complexities when computing $C = C - AB^t$.

	GEMM (**Dense**)	**LR2GE (Just-In-Time)**		**LR2LR (Minimal Memory)**	
		SVD	RRQR	SVD	RRQR
LR matrix product	–	(3.1): $\Theta(n_A\ r_A\ r_B)$ (3.2): $\Theta(r_A^2\ r_B)$ (3.2): $\Theta(r_A\ r_B\ r_{AB})$ (3.3), (3.4): $\Theta(m_A\ r_A\ r_{AB})$		(3.1): $\Theta(n_A\ r_A\ r_B)$ (3.2): $\Theta(r_A^2\ r_B)$ (3.2): $\Theta(r_A\ r_B\ r_{AB})$ (3.3), (3.4): $\Theta(m_A\ r_A\ r_{AB})$	
LR matrix addition	–	–		(3.7): $\Theta(m_C(r_C + r_{AB})^2)$ (3.12): $\Theta(m_C\ r_C\ r_{AB})$ (SVD): $\Theta((r_C + r_{AB})^3)$ (3.10): $\Theta(n_C(r_C + r_{AB})r_{C'})$ (3.8): $\Theta(m_C(r_C + r_{AB})r_{C'})$ (3.11): $\Theta(m_C(r_C + r_{AB})r_{C'})$	
Dense update	$\Theta(m_A\ m_B\ n_A)$	$\Theta(m_A\ m_B\ r_{AB})$		–	–
Main factor	$\Theta(m_A\ m_B\ n_A)$	$\Theta(m_A\ m_B\ r_{AB})$	$\Theta(m_A\ m_B\ r_{AB})$	$\Theta(m_C(r_C + r_{AB})^2)$	$\Theta(m_C(r_C + r_{AB})r_{C'})$

allocated. However, the low-rank structure needs to be maintained throughout the factorization process to lower the memory peak.

3.2.4 Kernel implementation

In practice, as we manage both dense and low-rank blocks in our solver, we adapt the extend-add operation for each basic case to be as efficient as possible. The optimizations are designed for the RRQR version, in practice SVD is only used as a reference for the optimal compression rates.

One of the important criteria for efficient low-rank kernels is the setup of the maximum rank above which the matrix will be considered as non compressible. The setting of this parameter is strategy and application dependent. In practice for an m-by-n matrix, if the main objective is the memory consumption, as in the **Minimal Memory** strategy, the ranks are limited to $\frac{mn}{m+n}$ to reduce as much as possible the final size of the factors without considering the number of flops that are generated. In contrast, if the objective is to reduce the time-to-solution, then the maximum ranks are defined by $\frac{min(m,n)}{4}$ for which the low-rank operations remains cheaper in number of operations than the full-rank ones. In practice, it also depends on the ranks of the blocks that will be involved in the update and cannot be computed before compression. For a real-life application, the criterion has to be set depending on the number of factorizations (second criterion is more important) and the number of solves (first criterion will reduce the size of the factors and thus the cost of the solve step). The impact of this parameter is studied in 3.3.6.3.

Eventually, when a low-rank block receives a contribution with a high rank, **i.e.**, $(r_C + r_{AB}) \geq max_{rank}$, the C matrix is decompressed to receive the update and then recompressed, instead of directly adding low-rank matrices. This is denoted as **Decompression / Recompression of** C in the following section.

There is also a trade-off between using a strict compression criterion and a smaller one due to the efficiency of low-rank operations with respect to dense operations. In practice, we add another parameter named rank ratio to control the maximum rank authorized during compression as a percentage of the strict theoretical rank.

3.3 Experiments

In this section, we start with a comparison of compression kernels in Section 3.3.1. We then present the performance of low-rank compression strategies in Section 3.3.2, before analyzing the impact on memory consumption in Section 3.3.3. Convergence and numerical stability are studied in Section 3.3.4. We study parallelism in Section 3.3.5, before analyzing the performance of elementary kernels in Section 3.3.6. The parameters used to tune the solver are summarized in Appendix A.2.

Note that for low-rank strategies, we never perform LL^t factorization because compression can destroy the positive-definite property. In the case where the matrix is SPD, we use LDL^t factorization for low-rank strategies.

3.3.1 SVD versus RRQR

The first experiment studies the behavior of the two compression methods coupled with both **Minimal Memory** and **Just-In-Time** scenarios on the matrix **atmosmodj**. Table 3.2 presents the sequential timings of each operation of the numerical factorization with a tolerance of 10^{-8}, as well as the memory used to store the final coefficients of the factorized matrix.

Table 3.2: Cost distribution on the **atmosmodj** matrix with $\tau = 10^{-8}$ for full-rank and both low-rank strategies. SVD compression kernel is studied to analyze the optimal memory consumption gain while RRQR is presented for the time-to-solution analysis.

	Full-rank	**Just-In-Time**		**Minimal Memory**	
		SVD	RRQR	SVD	RRQR
Factorization time (s)					
Compression	-	4.1e+02	3.4e+01	1.8e+02	5.6+00
Block factorization (GETRF)	7.2e-01	7.4e-01	7.3e-01	7.8e-01	7.6e-01
Panel solve (TRSM)	1.7e+01	6.9e+00	7.4e+00	7.6e+00	7.9e+00
Update					
Formation of contribution	-	-	-	9.9e+01	4.2e+01
Addition of contribution	-	-	-	3.0e+03	7.3e+02
Dense udpate (GEMM)	4.6e+02	1.3e+02	9.7e+01	2.8e+01	2.4e+01
Total	**4.7e+02**	**5.5e+02**	**1.4e+02**	**3.6e+03**	**8.1e+02**
Solve time (s)	6.3e+00	1.9e+00	3.0e+00	**1.5e+00**	3.2e+00
Factor final size (GB)	16.3	6.95	7.49	**6.85**	7.31
Memory peak for the factors (GB)	16.3	16.3	16.3	**6.85**	7.31

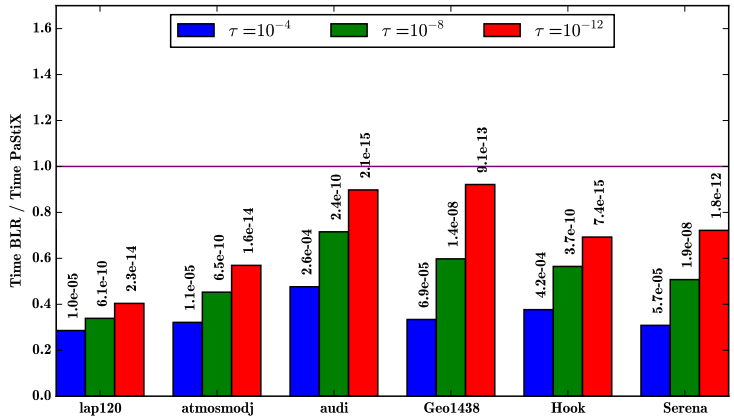
We can first notice that SVD compression kernels are much more time-consuming than the RRQR kernels in both scenarios following the complexity study from Section 3.2. Indeed, RRQR compression kernel stops the computations as soon as the rank is found which reduces by a large factor the complexity, and this reduction is reflected in the time-to-solution. However, the SVD allows, for a given tolerance, to get better memory reduction in both scenarios.

Comparing the **Minimal Memory** and the **Just-In-Time** scenarios, the compression time is minimized in the **Minimal Memory** scenario because the compression occurs on the initial blocks which hold more zero and are lower rank than when they have been updated. The time for the update addition, **extend-add** operation, becomes dominant in the **Minimal Memory** scenario, and even explodes when SVD is used. This is expected as the complexity depends on the largest blocks in the addition even for small contributions (see Section 3.2). Note that this ratio will evolve in favor of the **extend-add** operation on larger matrices where the ratio of updates of same size becomes dominant with respect to the number of updates from small blocks. For both compression methods, both scenarios compress the final coefficients with similar rates.

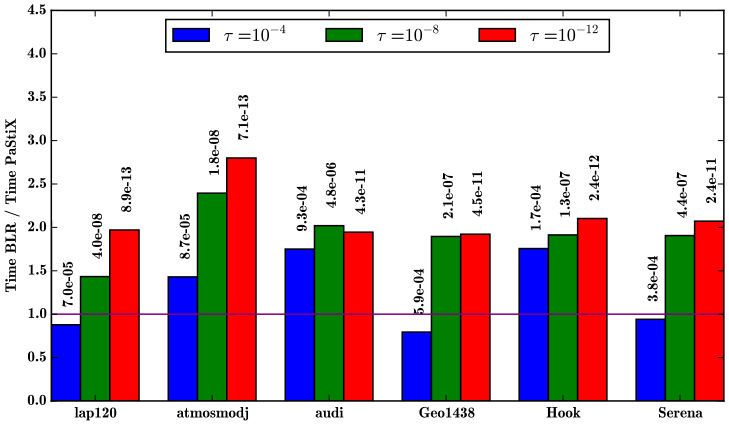
The diagonal block factorization time is invariant in the five strategies: the block sizes and kernels are identical. Panel solve, update product, and solve times are reduced in all low-rank configurations compared to the dense factorization and the timings follow the final size of the factors, since this size reflects the final ranks of the blocks.

To conclude, the **Minimal Memory** scenario is not always able to compete with the original direct factorization on these small test cases due to the costly update addition. However, it reduces the memory peak of the solver to the final size of the factors. The **Minimal Memory**/RRQR offers a 50% memory reduction while doubling the sequential time-to-solution. The **Just-In-Time** scenario competes with the original direct factorization, and divides by three the time-to-solution with RRQR kernels.

3.3.2 Performance



(a) **Just-In-Time** scenario using RRQR.



(b) **Minimal Memory** scenario using RRQR.

Figure 3.5: Performance of both low-rank strategies with three tolerance thresholds. The backward error of the solution is printed on top of each bar.

Figure 3.5 presents the overall performance achieved by the two low-rank scenarios with respect to the original version of the solver (where lower is better) on a set of six matrices. All versions are multi-threaded implementations and use the 24 cores of one node. The scheduling used is the PaStiX static scheduler developed for the original version, this might have a negative impact for low-rank strategies that have an important load imbalance. We study only the RRQR kernels as the SVD kernels have shown to be much slower. Three tolerance thresholds are studied for their impact on the time-to-solution and the accuracy of the backward error. The backward errors printed on top of each bar correspond to the use of one refinement step, obtained with $\frac{||Ax-b||_2}{||b||_2}$.

Figure 3.5(a) shows that the **Just-In-Time**/RRQR scenario is able to reduce the time-to-solution in almost all cases of tolerance, and for all matrices which have a large spectrum of numerical properties. These results show that applications which require low accuracy, as the seismic application for instance, can benefit by up to a factor of 3.5. Figure 3.5(b) shows that it is more difficult for the **Minimal Memory**/RRQR scenario to be competitive. The performance is often degraded with respect to the original PaStiX performance, with an average loss of around a factor 2, and the tolerance has a much lower impact than for the previous case.

For both scenarios, the backward error of the first solution is close to the entry tolerance. It is a little less accurate in the **Minimal Memory** scenario, because approximations are made earlier in the computation, and information is lost from the beginning. However, these results show that we are able to catch algebraically the information and forward it throughout the update process.

3.3.3 Memory consumption

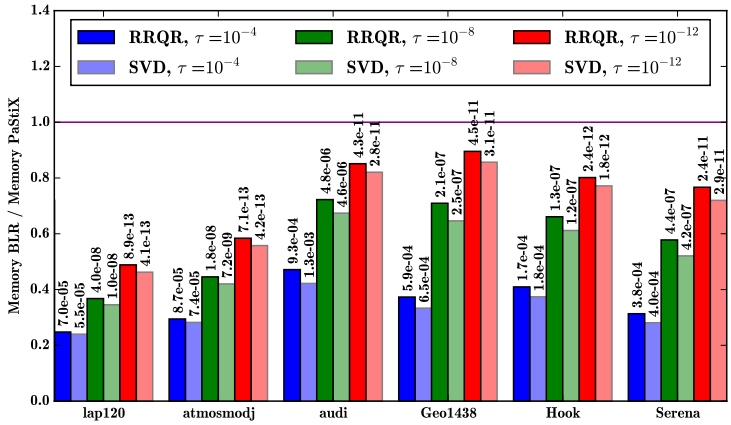


Figure 3.6: Memory peak for the **Minimal Memory** scenario with three tolerance thresholds and both SVD and RRQR kernels. The backward error of the solution is printed on top of each bar.

The **Minimal Memory** scenario is slower than the original solver in most cases, but it is a strategy that efficiently reduces the memory peak of the solver. Figure 3.6

presents the gain in the memory used to store the factors at the end of the factorization of the set of six matrices with respect to the **block dense** storage of PaStiX. In this figure, we also compare the memory gain of the SVD and RRQR kernels. We observe that in all cases, SVD provides better compression rate by finding smaller ranks for a given matrix and a given tolerance. The quality of the backward error is in general slightly better with the SVD kernels despite the smaller ranks. The second observation is that the smaller the tolerance (10^{-12}), the larger the ranks and the memory consumption. However, the solver always presents a memory gain larger than 50% with larger tolerance (10^{-4}).

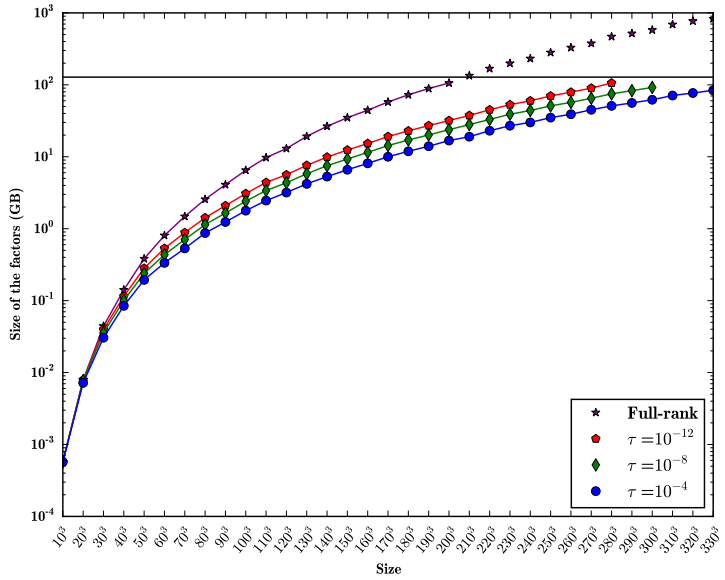
Figure 3.7(a) presents the evolution of the size of the factors as well as the full consumption of the solver (factors and management structures) on 3D Laplacians with an increasing size. The memory limit of the system is 128 GB. The original version is limited on this system to a 3D Laplacian of 8 million unknown, and the size of the factors quickly increases for larger number of unknowns. With the **Minimal Memory**/RRQR scenario, we have now been able to run a 3D problem on up to 36 million unknown when relaxing the tolerance to 10^{-4}. From the same experiment, Figure 3.7(b) presents the number of operations evolution depending on the Laplacians size. One can note that for a small number of unknowns, **Minimal Memory**/RRQR scenario performs more operations than the original, full-rank, factorization. However, for a large number of unknowns, the number of operations is reduced by a large factor. For instance, for a 280^3 Laplacian with a 10^{-8} tolerance, the number of operations is reduced by a factor larger than 36. It demonstrates the potential of our approach: even if operations are less efficient and may lead to a time-to-solution overhead for relatively small problems, the **Minimal Memory** strategy enables the computation of larger problem and reduces its time-to-solution. In this experiment, the **Minimal Memory** strategy becomes faster for Laplacian larger than 150^3 with a 10^{-8} tolerance.

The memory of the **Just-In-Time** scenario has not been studied, because in our supernodal approach, each supernode is fully allocated in a full-rank fashion in order to accumulate the updates before being compressed. Thus, the memory peak corresponds to the totality of the factorized matrix structure without compression and is identical to the original version. To reduce this memory peak, a solution would be to modify the scheduler to a **Left-Looking** approach that would delay the allocation and the compression of the original blocks. However, it would need to be carefully implemented to keep a certain amount of parallelism in order to save both time and memory. A possible solution is the scheduling strategy presented in [2] to keep the memory consumption of the solver under a given limit.

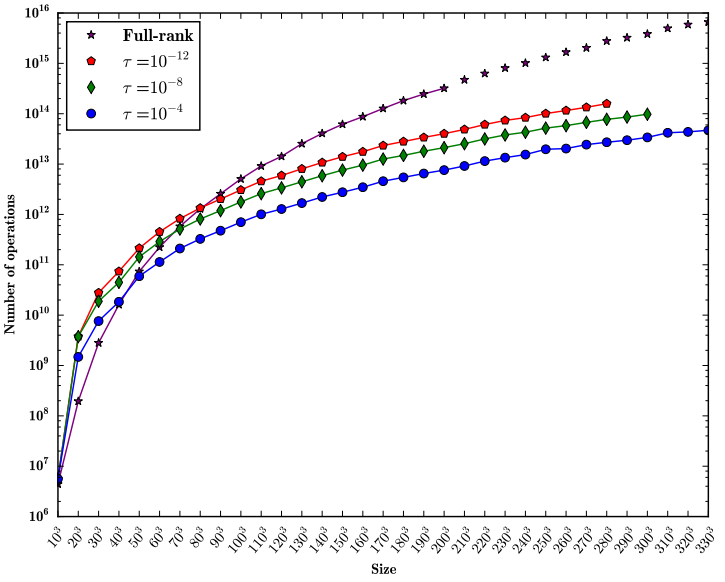
3.3.4 Convergence and numerical stability

Figure 3.8 presents the convergence of the GMRES iterative solver preconditioned with the low-rank factorization at tolerances of 10^{-4} and 10^{-8}. The iterative solver is stopped after reaching 20 iterations or a backward error lower than 10^{-12}.

With a tolerance of 10^{-8}, only a few iterations are required to converge to the solution. Note that on the **audi** and **Geo1438** matrices, which are difficult to compress, a few more iterations are required to converge. With a larger tolerance



(a) Memory.



(b) Number of operations.

Figure 3.7: Scalability of the memory on top and the number of operations on bottom with three tolerance thresholds of the **Minimal Memory**/RRQR scenario for 3D Laplacians of size n^3 with $n \in [10, 330]$. The full-rank scenario, in purple, is given as a reference with the numbers computed from the symbolic factorization.

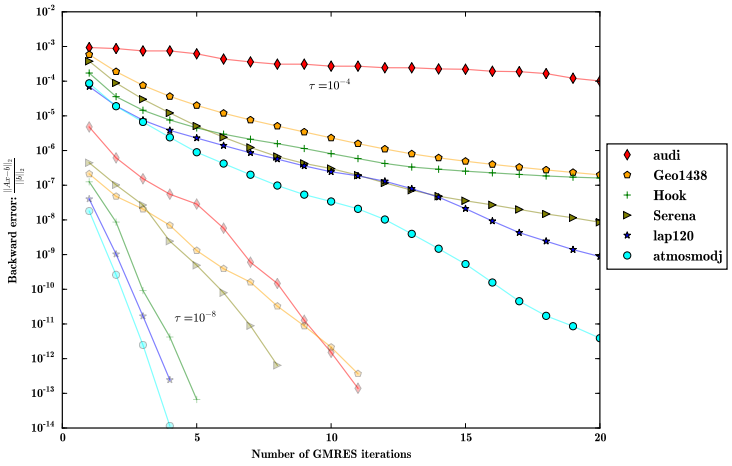


Figure 3.8: Convergence speed for the **Minimal Memory**/RRQR scenario with two tolerance thresholds using GMRES iterations. The set of curves on the top corresponds to $\tau = 10^{-4}$, while the set of curves on the bottom corresponds to $\tau = 10^{-8}$.

10^{-4}, it is difficult to recover all the information lost during the compression, but this is enough to quickly get solutions at 10^{-6} or 10^{-8}. Note that the GMRES process benefits from the compression through the solve step.

3.3.5 Parallelism

For the full-rank factorization, supernodes are split between processors depending on the corresponding number of operations. When a target block C receives a contribution, a lock is used to ensure that the block is not modified simultaneously by several threads. For the **Just-In-Time** strategy, a similar lock is used since the update operations directly apply dense modifications. However, for the **Minimal Memory** strategy, the update operation is decomposed into two parts: **Formation of contribution** and **Addition of contribution**. Because the formation of contribution does not depend on the target block C, the lock is only positioned for the addition of contribution, which may increase the level of parallelism.

Figure 3.9 presents the speedup of the full-rank factorization and low-rank strategies using tolerances of 10^{-4} and 10^{-8} for the **atmosmodj** matrix. The speedup for the full-rank version is above 12, for a relatively small matrix. The speedup of the **Minimal Memory** strategy is above 11, while the speedup of the **Just-In-Time** strategy is around 8. As the supernodes distribution cannot predict the ranks and the corresponding number of operations, load balancing may be degraded. **Minimal Memory** strategy scalability exceeds **Just-In-Time** strategy because there are fewer constraints on locks. Note that on recent architectures, the maximal speedup can not be obtained, as the CPU frequency is reduced when all cores are used on the node, while it is increased for single core operation. We thus computed an approximate maximum speedup of 20.7 on this architecture for the Intel MKL BLAS GEMM operation.

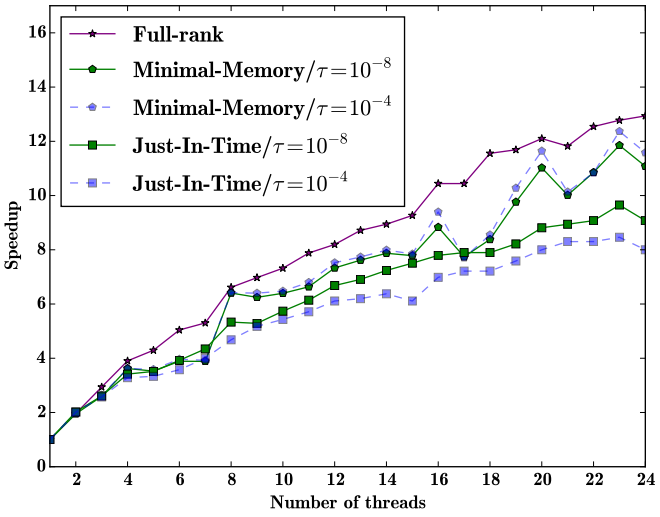


Figure 3.9: Speedup of the factorization for the **atmosmodj** matrix with 1 to 24 threads for full-rank and both low-rank strategies with $\tau = 10^{-4}$ and $\tau = 10^{-8}$.

3.3.6 Kernels analysis

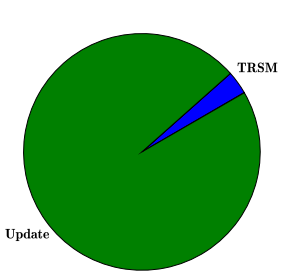
In this section, we focus on the **atmosmodj** matrix to illustrate the performance of basic kernels as well as the impact of several parameters. As in practical cases we use RRQR instead of SVD, we will focus on this compression kernel for each strategy.

3.3.6.1 Performance of basic kernels

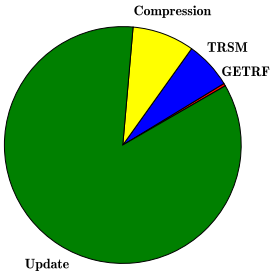
We evaluate the performance rate (in GFlops/s) for the full-rank factorization and for the two low-rank strategies, and use a 10^{-8} tolerance. Figure 3.10(a) (respectively Figure 3.10(b)) presents the runtime distribution among kernels for the full-rank (respectively **Just-In-Time**) factorization. We can note that in both cases, the **Update** process is the most time-consuming. For the **Just-In-Time** strategy, **Compression** and **TRSM** are not negligible, because the **Update** runtime is much reduced with respect to full-rank factorization.

Figure 3.10(c) presents the runtime distribution among kernels for the **Minimal Memory** strategy with three different levels from the left to the right: the main steps of the solver, the details for the **Update** kernels, and the details for the low-rank updates. Note that **xx2fr** refers to a full-rank update within a compressible supernode (**i.e.**, an update to a dense block which was originally considered compressible) while **xx2lr** is a low-rank update, **xx** being one of the four possible matrix products: low-rank/low-rank, low-rank/full-rank, full-rank/low-rank or full-rank/full-rank. **Update dense** corresponds to blocks that were not considered compressible and managed in dense arithmetic throughout all operations; as expected the underlying operations represent a small time of computation.

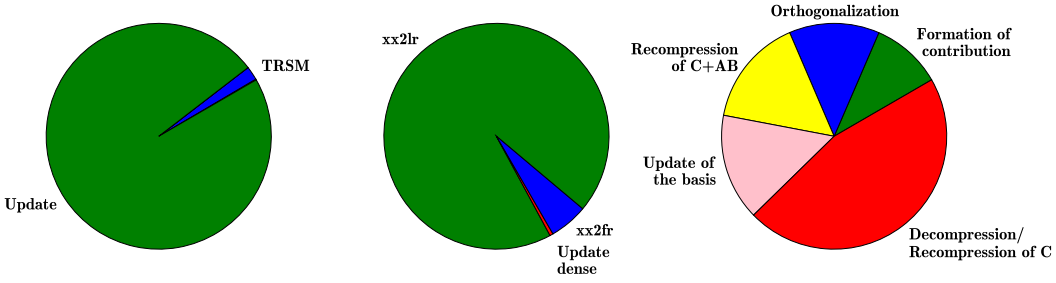
We observe that the low-rank update is the most time-consuming part. In practice, the **Formation of contribution** (cf. Section 3.2.2.1) is quite cheap, while applying the update is expensive; as we have seen in Table 3.1, it depends on the



(a) Full-rank strategy.



(b) **Just-In-Time** strategy.



(c) **Minimal Memory** strategy.

Figure 3.10: Breakdown of the most time-consuming kernels for the full-rank strategy (**top-left**), the **Just-In-Time** strategy (**top-right**), and the **Minimal Memory** strategy (**bottom**) on the **atmosmodj** case using the RRQR kernels with $\tau = 10^{-8}$. The analysis of the **Minimal Memory** strategy is shown at three different levels from the left to the right: the global operations, the partition of the updates between low-rank and full-rank, and the details of the low-rank updates.

size and the rank of the target C. This update addition is decomposed into three main operations: **Orthogonalization** of the $[U_C, U_{AB}]$ matrix (cf. Section 3.2.2.3), **Recompression of** $C+AB$ (cf. Eq (3.2)) and **Update of the basis** (cf. Eq (3.11)). If the contribution rank is too high to take advantage of recompression, we perform a **Decompression / Recompression of** C (cf. Section 3.2.4).

Table 3.3 presents the performance of most time-consuming kernels for each type of factorization on a machine where around 32 GFlops/s can be raised for each CPU core when all cores are used. One can note that the performance of kernels for the full-rank factorization is close to the machine peak: the original solver makes good use of Level 3 BLAS even if there are many small blocks. On the other hand, low-rank kernels performing Level 3 BLAS kernels, **i.e.**, **Formation of contribution**, **Update of the basis**, **TRSM**, are running at $\frac{1}{3}$ of the peak. It is due to lower granularity generated by the smaller blocks, which reduce the arithmetic intensity of the kernels. For **Compression** and **Recompression of C+AB** kernels, the efficiency is even worse due to the behavior of RRQR.

In our implementation, RRQR is a modification of LAPACK `xgeqp3` and `xlaqps` routines. Some stability issues presented in [64] prevent to make efficient use of Level 3 BLAS kernels.

To summarize, low-rank strategies are useful to reduce the overall number of operations. However, due to the poor efficiency of low-rank kernels, the gain in flops does not directly translate into timing reduction. One can expect that for larger problems, the reduction in flops will more easily translate into time-to-solution reduction. It was shown for 3D Laplacian, for which **Minimal Memory** strategy becomes faster than full-rank for Laplacians larger than 150^3 with a 10^{-8} tolerance.

3.3.6.2 Impact of blocking parameters

In previous experiments, we always used a splitting criterion of 256 with a minimal size of 128: blocks larger than 256 are split into a subset of blocks larger than 128. For low-rank strategies, we consider only blocks larger than 128 as compressible. In Table 3.4, we evaluate the impact of using different blocking sizes (between 128 and 256 or between 256 and 512) for the full-rank factorization and for three different tolerances using **Minimal Memory** and **Just-In-Time** strategies. The minimum width of compressible supernodes is set to the minimum blocksize: either 128 or 256. All results are performed using 24 threads on the **atmosmodj** matrix.

The first observation concerns the full-rank factorization. The blocking sizes impact both the granularity and the level of parallelism. For a sequential run, it is suitable to use large blocking sizes to increase granularity and thus the efficiency of Level 3 BLAS operations. On the other hand, it may degrade parallel performance if there are many workers and not enough supernodes.

For the **Minimal Memory** strategy, we have seen in Table 3.1 that the number of operations depends on the target size and rank and thus increases a lot with the blocking size. This is true even in the case where ranks are not that much impacted by the blocking size. In practice, we observe that for each tolerance, increasing blocking size degrades factorization time. In addition, as fewer data is considered as compressible, the size of the factors is growing and this will increase the solve cost.

Table 3.3: Kernels efficiency per core for full-rank, **Just-In-Time** and **Minimal Memory** strategies, on **atmosmodj** with $\tau = 10^{-8}$.

Strategy	Name	Nb calls	Time (s)	GFlops	Perf (GFlops/s)
-	GETRF	74665	981.6	6.4	6.5
Full-rank	TRSM	1572464	17862.0	421.2	23.6
	Update	10993468	446003.1	12869.6	28.9
Just-In-Time	Compression	44544	32831.8	146.6	4.5
	TRSM	1572464	9312.8	111.1	11.9
	Update	10993468	126681.3	1464.1	11.6
Minimal Memory	TRSM	1572464	7355.9	103.8	14.1
	Update				
	dense	5782698	4838.5	24.6	5.1
	xx2fr	2207642	31280.6	285.0	9.1
	xx2lr				
	Formation of contribution	3036024	39319.2	493.7	12.6
	Orthogonalization	2688031	115957.9	630.2	5.4
	Recompression of $C + AB$	2499058	301204.3	755.4	2.5
	Update of the basis	2499058	54132.7	742.5	13.7
	Decompression / Recompression of C	556650	256821.7	2236.3	8.7

Table 3.4: Impact of the blocking size parameter on the number of operations, the factorization time, and the memory for the **atmosmodj** on the Full-rank strategy (top) and on both **Minimal Memory** and **Just-In-Time** strategies (bottom). Both low-rank strategies are using RRQR kernels.

(a) Full-rank strategy.

	Blocksize	128–256	256–512
Full-rank	Operations (TFlops)	12.1	12.1
	Fact. time (s)	**39.5**	51.5
	Memory (GB)	**16.3**	16.5

(b) Low-rank strategies.

	Precision	$\tau = 10^{-4}$		$\tau = 10^{-8}$		$\tau = 10^{-12}$	
	Blocksize	128–256	256–512	128–256	256–512	128–256	256–512
Minimal Memory	Operations (TFlops)	**1.8**	3.9	**4.8**	10.8	**8.3**	18.0
	Fact. time (s)	**49.7**	76.4	**100.1**	166.6	**147.7**	253.0
	Memory (GB)	**4.7**	5.8	**6.53**	7.29	**8.31**	8.81
Just-In-Time	Operations (TFlops)	**0.5**	0.7	**0.8**	1.1	**1.3**	1.5
	Fact. time (s)	12.0	**10.0**	15.6	**14.4**	20.5	**20.2**
	Memory (GB)	**4.9**	6.0	**6.71**	7.47	**8.43**	8.93

For the **Just-In-Time** strategy, the impact of blocking size will mostly depend on ranks. If ranks are small, using larger blocks will increase the performance of RRQR and thus reduce the time-to-solution. However, if ranks are higher, it will reduce the level of parallelism as in the full-rank factorization. We always observe a gain using a larger blocking size, but the ratio between the use of 256/512 versus 128/256 is decreasing when the tolerance is lower: the granularity gain causes some parallelism issues.

3.3.6.3 Impact of rank ratio parameter

In previous experiments, we use strict maximum ranks, **i.e.,** the limit ranks to reduce the number of flops or the memory consumption: $\frac{mn}{m+n}$ for **Minimal Memory** strategy and $\frac{min(m,n)}{4}$ for **Just-In-Time** strategy. In practice, we have seen that for both strategies, kernel efficiency is poor with respect to classical Level 3 BLAS operations. In Table 3.5 we evaluate the impact of relaxing the constraint on a strict rank for the **Minimal Memory** and the **Just-In-Time** strategies. The ratio parameter corresponds to a percentage of the strict maximum rank (used to obtain smaller ranks) and avoid the overhead of managing low-rank blocks with a high rank by turning back these blocks into full-rank form. All results are performed using 24 threads on the **atmosmodj** matrix.

For the **Minimal Memory** strategy, there are two main observations. Firstly, considering that the update complexity depends on the size but also on the rank of the target C, the burden on recompression of high-rank blocks is reduced, and sometimes even the number of operations. Secondly, managing more blocks in full-rank fashion can reduce time-to-solution due to the poor efficiency of low-rank kernels. In practice, we observe that using a relaxed criterion always reduces time-to-solution while having

Table 3.5: Impact of the maximum rank ratio on the number of operations, the factorization time, and the memory for the **atmosmodj** case with RRQR kernels for both **Minimal Memory** and **Just-In-Time** strategies.

	Precision	$\tau = 10^{-4}$			$\tau = 10^{-8}$			$\tau = 10^{-12}$		
	Ratio	1	0.5	0.25	1	0.5	0.25	1	0.5	0.25
Minimal Memory	Operations (TFlops)	**1.8**	1.9	2.3	4.8	4.8	**4.5**	8.3	**7.2**	7.4
	Fact. time (s)	48.7	49.7	**43.7**	100.0	85.4	**56.0**	146.0	105.1	**65.6**
	Memory (GB)	**4.7**	**4.7**	5.4	**6.53**	7.0	8.94	**8.31**	9.26	11.71
Just-In-Time	Operations (TFlops)	**0.5**	0.6	0.9	**0.8**	1.4	2.6	**1.3**	2.8	5.1
	Fact. time (s)	11.9	11.6	**10.9**	**14.2**	15.4	17.8	**19.0**	20.8	25.3
	Memory (GB)	**4.9**	5.0	5.9	**6.71**	7.23	9.12	**8.43**	9.36	11.73

only a little impact on the memory.

For the **Just-In-Time** strategy, the maximum rank criterion cannot be set theoretically, because it depends on all ranks within a same supernode. However, contrary to the **Minimal Memory** approach, the number of operations being really reduced with respect to full-rank factorization, there is a time-to-solution gain and it seems suitable to compress as many blocks as possible. In practice, relaxing the max-rank criterion is only interesting for 10^{-4} tolerance, for which the granularity is really small.

Note that in both cases, relaxing the burden on large ranks increases the size of the factors, and in the same way the cost of the solve. Thus, selecting a suitable criterion will depend on the application and the ratio between the number of factorizations and the number of solves.

3.3.6.4 Orthogonalization cost

In previous **Minimal Memory** experiments, we used CGS as orthogonalization process. As presented in Section 3.2.2.3, some other approaches can be investigated. In Table 3.6, we present the impact of using CGS, Householder QR or PartialQR on the number of operations as well as on the efficiency of the solver. It only impacts intermediate ranks and not the final size of the factors.

Table 3.6: Impact of the orthogonalization method on the number of operations and the factorization time for **Minimal Memory** strategy on **atmosmodj**.

Precision	$\tau = 10^{-4}$			$\tau = 10^{-8}$			$\tau = 10^{-12}$		
Orth. Method	CGS	QR	PartialQR	CGS	QR	PartialQR	CGS	QR	PartialQR
Operations (TFlops)	**1.8**	2.8	2.0	**4.8**	8.0	5.1	**8.3**	13.2	8.7
Fact. time (s)	50.5	60.3	**48.5**	99.1	128.9	**96.3**	145.2	191.8	**138.6**

As predicted by its complexity, Householder QR factorization is more expensive than both CGS and PartialQR. The difference between CGS and PartialQR is related to the number of columns of zero. While CGS can remove those columns during computations, PartialQR can only deal with those zero after all operations: the number of operations increases for each tolerance.

However, in terms of time, PartialQR outperforms CGS especially for small tolerances (10^{-12}) due to the efficiency of Level 3 BLAS operations. One can expect that for a larger tolerance (10^{-4}), CGS may be faster than PartialQR, because the smaller granularity will degrade the assets of Level 3 BLAS operations.

3.4 Positioning with respect to other methods

In this section, we discuss the positioning of our low-rank strategies with the closest related works, and we give some perspectives in extending this work to a hierarchical format.

Sparse $\mathcal{H}$ solver In [55], different approaches using $\mathcal{H}$-matrices are summarized, including the extension to the sparse case. The authors use a nested dissection ordering as in our solver, and thus there is no fill-in between distinct branches of the elimination tree. However, contributions of a supernode to its ancestors are considered as full, in the sense that all structural zero are included to generate the low-rank representation. Thus, they do not have extend-add (**LR2LR**) operations between low-rank blocks of different sizes: assembly is performed as in our **Minimal Memory** scenario, but without zero padding. The memory consumption is higher with their approach because some structural zeroes are not managed: unlike them, we perform a block symbolic factorization to consider sparsity in all contributions between supernodes.

Dense BLR solver A dense BLR solver was designed by Livermore Software Technology Corporation (LSTC) [16]. In this work, the full matrix is compressed at the beginning and operations between low-rank blocks are performed. This approach performs low-rank assembly as in our **Minimal Memory** scenario. As it handles dense matrices, the extend-add process is performed between low-rank matrices of the same size and zero padding is not required. Thus, the **LR2LR** operation is less costly than the full-rank update in this context.

Sparse Multifrontal BLR solver A BLR multifrontal sparse direct solver was designed for the MUMPS solver. The different strategies are summarized in [88] and a theoretical study of the complexity of the solver for regular meshes is presented in [10]. In the current implementation, the fronts are always assembled in a full-rank form before being compressed. The authors suggest as an alternative solution to assemble the fronts panel by panel to avoid allocating the full front in a dense fashion, but this was not implemented yet and it may impact the potential parallelism. MUMPS is a multifrontal solver, and BLR is considered when eliminating a dense front and not between fronts while our supernodal approach has a more global view of all supernodes.

Our scenario **Just-In-Time** is similar to the FCSU (Factor, Compress, Solve, Update) strategy from [88] where the fronts are compressed panel by panel during their factorization and where only full-rank blocks are updated (**LR2GE** kernel). The LUAR (Low-Rank Update Accumulation with Recompression) groups together

multiple low-rank products to exploit the memory locality during the product recompression process. This could be similarly used in the **Just-In-Time** strategy, but would imply larger ranks in the extend-add operations of the **Minimal Memory** strategy.

The CUFS (Compress, Update, Factor, Solve) [88] strategy is the closest to the **Minimal Memory** scenario, as it performs low-rank assembly (**LR2LR** kernel) within a front. However, the fronts are still first allocated in full-rank to assemble the contributions from the children. Within a front, the CUFS strategy is similar to what was proposed by LSTC for dense matrices. In that case, since blocks are dense and low-rank operations are performed between blocks of equal sizes, zero padding is not necessary as in the **Minimal Memory** strategy.

We believe that there is more room for memory savings using the **Minimal Memory** strategy for two reasons: 1) we avoid the extra cost of fronts inherent to the multifrontal method and 2) the low-rank assembly avoids forming dense blocks before compressing them. We demonstrated that the **Minimal Memory** strategy reduces the number of operations performed on 3D Laplacians and other matrices coming from various applications, which is the main contribution of this chapter. A theoretical complexity study has to be performed to investigate if such an approach performs asymptotically the same number of operations as **Just-In-Time** or MUMPS strategies with a constant factor overhead. Low-rank assembly has only been studied in [112] for 2D problems with the knowledge of the underlying problem, or using randomization techniques such as those presented in Section 1.2.2.3. Computing the complexity bounds for a more general case is still an open research problem, we are currently investigating.

Extension to hierarchical formats With the aim of extending our solver to hierarchical compression schemes, such as $\mathcal{H}$, HSS, or HODLR, we consider graphs coming from real-life simulations of 3D physical problems. From a theoretical point of view, the majority of these graphs have a bounded degree or are bounded density graphs [90], and thus good separators [81] can be built. For an n-vertices mesh, time complexity of a direct solver is in $\Theta(n^2)$, and we expect to build a low-rank solver requiring $\Theta(n^{\frac{4}{3}})$ operations. For memory requirements, the direct approach leads to an overall storage of $\Theta(n^{\frac{4}{3}})$, while we target a $\Theta(n\log(n))$ complexity.

Let us consider the last separator of size $\Theta(n^{\frac{2}{3}})$ for a 3D mesh, and one of the largest low-rank blocks of this separator in a hierarchical clustering. They have asymptotically the same size. Previous studies have shown that such a block may have a rank of order $\Theta(n^{\frac{1}{3}})$. For the **Just-In-Time** scenario, maintaining such a block in a dense form before compressing it requires $\Theta(n^{\frac{4}{3}})$ memory. Thus, we will still have the same memory peak, but we might encounter a large overhead when compressing off-diagonal blocks with current RRQR and SVD kernels. For the **Minimal Memory** scenario, we have seen that the cost of the solver can be split into two stages. The low-level one from the elimination tree that generates small updates to large contribution blocks and that might become more expensive with the hierarchical compression, and the high-level one where blocks fit the hierarchical structure and generate flops savings.

To overcome the issue of the low-level contributions, new ordering techniques need to be investigated. This is the context of Chapter 5 and Chapter 6. The objective is to exhibit suitable low-rank structures while minimizing the number of updates on larger off-diagonal blocks. Another approach, out-of-scope of this thesis, would be the use of randomization techniques to accumulate more easily low-rank updates. Otherwise, as we presented in Section 1.2.2.3, the control of the error is not straightforward.

3.5 Discussion

In this chapter, we presented a new Block Low-Rank sparse solver that introduces low-rank compression kernels in a supernodal solver. This solver reduces the memory consumption and/or the time-to-solution depending on the scenario. Two scenarios were developed. For the set of real-life problems studied, **Minimal Memory** saves memory up to a factor of 4 using RRQR kernels, with a time overhead that is limited to 2.8. Large problems that could not fit into memory when the original solver was used can now be solved thanks to the lower memory requirements, especially when low accuracy solutions. For larger problems, one can expect that the reduction of the number of operations will translate into a time-to-solution reduction. We experienced this behavior with large Laplacians: we are now able to solve a 330^3 unknown Laplacian while the original solver was limited to a 200^3 unknown Laplacian, the time-to-solution being reduced over 150^3 unknowns.

Just-In-Time reduces both the time-to-solution by a factor up to 3.5 and the memory requirements of the final factorized matrix with similar factors to **Minimal Memory**. However, with the current scheduling strategy, this gain is not reflected in the memory peak.

Two compression kernels, SVD and RRQR, were studied and compared. We have shown that, for a given tolerance, both approaches provide correct solutions with the expected accuracy, and that RRQR, despite larger ranks, provides faster kernels. In addition, we demonstrated that the solver can be used either as a low-tolerance direct solver or as a good preconditioner for iterative methods, that normally require only a few iterations before reaching the machine precision. A comparison with other preconditioners (AMG, ILU(k)) will be performed in future work to measure the impact of using a low-rank factorization as a preconditioner.

In the future, new kernel families, such as RRQR with randomization techniques, will be studied in terms of accuracy and stability in the context of a supernodal solver. To further improve the performance of **Minimal Memory** and close up the gap with the original solver, aggregation techniques on small contributions will also be studied.

Regarding **Just-In-Time**, future work is focused on studying smart scheduling strategies that combine **Right-Looking** and **Left-Looking** approaches in order to find a good compromise between memory and parallelism for the target architecture. Sergent et al. [105] studied scheduling with memory constraints for dense factorization using BLR, and we expect it can be adapted for sparse direct solvers. This will follow up recent work on applying parallel runtime systems [74] to the PaStiX

solver.

A challenging future work will consist of extending this contribution for distributed architectures. The volume of data communications should be reduced thanks to low-rank structure. However, aggregating small updates together in order to control the number of communications is not straightforward, especially in a supernodal approach together with low-rank updates.

Chapter 4

Reordering strategy to reduce the number of off-diagonal blocks

The preprocessing steps of sparse direct solvers, ordering and block-symbolic factorization, are two major steps that lead to a reduced amount of computation and memory and to a better task granularity to reach a good level of performance when using BLAS kernels. With the advent of GPUs, the granularity of the block computation became more important than ever. In this chapter, we present a reordering strategy that increases this block granularity. This strategy relies on the block-symbolic factorization to refine the ordering produced by tools such as Metis or Scotch, but it does not impact the number of operations required to solve the problem. In Section 4.1, we illustrate this problem on a simple case, before presenting our heuristic in Section 4.2. We present the impact of using this heuristic in Section 4.3, with experiments on a large set of matrices. In Section 4.4, we discuss the impact of this work as well as some future works.

4.1 Intra-node reordering

Let us illustrate the problem of current ordering solutions and how to overcome this problem. For this purpose, we consider a regular 3D cube of n^3 vertices presented in Figure 4.1. We apply the nested dissection process to this cube. Naturally, the first separator, in gray, is a plane of n^2 vertices cutting the cube into two halves' of balanced parts. Then, by recursively applying the nested dissection process, we partition the two-halves' subparts with the two red separators, and again dissect the resulting partitions by the four third-level green separators, giving us eight final partitions. We know from this process that each separator will be ordered with higher indices than those at lower levels.

Inside each separator, vertices have to be ordered as well, and it is common to use techniques such as the Reverse Cuthill-McKee [45] (RCM) algorithm in order to have an internal separator ordering "as contiguous as possible" to limit the number of off-diagonal blocks in the associated column block. This strategy works with only the local graph induced by the separator. It starts from a peripheral vertex and orders, consecutively, vertices at distance 1, then at distance 2, and so on, giving indices in

reverse order. It is close to a Breadth-First Search (BFS) algorithm. However, such an algorithm uses only interactions within a supernode, without taking into account contributing supernodes. On the quotient graph of Figure 1.2, this means that this will reorder unknowns inside a node of this graph without considering interactions with other nodes of this graph. However, these interactions are those related to off-diagonal blocks in the factorized matrix. Therefore, it is important to note that the ordering inside a supernode can be rearranged to take into account interactions with vertices outside its local graph without changing the final fill-in of the L block structure used by the solver. Then we can expect that complete knowledge of the local graph and of its outer interactions will lead to better quality ordering in terms of the number of off-diagonal blocks.

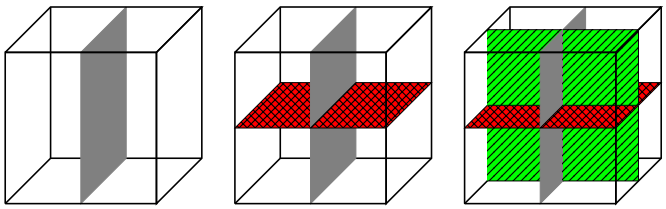
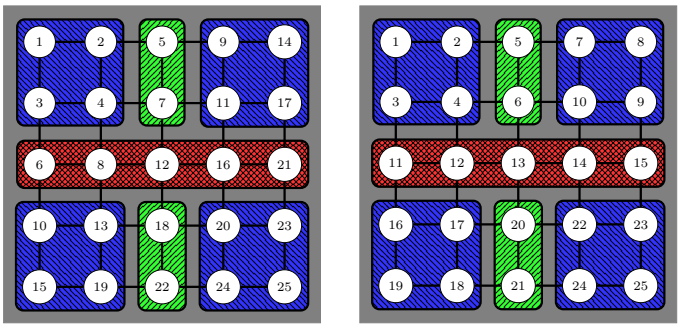


Figure 4.1: Three levels of nested dissection on a regular cube.

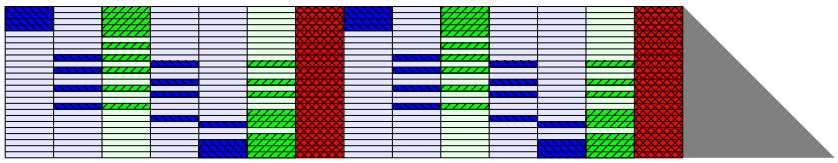


(a) With RCM ordering. (b) With optimal ordering.

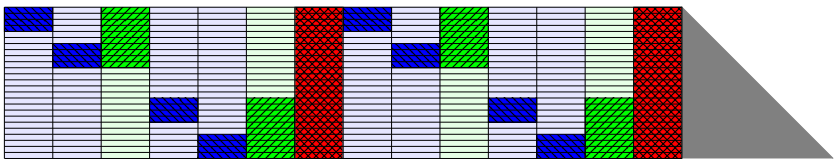
Figure 4.2: Projection of contributing supernodes and ordering on the first separator (gray in Figure 4.1).

Figure 4.2 presents the vertices of the gray separator from the 3D cube case with $n = 5$. The projection of contributing supernodes on this separator is shown. The blue parts are the vertices connected only to the leaves of the elimination tree. Thanks to the nested dissection process, the nodes of the gray separator have the largest numbers, and their connections to other supernodes represent the off-diagonal contributions. Based on this, we propose an optimal ordering (see Figure 4.2(b)), computed by hand, as opposed to an RCM algorithm (see Figure 4.2(a)). This ordering is considered optimal as it minimizes the number of off-diagonal blocks to one per column. One can note that RCM will not order consecutively vertices that will receive contributions from the same supernodes, leading to a substantially larger

number of off-diagonal blocks than the optimal solution. For instance, the four blue vertices in the top right of the RCM ordering will create four different off-diagonal blocks. The general idea is that some projections will be cut by RCM following the neighborhood, while those vertices could have been ordered together to reduce the number of off-diagonal blocks. On the right, the optimal ordering tries to consider this rule by ordering vertices with similar connections in a contiguous manner. This leads to a smaller number of off-diagonal blocks, as shown in the block-data structure computed by block-symbolic factorization for these two orderings in Figures 4.3(a) and 4.3(b). The ordering proposed in Figure 4.2(b) is optimal in terms of number of off-diagonal blocks since it is impossible to exhibit a block-symbolic structure with less than 14 off-diagonal blocks: there is no more than one off-diagonal block per column block.



(a) With RCM ordering.



(b) With optimal ordering.

Figure 4.3: Off-diagonal blocks contributing to the first separator in Figure 4.1. With the RCM strategy, there are 46 off-diagonal blocks and only 14 with the optimal ordering strategy.

We have demonstrated with this simple example that RCM does not fulfill the correct objective in a more global view of the problem. This is especially true in the context of 3D graphs, where the separator is a 2D structure, receiving contributions from 3D structures on both sides. With 2D graphs, the separator is a 1D structure and in such a case RCM will generally provide a good solution by following the neighborhood in the BFS algorithm. However, it often happens that the separators found by generic tools such as Metis or Scotch are disconnected graphs, making this previous statement incorrect as it is impossible to recover and follow the spatial neighborhood of the vertices (issued from the associated mesh).

Note that if it is quite easy to manually compute the optimal ordering on our example, it is harder in practice. Indeed, given an initial partition $V = A \cup B \cup C$, nothing guarantees that subparts A and B will be partitioned in a similar fashion, and that the resulting projection will match. For instance, Figure 4.4 presents the projection of level-1 (in red) and level-2 (in green) supernodes on the first separator of a $40 \times 40 \times 40$ Laplacian partitioned with Scotch. One can note that there are crossed contributions, meaning that subparts A and B are partitioned differently.

In the next section, we propose a new reordering strategy that permutes the

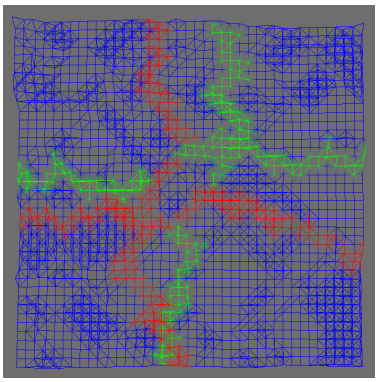


Figure 4.4: Projection of contributing supernodes on the first separator of a 3D Laplacian of size $40 \times 40 \times 40$, using Scotch.

rows to compact the off-diagonal information. Note that such a reordering strategy will not impact the global fill-in since the diagonal blocks are considered as dense blocks. The first solution, shown in Figure 4.4, would be to cluster vertices by common connections to nodes of the quotient graph. However, in most cases, that would result in clusters of $O(1)$ size that would still need to be ordered correctly, taking into account their level in the elimination tree of the connected supernodes. The solution we propose to remedy this problem relies on the computed block-data structure. Our objective is to express an algorithm providing the optimal solution before proposing a heuristic with reasonable complexity.

4.2 Improving the blocking size

As presented in Section 4.1, the RCM algorithm, widely used to order supernodes, generates many extra off-diagonal blocks by not considering supernode interactions, which leads to an increased number of less efficient block operations. In this section, we present an algorithm that reorders supernodes using a global view of the nested dissection partition. We expect that considering contributing supernodes will lead to a better quality — a smaller number of larger blocks. Our main idea is to consider the set of contributions for each row of a supernode, before using a distance metric to minimize the creation of off-diagonal blocks when permuting rows.

4.2.1 Problem modeling

The strategy is to rely on the block-symbolic factorization of L instead of the original graph of A. Indeed, it allows us to take into account fill-in elements that were computed in the block-symbolic factorization process instead of recomputing those elements with the matrix graph. Let us consider the ℓth diagonal block C_ℓ of the factorized matrix that corresponds to a supernode, and the set of supernodes C_k with $k < \ell$ corresponding to the supernodes at levels of the elimination tree lower than C_ℓ. Note that we refer to N as the total number of diagonal blocks appearing in the structure of the factorized matrix, as opposed to n for the total number of

unknowns.

We define for each supernode C_ℓ

$$row_{ik}^{\ell} = \begin{cases} 1 \text{ if vertex } i \text{ from } C_\ell \text{ is connected to } C_k \\ 0 \text{ otherwise} \end{cases}, k \in [\![1, \ell-1]\!], i \in [\![1, |C_\ell|]\!]. \tag{4.1}$$

row_{ik}^{ℓ} is then equal to 1 when the vertex i, or row i, of the supernode C_ℓ is connected to any vertex of the supernode k belonging to a lower level in the elimination tree. Otherwise, it is equal to 0, meaning that no non-zero element connects the two in the initial matrix, or no fill-in will create that connection. Let us now define for each vertex the binary vector $B_i^\ell = (row_{ik}^\ell)_{k \in [\![1,\ell-1]\!]}$. We can then define w_i^ℓ, the weight of a row i, as in Equation (4.2), which represents the number of supernodes contributing to that row i, and the distance between two rows i and j, $d_{i,j}^\ell$, as in Equation (4.3). This is known as the Hamming distance [58] between two binary vectors and allows for measuring the number of off-diagonal blocks induced by the succession of two rows i and j. Indeed, $d_{i,j}^\ell$ represents the number of off-diagonal blocks that belongs to only one of the two rows, which can be seen as the number of blocks that end at row i or start at row j.

$$w_i^\ell = \sum_{k=1}^{\ell-1} row_{ik}^\ell, \tag{4.2}$$

$$d_{i,j}^\ell = d(B_i^\ell, B_j^\ell) = \sum_{k=1}^{\ell-1} row_{ik}^\ell \oplus row_{jk}^\ell, \tag{4.3}$$

where $\oplus$ is the exclusive `or` operation.

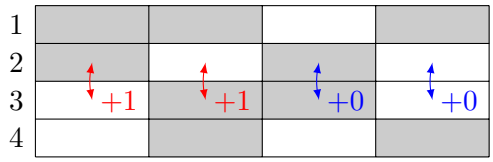
Thus, the total number of off-diagonal blocks, odb^ℓ, contributing to the diagonal block C_ℓ can be defined as:

$$odb^\ell = \frac{1}{2}(w_1^\ell + \sum_{i=1}^{|C_\ell|-1} d_{i,i+1}^\ell + w_{|C_\ell|}^\ell), \tag{4.4}$$

where the Hamming weights of the first and last rows of the supernode C_ℓ correspond, respectively, to the number of blocks in the first row and in the last one, and the distances between two consecutive rows give the evolution in the number of blocks when traveling through them.

Figure 4.5 illustrates the computation of the number of off-diagonal blocks with Equation (4.4) on an example of four rows. The computation of the distance from the second to the third row is illustrated on the left: there are two differences, making it a distance of 2. Figure 4.5(b) summarizes the distances between each couple of rows in this example. With this information and the weight of the first and last rows, respectively, 3 and 2, one can compute the number of off-diagonal blocks, odb, from the formula: $\frac{1}{2}(w_1 + d_{1,2} + d_{2,3} + d_{3,4} + w_4) = \frac{1}{2}(3+3+2+2+2) = 6$.

Thus, to reduce the total number of off-diagonal blocks in the final structure, the goal is to minimize this metric odb^ℓ for each supernode by computing a minimal path



(a) Symbol structure.

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

(b) Distance matrix.

Figure 4.5: Example of a symbolic structure and its associated distance matrix. Computation of the distance between rows 2 and 3 is illustrated on the left with the difference for each block.

visiting each node, with a constraint on the first and the last node. This problem is known as the Shortest Hamiltonian Path Problem, and is NP-hard.

4.2.2 Proposed heuristic

We first propose to introduce an extra virtual vertex, S_0, for which B_0 is the null set. Thus, we have:

$$\forall i \in [\![1, |C_\ell|]\!],\ d^\ell_{0,i} = d^\ell_{i,0} = w_i, \tag{4.5}$$

The problem can now be transformed to a Traveling Salesman Problem [17] (TSP):

$$\sum_{i=0}^{|C_\ell|} d^\ell_{i,(i+1)}, \tag{4.6}$$

which is also an NP-Hard problem, but for which many heuristics have been proposed in the literature [69], contrary to the Shortest Hamiltonian Path Problem. Furthermore, our problem presents properties that make it suitable for better heuristics and theoretical models that bound the maximum distance to the optimal solution. First, our problem is symmetric since:

$$d^\ell_{ij} = d^\ell_{ji}, \forall (i,j) \in [\![1, |C_\ell|]\!]^2, \tag{4.7}$$

and second, it respects the triangular inequality:

$$d^\ell_{ij} \leq d^\ell_{ik} + d^\ell_{kj}, \forall (i,j,k) \in [\![1, |C_\ell|]\!]^3. \tag{4.8}$$

This means our problem is an Euclidean TSP, and so heuristics for this specific case can be used. Different TSP heuristics that can be used to solve this problem, with their respective cost and quality with respect to the optimal, are presented in Table 4.1.

To keep a global complexity below that of the numerical factorization, we explain in Section 4.2.3 that the complexity of the TSP algorithm has to remain equal to or lower than $\Theta(p^2)$, where p is the number of vertices in the cycle. Thus, it prevents advanced algorithms such as the Christofides algorithm [31] from being used. Furthermore, as p might reach several hundred or more, the use of the nearest neighbor

or Clarke and Wright heuristics might provide low quality results. From the remaining options, we decided to use the nearest insertion method which is a quadratic algorithm and guarantees a maximal distance to the optimal of at most 2 [101]. A quality comparison of our algorithm over 3D Laplacian matrices and real matrices against the CONCORDE [91] TSP solver that returns optimal solutions has shown that our nearest insertion algorithm provides results within less than 10% from the optimal.

Algorithm	Complexity for p nodes	Quality (wrt optimal)
Nearest neighbor	$\Theta(p^2)$	$\frac{1}{2}(1 + log(p))$
Nearest insertion	$\Theta(p^2)$	2
Clarke and Wright	$\Theta(p^2\ log(p))$	$\Theta(log(p))$
Cheapest insertion	$\Theta(p^2\ log(p))$	2
Minimum spanning tree	$\Theta(p^2)$	2
Christofides	$\Theta(p^3)$	1.5

Table 4.1: Complexity and quality of different TSP algorithms.

Our final algorithm is then decomposed in three stages presented in Algorithm 5 that are applied to each separator of the nested dissection. Note that it is not applied on the leaves of the elimination tree since they will not receive contributions from other supernodes. The first step is to compute the B_i^ℓ vectors for each row i of the current separator. Then it computes the distance matrix of the separator: $D_\ell = (d_{i,j})_{(i,j)\in[\![0,|C_\ell|]\!]^2}$. Finally, the TSP algorithm is executed using this matrix to produce the local ordering of the supernode that minimizes Equation (4.6).

Algorithm 5 Reordering algorithm

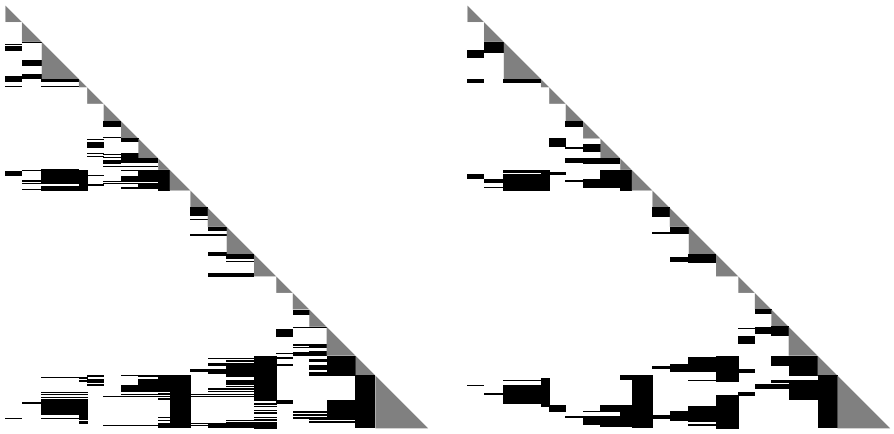
```
For each supernode C_ℓ in the elimination tree Do
  For each row i in the supernode C_ℓ Do
    For each contributing node k ∈ [[1, ℓ − 1]] Do
      Set row_ik^ℓ to 1                          ▷ Build the structure B_i^ℓ
    End For
  End For
  For each row i in the supernode C_ℓ Do
    For each row j in the supernode C_ℓ Do
      Compute the distance between rows i and j  ▷ Compute the distances
    End For
  End For
  Cycle^ℓ = {S_0, 1}
  For i ∈ [[2, |C_ℓ|]] Do
    Insert row i in Cycle^ℓ such that (4.6) is minimized   ▷ Order rows
  End For
  Split Cycle^ℓ at S_0
End For
```

The first stage of this algorithm builds the vector B_i^ℓ for each row i. In fact,

to minimize the storage, only contributing supernodes ($row_{ik}^{\ell} = 1$) are stored for B_i^{ℓ}. In order to do so, we rely on the structure of the block-symbolic factorization, which provides a compressed storage of the information similar to the compressed sparse row (CSR) format. Given a supernode, one can easily access the off-diagonal blocks contributing to this supernode, and due to the sparse property, the number of these blocks is much smaller than $\ell - 1$. The accumulated operations for all the supernodes in the matrix are in $\Theta(n)$. Note that we store the contributing supernode numbers in an ordered fashion for faster computation of the distances. Furthermore, the memory overhead of this operation is limited by the fact that each supernode is treated independently.

The second stage computes the distance matrix. When computing the distance d_{ij}^{ℓ} between rows i and j from C_{ℓ}, we take advantage of the sorted sets B_i^{ℓ} and B_j^{ℓ} to realize this computation in $\Theta(|B_i^{\ell}| + |B_j^{\ell}|)$ operations.

The third stage executes the nearest insertion heuristics to solve the TSP problems on the vertices of the supernode based on the previously computed distance matrix. As stated previously, this step is computed in $\Theta(|C_{\ell}|^2)$ operations. It is known that the solution given is not optimal but will be at a distance 2 of the optimal in the worst case.



(a) Without reordering strategy. (b) With reordering strategy.

Figure 4.6: Block-symbolic factorization of $8 \times 8 \times 8$ Laplacian. On the left, it represents the structure obtained with the initial Scotch ordering and on the right the structure obtained after reordering.

Figure 4.6 presents the block-symbolic factorization of a 3D Laplacian of size $8 \times 8 \times 8$ reordered with the Scotch nested dissection algorithm. In Figure 4.6(a), our reordering algorithm has not been applied, and supernode ordering comes only from the local RCM applied by Scotch. One can notice that some rows can be easily aggregated to reduce the number of off-diagonal blocks. In Figure 4.6(b), our algorithm has to reorder unknowns within each supernode. The final structure exhibits more compact blocks that are larger. Note that the fill-in of the matrix has not changed due to the dense storage of the diagonal blocks. Our algorithm does not impact the fill-in outside those diagonal blocks.

4.2.3 Complexity study

For this study we consider graphs issued from finite element meshes coming from real-life simulations of 2D or 3D physical problems. From a theoretical point of view, the majority of those graphs have a bounded degree and are specific cases of bounded-density graphs [90]. In this section, we provide a complexity study of our reordering algorithm in the context of a nested dissection partitioning strategy for this class of graphs.

Good separators can be built for bounded-density graphs or, more generally, for overlap graphs [89]. In d-dimensions, such n-node graphs have separators whose size grows as $\Theta(n^{(d-1)/d})$. In this study, we consider the general framework of separator theorems introduced by Lipton and Tarjan [81] for which we will have $\sigma = \frac{d-1}{d}$.

Definition 4.1. A class φ of graphs satisfies an n^σ-separator theorem, $\frac{1}{2} \leq \sigma < 1$, if there are constants $\frac{1}{2} \leq \alpha < 1, \beta > 0$ for which any n-vertex graph in φ has the following property: the vertices of G can be partitioned into three sets A, B, and C such that:

- no vertex in A is adjacent to any vertex in B;
- $|A| \leq \alpha n$, $|B| \leq \alpha n$; and
- $|C| \leq \beta n^\sigma$, where C is the separator of G.

Theorem 4.1. *(From [29]) The number of off-diagonal rows in the block-data structure for the factorized matrix L is at most $\Theta(n)$.*

This result comes from [29]. In that paper, the authors demonstrated that the number of off-diagonal blocks is at most $\Theta(n)$ and this was achieved by proving that this upper bound is in fact true for the total number of rows inside the off-diagonal blocks, leading to Theorem 4.1. Using this theorem, we demonstrate Theorem 4.2.

Theorem 4.2. *(Reordering Complexity) For a graph of bounded degree satisfying an n^σ-separation theorem, the reordering algorithm complexity is bounded by $\Theta(n^{\sigma+1})$.*

Proof

The main cost of the reordering algorithm is issued from the distance matrix computation. As presented in Section 4.2.2, we compute a distance matrix for each supernode. This matrix is of size $|C_\ell|$, and each element of the matrix, D_ℓ, is the distance between two rows of the supernode. The overall complexity is then given by:

$$\mathcal{C} = \sum_{\ell=1}^{N} \sum_{i=1}^{|C_\ell|} (row_{ik}^{\ell})_{k \in [\![1,\ell-1]\!]} \times (|C_\ell| - 1). \tag{4.9}$$

More precisely, for a supernode C_ℓ, the complexity is given by the number of off-diagonal rows that contribute to it multiplied by the number of comparisons: $(|C_\ell| - 1)$. For instance, given Figure 4.3(a), one can note that the complexity will be proportional to the colored surface (blue, green, and red blocks), where $row_{ik}^{\ell} = 1$, as well as in the number of rows. Using the compressed sparse information (colored blocks) only – instead of the dense matrix – is important for reaching a reasonable theoretical complexity, since this number of off-diagonal blocks is bounded in the context of finite element graphs.

Given Theorem 4.1, we know that the number of off-diagonal contributing rows in the complete matrix L is in $\Theta(n)$. In addition, the largest separator is asymptotically smaller than the maximum size of the first separator, that is, $\Theta(n^\sigma)$. The complexity is then bounded by:

$$\mathcal{C} \leq \underbrace{\max_{1\leq\ell\leq N}(|C_\ell| - 1)}_{\Theta(n^\sigma)} \times \underbrace{\sum_{\ell=1}^{N}(\sum_{i=1}^{|C_\ell|}(row_{ik}^{\ell})_{k\in[\![1,\ell-1]\!]})}_{\Theta(n)} = \Theta(n^{\sigma+1}).$$

For graphs of bounded degree, this result leads to the following:

- For the graph family admitting an $n^{\frac{1}{2}}$-separation theorem (2D meshes), the reordering cost is bounded by $\Theta(n\sqrt{n})$ and is – at worst – as costly as the numerical factorization.
- For the graph family admitting an $n^{\frac{2}{3}}$-separation theorem (3D meshes), the reordering cost is bounded by $\Theta(n^{\frac{5}{3}})$ and is cheaper than the numerical factorization, which grows as $\Theta(n^2)$.

Analysis

Note that this complexity is, as said before, larger than the complexity of the TSP nearest insertion heuristic. For a subgraph of size p respecting the p^σ-separation theorem, this heuristic complexity is in $\Theta(p^{2\sigma})$. Using [29], we can compute the overall complexity as a recursive function depending on the complexity on one supernode. This leads to an overall complexity in $\Theta(n\ log(n))$ for 2D graphs and $\Theta(n^{\frac{4}{3}})$ for 3D graphs and is then less expensive than the complexity of computing the distance matrix.

The reordering is as costly as the numerical factorization for 2D meshes, but RCM usually gives a good ordering on 2D graphs, since the separators are contiguous lines. For the 3D cases, the reordering strategy is cheaper than the numerical factorization. Thus, this reordering strategy is interesting for any graph with $\frac{1}{2} < \sigma < 1$, including graphs with a structure between 2D and 3D meshes. This algorithm can easily be parallelized since each supernode is an independent subproblem, and the distance matrix computation can also be computed in parallel. Thus, the cost of this reordering step can be lowered and should be negligible compared to the numerical factorization.

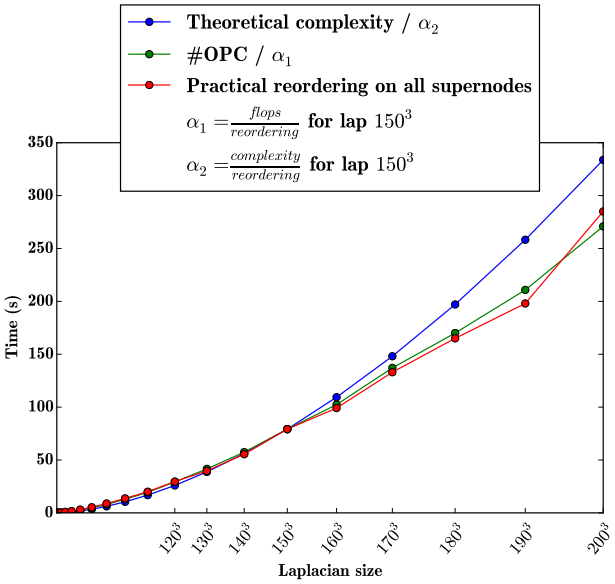


Figure 4.7: Comparison of the time of reordering against theoretical and practical complexities on 3D Laplacians.

Figure 4.7 presents the complexity study on 3D Laplacian matrices. We computed the practical complexity of our reordering algorithm with respect to the upper bound we demonstrated. The red curve represents the sequential time taken by our reordering algorithm. It is compared to the theoretical complexity demonstrated previously, but scaled to match on the middle point (size 150^3) to ease readability, so we can confirm that the trends of both curves are identical to a constant factor. Finally, in green we also plotted the practical complexity: total number of comparisons performed during our reordering algorithm, to see if the theoretical complexity was of the same order. This curve is also scaled to match on the middle point. One can note that the three curves are quite close, which confirms that we found a good upper bound complexity for a large set of sizes.

Note that this complexity seems to be significant with respect to the factorization complexity. Nevertheless, the nature of operations (simple comparisons between integers) is much cheaper than the numerical factorization operations. In addition, if we use the partitioner to obtain large enough supernodes, it will reduce by a notable factor the complexity of our algorithm, as we operate on a column block and not on each element contributing to each row. This parameter can be set in ordering tools such as Metis and Scotch and has an impact on the global fill-in of the matrix. As presented before, the reordering stage takes part of the preprocessing steps and can be used for many numerical steps, and it enhances both factorization and solve steps.

4.2.4 Strategies to reduce computational cost

As we have seen, the total complexity of the reordering step can still reach the complexity of the numerical factorization. We now introduce heuristics to reduce the computational cost of our reordering algorithm.

Multi-Level: partial computation based on the elimination tree

As presented in Section 1.1, the obtained partition allows us to decompose contributing supernodes according to the elimination tree. With the characterization theorem, we know that when we consider one row, if this row receives a contribution from a supernode in the lowest levels, then it will receive contributions from all its descendants to this node. This helps us divide our distance computation into a two dimension distance $d_{i,j} = d_{i,j}^{\mathbf{high}} + d_{i,j}^{\mathbf{low}}$ to reduce its cost. Given a $split_{level}$ parameter, we first compute the **high-level** distance, $d_{i,j}^{\mathbf{high}}$, by considering only the contributions from the supernodes in the $split_{level}$ levels directly below the studied supernode. This distance gives us a minimum of the distance between two rows. Indeed, if considering all supernodes, the distance will be necessarily equal to or larger than the **high-level** distance by construction of the elimination tree. Then we compute the **low-level** distance, $d_{i,j}^{\mathbf{low}}$, only if the first one is equal to 0.

In practice, we observed that for a graph of bounded degree, not especially regular, a ratio of 3 to 5 between the number of lower and upper supernodes largely reduces the number of complete distances computed while conserving a good quality in the results. The $split_{level}$ parameter is then adjusted to match this ratio according to the part of the elimination tree considered. It is important to notice that it is impossible to consider the distances level by level, since the goal here is to group together the rows which are connected to the same set of leaves in the elimination tree. This means that they will receive contributions from nodes on identical paths in this tree. The partial distances consider only the beginning of those paths and not their potential *reconnection* further down the tree. That is why it is important to take multiple levels at once to keep a good quality.

Stopping criteria: partial computation based on distances

The second idea we used to reduce the cost of our reordering techniques is to stop the computation of a distance if it exceeds a threshold parameter. This solution helps to quickly disregard the rows that are "far away" from each other. This limits the complexity of the distance computation, reducing the overall practical complexity. In most cases, a small value such as 10 can already provide good quality improvement. However, it depends on the graph properties and the average number of differences between rows. Unfortunately, if this heuristic is used alone, this improvement is not always guaranteed and it might lead to a degradation in quality. In association with the previous multi-level heuristic, the results are always improved, as we will see in the following section.

4.3 Experimental study

In this section, we present experiments with our reordering strategy, both in terms of quality (number of off-diagonal blocks) and impact on the performance of numerical factorization. We compare here three different strategies to the original ordering provided by the Scotch library. Two are based on our strategy, namely TSP, with the full distance computation or with the multi-level approximation. The third, namely HSL, is the one implemented in the HSL library for the MA87 supernodal solver (`optimize_locality` routine from MA87). Note that those three reordering strategies are applied to the partition found by Scotch, and they do not modify the fill-in. The parameters used in the solver, the platform on which experiments were performed and the matrices studied are presented in Appendix A.1.

4.3.1 Reordering quality and time

First, we study the quality and the computational time of the three reordering algorithms, the two versions of our TSP and HSL, compared to the original ordering computed by Scotch that is known to be in $\Theta(n\ log(n))$. Note that sequential implementation is used for all algorithms, except in Section 4.3.1.

For the quality criteria, the metric we use is the number of off-diagonal blocks in the matrix. We always use Scotch to provide the initial partition and ordering of the matrix, thus the number of off-diagonal blocks only reflects the impact of the reordering strategy. Another related metric we could use is the number of off-diagonal blocks per column block. In ideal cases, it would be, respectively, 4 and 6 for 2D and 3D meshes. However, since the partition computed by scotch is not based on the geometry, this optimum is never reached and varies a lot from one matrix to another, so we stayed with the global number of off-diagonal blocks and its evolution compared to the original solution given by Scotch.

Quality

Figure 4.8 presents the quality of reordering strategies in terms of the number of off-diagonal blocks with respect to the Scotch ordering. We recall that Scotch uses RCM to order unknowns within each supernode. Three metrics are represented: one with HSL reordering, one for our multi-level heuristic, and finally one for the full distance computation heuristic. We can see that our algorithm reduces the number of off-diagonal blocks in all test cases. In the 3D problems, our reordering strategy improves the metric by 50-60%, while in the 2D problems, the improvement is of 20-30%. Furthermore, we can observe that the multi-level heuristic does not significantly impact the quality of the ordering. It only reduces it by a few percent in 11 cases over the 104 matrices tested, while giving the same quality in all other cases. The HSL heuristic improves the initial ordering on average by approximately 20%, and up to 40%, but is outperformed by our TSP heuristic regardless of the multi-level distances approximation on all cases.

In addition, we observed that for matrices not issued from meshes with an unbalanced elimination tree, and not presented here, using the multi-level heuristic can deteriorate the solution. Indeed, in this case, the multi-level heuristic is unable to

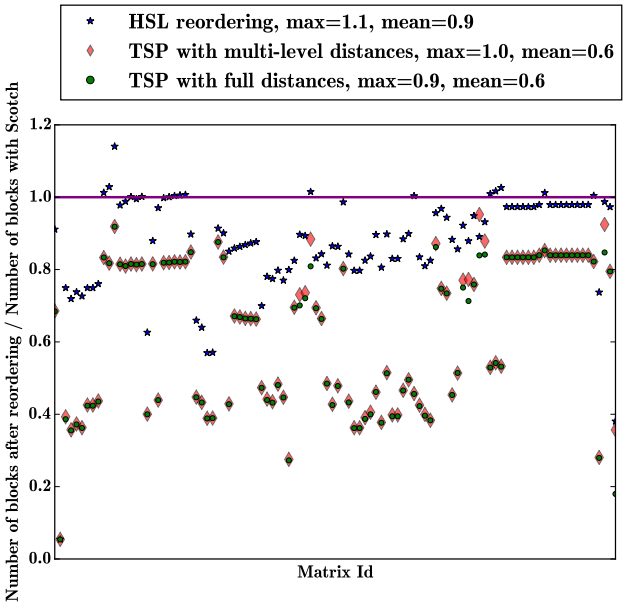


Figure 4.8: Impact of the heuristic used on the ratio of off-diagonal blocks over those produced by the initial SCOTCH ordering on a set of 104 matrices from The SuiteSparse Matrix Collection. The lower the better.

distinguish close interactions from far interactions with only the first levels, leading to incorrect choices.

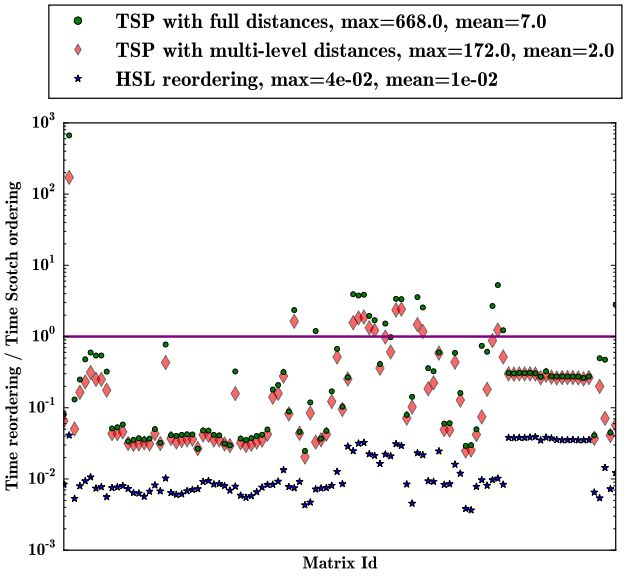


Figure 4.9: Time of the sequential reordering step with respect to the initial SCOTCH ordering on a set of 104 matrices from The SuiteSparse Matrix Collection. The lower the better.

Time

Figure 4.9 presents the cost of reordering strategies used in sequential with respect to the cost of the initial ordering performed by Scotch. The reordering is in fact an extra step in the preprocessing stage of sparse direct solvers. One can note that despite the higher theoretical complexity, the reordering step of our TSP heuristic is 2 to 10 times faster than Scotch. Thus, adding the reordering step creates an overhead of no more than 10-50% in most cases when used sequentially. However, on specific matrix structures, with a lot of connections to the last supernode, the reordering operation can be twice as expensive as Scotch. In those cases, the overhead is largely diminished by the multi-level heuristic, which reduces the time of the reordering step to the same order as Scotch. We observe that the multi-level heuristic is always beneficial to the computational time. For the second matrix – an optimization problem with a huge density on the left of the figures – we can observe a quality gain of more than 95%, while the cost is more than 600 times larger than the ordering time. This problem illustrates the limitation of our heuristic using a global view compared to the local heuristic of the HSL algorithm which is still faster than the Scotch ordering but gives only 20% improvement. This problem is typically not suited for sparse linear solvers, due to its large number of non-zeroes as well as its consequent fill-in.

Strategy	Number of blocks		Time (s)	
	Full	Multi-level	Full	Multi-level
Scotch	9760700		360	
Reordering / Stop= 10	4100616	4095986	33.2	**31.1**
Reordering / Stop= 20	3896248	3897179	42.6	38.5
Reordering / Stop= 30	**3891210**	3891262	50.7	43.3
Reordering / Stop= 40	3891803	3891962	58.1	46.3
Reordering / Stop= ∞	3891825	3892522	64.8	47.7

Table 4.2: Number of off-diagonal blocks and reordering times on a CEA matrix with 10 million unknowns. The first line represents statistics obtained using only Scotch and the next five lines correspond to the use of the reordering strategy after Scotch ordering.

Stopping criteria

Table 4.2 shows the impact of the stopping criteria on a large test case issued from a 10 million unknowns matrix from the CEA. The first line presents the results without reordering and the time of the Scotch step. We compared this to the number of off-diagonal blocks and the time obtained with our reordering algorithm when using different heuristics. The `STOP` parameter refers to the criteria introduced in Section 4.2.4 and defines after how many differences a distance computation must be stopped. One can notice that with all configurations the quality is within 39-42% of the original, which means that those heuristics have a low impact on the

quality of the result. However, this can have a large impact on the time to solution, since a small `STOP` criterion combined with the multi-level heuristic can divide the computational time or the reordering by more than 2.

In conclusion, we can say that for a large set of sparse matrices, we obtain a resulting number of off-diagonal blocks between two and three times smaller than the original Scotch RCM ordering, while the HSL heuristic reduces them on average only by a fifth. It is interesting as it should reduce by the same factor the overhead associated with the tasks management in runtimes, and should improve the kernel efficiency of the solver. In our experiments, we reach a practical complexity close to the Scotch ordering process, leading to a preprocessing stage that is not too costly compared to the numerical factorization. Furthermore, it should accelerate the numerical factorization and solve steps to hide this extra cost when only one numerical step is made, and give some global improvement when multiple factorizations or solves are performed. While HSL reordering overhead might be much smaller than our heuristic, we hope that the difference in the quality gain, as well as the fact that our strategy improves all children instead of giving advantage to the largest one, will benefit the factorization step by a larger factor.

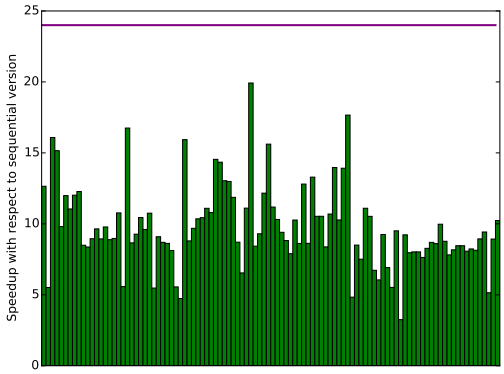


Figure 4.10: Speedup of the reordering step (full distance heuristic) with 24 threads on a set of 104 matrices from The SuiteSparse Matrix Collection.

Parallelism

As previously stated, the reordering algorithm is largely parallel as each supernode can be reordered independently of the others. The first level of parallelism is a dynamic bin-packing that distributes supernodes in reverse order of their sizes. However, some supernodes are too large and take too long to be reordered compared to all others. They represent almost all the computational requirements. We then divided the set of supernodes into two parts. For the smaller set, we just reorder different supernodes in parallel, and for the larger set, we parallelize the distance matrix computation. Figure 4.10 shows the speedup obtained with 24 threads over the best sequential version on a `miriel` node. This simple parallelization accelerates the algorithm by 10 on average and helps to totally hide the cost of the reordering step in a multi-threaded context, where ordering tools are hard to parallelize. Note

that for many matrices, the parallel implementation of our reordering strategy has an execution time smaller than 1s. In a few cases, the speedup is still limited to 5 because the TSP problem on the largest supernode remains sequential and may represent a large part of the sequential execution.

4.3.2 Impact on supernodal method: PaStiX

In this section, we measure the performance gain brought by the reordering strategies. For these experiments, we extracted 6 matrices from the previous collection, and we use the number of operations (Flops) that a scalar algorithm would require to factorize the matrix with the ordering returned by Scotch to compute the performance of our solver. We recall that this number is stable with all reordering heuristics.

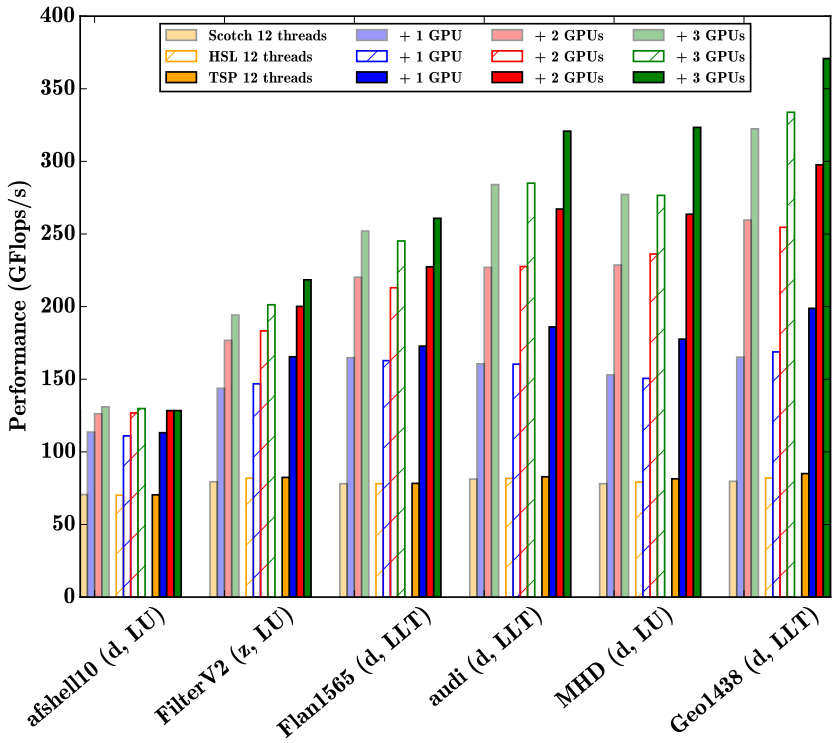


Figure 4.11: Performance impact of the reordering algorithms on the PaStiX solver on top of the Parsec runtime with 1 node of the hybrid mirage architecture.

Figure 4.11 presents the performance on a single `mirage` node, for three algorithms based on the original ordering from Scotch. The first one leaves the Scotch ordering untouched. The HSL heuristic is applied on the second one, and finally the third one includes the TSP heuristic with full distances. For each matrix and each ordering, scalability of the numerical factorization is presented with all 12 cores of the architecture enhanced by 0 to 3 GPUs. All results are an average performance over five runs.

To explain the different performance gains, we rely on Table 4.3, which presents the average number of rows in the off-diagonal blocks with and without reordering, and not the total number of blocks to give some insight into the size of the updates. The block width is defined by the SCOTCH partition and is the same for all experiments on each matrix. This number is important, as it is especially beneficial to enlarge blocks when the original solution provides small data blocks.

Matrix	Avg. number of rows of off-diag. block			TSP times	
	SCOTCH	HSL	TSP	Overhead	Gain
afshell10	46.05	45.52	54.09	0.133 s	0 s
FilterV2	8.794	11.33	19.91	0.164 s	1.23 s
Flan1565	29.13	32.33	62.40	0.644 s	0.52 s
audi	17.94	20.57	41.76	0.748 s	2.08 s
MHD	16.86	17.04	27.64	1.16 s	4.42 s
Geo1438	18.79	23.17	49.74	1.78 s	7.48 s

Table 4.3: Impact of the reordering strategies on the number of rows per off-diagonal block and on the timings. The timings show the overhead of the parallel TSP and the gain it provides on the factorization step using the mirage architecture.

For multi-threaded runs, both reordering strategies give a slight benefit up to 7% on the performance. Indeed, on the selected matrices, the original off-diagonal block height is already large enough to get a good CPU efficiency since the original solver already runs at up to 67% of the theoretical peak of the node (128.16 GFlop/s). This is also true for HSL reordering. In general, when the solver exploits GPUs, the benefit is more important and can reach up to 20%.

In Figure 3.5, we can see that with the **afshell10** matrix, extracted from a 2D application, reordering strategies have a low impact on the performance, and the accelerators are also not helpful for this lower computation case. For the **Flan1565** matrix, the gain is not important for both reordering strategies because the original off-diagonal block height is already large enough for efficiency. On other problems, issued from 3D applications, we observe significant gains from 10-20% which reflect the block height increase of 1.5 to 2.5 presented in Table 4.3.

If we compare this with the HSL reordering strategy, we can see that our reordering is helpful in slightly improving the performance of the solver. Hence, choosing between both strategies depends on the number of factorizations that are performed.

Table 4.3 also presents our TSP reordering strategy overhead with the parallel implementation and the resulting gain on the numerical factorization time using the `mirage` architecture with the three GPUs. Those numbers reflect that it is interesting to use our reordering strategy in many cases with a small overhead that is immediately recovered by the performance improvement of the numerical factorization. Since several problems presenting the same structure are solved, this small overhead is again diminished. Similarly, if GPUs are used, the gain during the factorization is higher, completely hiding the overhead of the reordering. The cost of the HSL strategy being really small with respect to SCOTCH, it is always recommended to apply it for a single factorization or for homogeneous computations. However, if GPUs are

involved, HSL reordering impact on the performance of the numerical factorization is really slight and it goes from a slight slowdown to a slight speedup (around 3%). This validates the use of more complex heuristics than the proposed TSP.

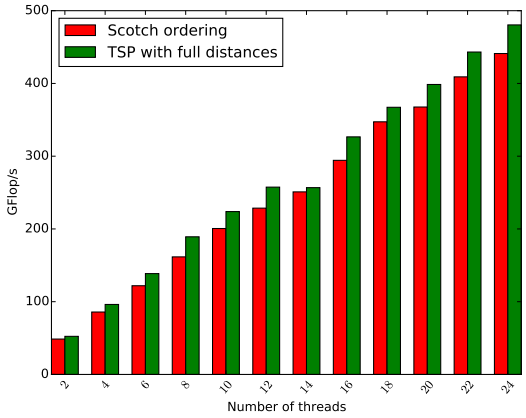


Figure 4.12: Scalability on the CEA 10 million unknowns matrix with 24 threads.

Figure 4.12 presents a scalability study on one `miriel` node with 24 threads, with and without our reordering stage on the 10 million unknowns matrix from the CEA. This matrix, despite being a large 3D problem, presents a really small average block size of less than 5 when no reordering is applied. The reordering algorithm rises up to 12.5, explaining the larger average gain of 8–10% that is observed. In both cases, we notice that the solver manages to scale correctly over the 24 threads, and even a little better when the reordering is applied. A slight drop in the performance on 14 threads is explained by the overflow on the second socket.

4.4 Discussion

This, we presented a new reordering strategy that reduces the number of off-diagonal blocks in the block-symbolic factorization. It allows one to significantly improve the performance of GPU kernels, and the BLAS CPU kernels in smaller ratios, as well as reducing the number of tasks when using a runtime system. The resulting gain can be up to 20% on heterogeneous architectures enhanced by NVIDIA Fermi architectures. Such an improvement is significant, since it is difficult to reach a good level of performance with sparse algebra on accelerators. This gain can be observed on both the factorization and the solve steps. It works particularly well for graphs issued from finite element meshes of 3D problems. In the context of 2D graphs, partitioner tools can be sufficient, as separators are close to 1D structures and can easily be ordered by following the neighborhood. For other problems, the strategy enhances the number of off-diagonal blocks, but might be costly on graphs where vertices have large degrees.

Furthermore, we proposed a parallel implementation of our reordering strategy, leading to a computational cost that is really low with respect to the numerical factorization and that is counterbalanced by the gain on the factorization. In addition,

if multiple factorizations are applied on the same structure, this benefits the multiple factorization and solve steps at no extra cost. We proved that such a preprocessing stage is cheap in the context of 3D graphs of bounded degree and showed that it works well for a large set of matrices. We compared it with HSL reordering, which targets the same objective of reducing the overall number of off-diagonal blocks. While the TSP heuristic is often more expensive, the quality is always improved, leading to better performance. In the context of multiple factorizations, or when using GPUs, the TSP overhead is recovered by performance improvement, while it may be better to use HSL for the other cases.

For future work, we plan to study the impact of our reordering strategy in a multifrontal context with the MUMPS [9] solver and compare it with the solution studied in [106], which performs the permutation during the factorization. The main difference with the static ordering heuristics studied in this thesis is that the MUMPS heuristic is applied dynamically at each level of the elimination tree. Such a reordering technique is also important in the objective of integrating variable-size batched operations currently under development for the modern GPU architectures. Finally, one of the most important perspectives is to exploit this result to guide matrix compression methods in diagonal blocks for using hierarchical matrices in sparse direct solvers. Indeed, considering the diagonal block by itself for compression without external contributions leads to incorrect compression schemes. Using the reordering algorithms to guide the compression helps to gather contributions corresponding to similar far or close interactions.

Jacquelin et al. [67] presented a reordering strategy using partition refinement, developed after the TSP heuristic proposed in this chapter. They demonstrate that, if their solution degrades slightly the quality in terms of number of off-diagonal blocks, its computational cost is less expensive than the strategy we developed. Both approaches can take advantage of parallelism by reordering independently each separator. Otherwise, there may be more room for parallelism when reordering a single separator. A hybrid approach would be to combine both methods, to exhibit a coarse ordering of columns using the strategy presented in [67], before applying our strategy on smaller set of unknowns.

Chapter 5

Block Low-Rank clustering

The low-rank clustering consists into splitting unknowns of a separator among clusters. More precisely, as presented in Section 1.2.1.2, its objective is to form clusters that are well separated such that most of the interactions are low-rank. For the dense case, this clustering has to maximize compressibility, while in the sparse case it also impacts the granularity of the data structures, which makes the clustering of sparse matrices a challenging problem.

As mentioned in Section 4.1, one cannot classify vertices of a separator among set receiving exactly the same contributions. It would result in clusters of size $O(1)$ by considering all interactions in the elimination tree. Indeed, for a graph split into $A \cup B \cup C$ with C the separator, subparts A and B are ordered independently, which increases the number of possible sets of contributions a vertex can receive. For instance, let us consider only one level of children (two children), and the corresponding projections $A_A \cup B_A \cup C_A$ (respectively $A_B \cup B_B \cup C_B$) for the subpart A (respectively B). As, by definition of the nested dissection process, C is connected to both A and B subparts, vertices belonging to the separator can be classified into nine different parts (combination of each kind of vertex for each subpart). For a large number of children, this number grows quickly, so it is impossible to classify unknowns among clusters with this approach.

In Chapter 2, we described the problem of low-rank clustering in a simple example in Figure 2.1. When using algebraic partitioning tools such as Scotch, separators interactions are even more irregular, as it was presented in Figure 4.4. More precisely, let us consider the block matrix:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \tag{5.1}$$

where the set of unknowns belonging to the last separator corresponds to A_{22} and the remaining unknowns to A_{11}. The blocks A_{21} and A_{12} correspond to the interaction between A_{11} and A_{22}.

Given this representation, operations are divided into: 1) **POTRF**(A_{11}) to factorize A_{11}, 2) **TRSM**(A_{11},A_{21}) to solve the off-diagonal blocks A_{21} and A_{12}, 3) **HERK**(A_{21},A_{22}) to perform the updates that will contribute to A_{22} and 4) **POTRF**(A_{22}) to factorize A_{22}. The objective of a good clustering strategy is to

reduce the cost of factorizing a separator, **i.e.**, **POTRF**(A_{22}), but also to exhibit an efficient coupling to reduce the cost of **HERK**(A_{21},A_{22}).

The issue with existing clustering strategies is that they do not consider both intra (A_{22}) and inter (A_{12} and A_{21}) separators properties. For instance, k-way partitioning takes into account only intra-separator properties in A_{22}, but does not consider A_{21}. On the opposite, the reordering strategy presented in Chapter 4 orders correctly the coupling parts A_{12} and A_{21} without exhibiting a suitable low-rank structure for A_{22}.

In this chapter, we study the impact of clustering techniques on the PaStiX solver and propose a new heuristic to couple assets of existing methods. In Section 5.1, we describe the clustering operation and present assets and drawbacks of existing heuristics (k-way and reordering). In Section 5.2, we propose a new heuristic and evaluate its impact with respect to existing strategies in Section 5.3. Finally, we discuss limitations of the new heuristic and future works in Section 5.4.

5.1 Low-rank clustering problem

We recall that permuting vertices within a separator does not impact the fill-in since diagonal blocks are considered as dense blocks. Then, both the memory consumption and the number of operations are kept untouched for full-rank arithmetic, while it can impact low-ranl compressibility. Thus, the objective is to perform a clustering of unknowns that 1) enhances compression rates and 2) maintains efficient sparse structures, by permuting unknowns within a separator. We expect to couple strategies that were designed to obtain efficient sparse data structures with low-rank clustering strategies originally introduced for dense matrices. In a geometric context, the objective is to form as many large admissible blocks (according to some criterion given in Section 1.2.1.1) as possible, while in a fully algebraic context it is more challenging since the distances between points are unknown.

5.1.1 Problem of the low-rank clustering

Both hierarchical ($\mathcal{H}$, $\mathcal{H}^2$, HSS, HODLR) and flat (BLR) compression techniques require a suitable clustering of unknowns that achieves two conditions: 1) form compact clusters in the sense that unknowns belonging to a same cluster are close together in the graph and 2) ensure that a cluster has only a few neighbors, such that most clusters are well-separated with low-rank interaction.

Most sparse direct solvers using low-rank compression follow the multifrontal method, the only solvers—to the best of our knowledge—using the supernodal method are [28] in a geometric context with fixed ranks and our solver, presented in Chapter 3. In the multifrontal method, the commonly used approach is to consider the graph made of fully-summed variables of a front and to perform a partitioning of this graph to obtain the low-rank clustering. For hierarchical strategies, this partitioning is performed recursively while in the BLR case, where no hierarchy is required, a k-way partitioning is usually performed.

In the supernodal approach, one can use a similar method for unknowns of a separator, that correspond to fully summed variables in the multifrontal method. The main drawback of k-way partitioning is that it would consider only intra-separator

interactions (A_{22}) and not the contributions from the exterior of the separator. In a non-fully-structured approach, where updates are applied to full-rank blocks, it may be sufficient since there are fewer constraints on granularity of dense blocks addition. However, in a fully-structured approach, where low-rank updates are performed, one can expect that a clustering that avoids scattering updates among too many blocks will benefit the solver. For both **Minimal Memory** and **Just-In-Time** strategies presented in Chapter 3, reducing the number of low-rank blocks should enhance compression rates. For the **Minimal Memory** strategy, it will also reduce the burden on the **LR2LR** operation, which overall cost depends mostly on the number of elemental updates.

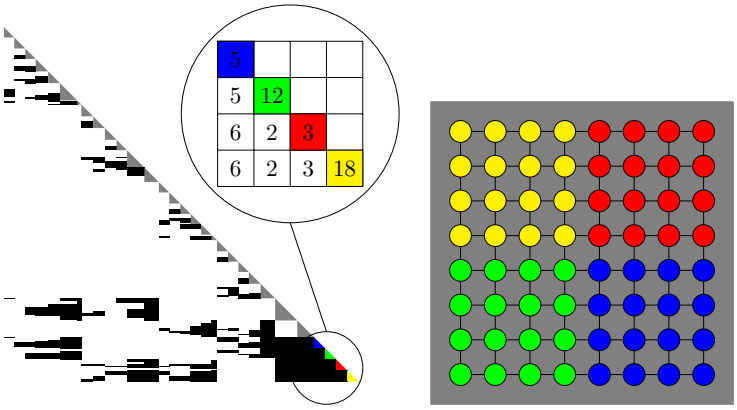
5.1.2 Example with advantages and drawbacks for k-way and reordering approaches

Both k-way and reordering approaches may not provide a suitable clustering of unknowns for supernodal solvers and when using low-rank assembly in general.

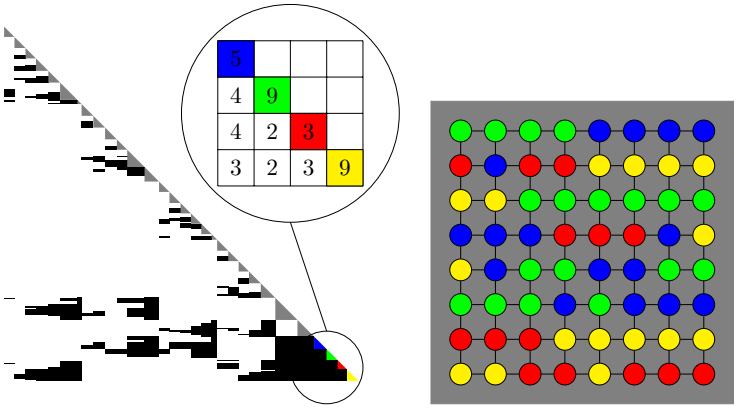
Let us illustrate the problem with a plane separator, by considering a 7-point stencil of size $8 \times 8 \times 8$ for which the first separator is a surface of size 8×8. In Figure 5.1(a), we present the block-symbolic factorization obtained by clustering unknowns of the last separator with a k-way partitioning into four parts. In the upper part of the figure, a zoom presents the number of external contributions received by each block. The clustering of the last separator is presented in the graph of the separator, where vertices belonging to a same cluster are marked with the same color.

In Figure 5.1(b), we present the block-symbolic factorization obtained by performing reordering on the last separator. From this block-symbolic structure, we perform the four parts clustering of the last separator with smart splitting [74], which gives parts of size 16, 18, 14 and 16. It avoids cutting too many off-diagonal blocks among different clusters, by computing an average block size and performing the actual split in an interval around the mean value to minimize the number of blocks affected by the cut. Similarly to the previous case, in the upper part of the figure, a zoom presents the number of external contributions received by each block after clustering. The figure also presents the graph of this last separator, where vertices belonging to a same part are marked with the same color.

One can note that both k-way and reordering are not optimal to obtain good data structures. The k-way clustering may provide good compression rates within the separator (A_{22}), however induces more off-diagonal updates. Furthermore, splitting the off-diagonal blocks in smaller contributions may make them incompressible. The reordering strategy reduces the number of off-diagonal blocks (A_{21} and A_{12}), as highlighted with a reduced number of contributions on last separator. However, vertices belonging to a same cluster are not close in the graph, which will degrade A_{22} compressibility.



(a) K-way partitioning.



(b) Reordering strategy.

Figure 5.1: Illustration of the clustering obtained through the k-way partitioning, on top, and the reordering heuristic, on bottom, for the top-level separator of size 8×8 of a $8 \times 8 \times 8$ regular grid. The symbolic factorizations, on the left, show the evolution of the off-diagonal blocks in number and size with a focus on the number of external contributions applied to each block of the matrix associated with the last separator (A_{22}). The meshes, on the right, show the distribution of the unknowns into the clusters on the graph of the separators.

5.2 Pre-selection heuristic

We propose here a new heuristic to perform the clustering of the unknowns in order to respect two conditions:

1. Minimize the rank of the interactions between clusters. This condition turns into maximizing the number of well-separated clusters, which can be performed by exhibiting clusters with a small diameter and only a few neighbors;

2. Minimize the number of contributions coming from children.

In addition, we expect to correctly identify interactions that are not well separated and that will lead to incompressible blocks to avoid performing needless low-rank compression.

In Figure 2.1, we defined the concept of traces, which correspond to the vertices of a separator that are directly connected to children which are close in the elimination tree. In Section 5.2.1, we present how traces are introduced to cluster vertices, before detailing the overall strategy in Section 5.2.2. Finally, we discuss some implementation detail in Section 5.2.3.

5.2.1 Using traces to pre-select and cluster vertices

The strategy to enhance supernodes clustering is to consider how children will contribute to a given separator. The objective is to order unknowns of a separator accordingly to the set of contributions it receives from the closest children in the elimination tree. Only closest children are considered, otherwise there are only few vertices receiving the same set of contributions. In addition, this strategy tries to isolate (pre-select) some vertices that represent strong connections, and that may not be compressible.

In practice, let us consider a separator and its closest children in the elimination tree. In order to cluster vertices of the separators depending on which contributions they receive, we consider traces of children on their ancestor. It was illustrated in Figure 2.1 for two levels of nested dissection, where green and red traces correspond to vertices of the separator that are directly connected to at least one children separator. In Figure 5.2, we present a separator with two red traces which correspond to interactions with direct children and four green traces that correspond to interactions with grand children in the elimination tree. From those traces, vertices belonging to a same connected subpart in the separator will receive the same set of contributions from the next two levels of children in the elimination tree. Naturally, the contributions coming from deeper children in the elimination tree will not be necessarily identical.

Vertices of a separator can be split into two categories: vertices belonging to one or more traces and vertices that are not connected to the closest children in the elimination tree. Vertices belonging to traces are named pre-selected vertices. Each connected subpart made of vertices that were not pre-selected forms a cluster, since those vertices will receive the same contributions (the sparsity pattern will be identical) from children whose traces were considered.

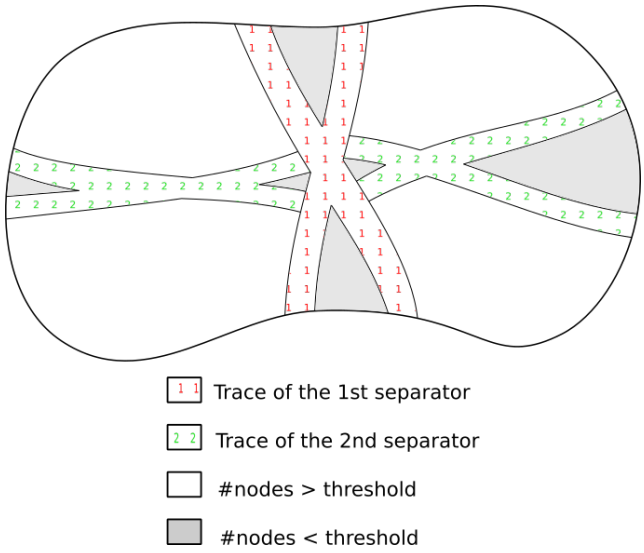


Figure 5.2: Two levels of traces on a generic separator.

Beyond the problem of forming suitable clusters, pre-selecting vertices intends to isolate some special vertices that represent strong interactions and thus are not compressible. For this reason, we do not try to compress intra-separator blocks corresponding to pre-selected vertices. More precisely, if the block A_{22} is split into A_{ss} for pre-selected vertices and A_{kk} for the rest of vertices, we obtain:

$$A_{22} = \begin{pmatrix} A_{kk} & A_{ks} \\ A_{sk} & A_{ss} \end{pmatrix} \tag{5.2}$$

From this representation, compressible blocks are only in A_{kk}, which corresponds to interaction between non-pre-selected vertices.

Some other blocks may be non compressible. For instance, off-diagonal blocks that just above or below the main diagonal include some vertices that are non compressible, so we do not compress those blocks. In addition, off-diagonal blocks that represent contributions between neighbors in the k-way partitioning include strong (distance-1) connections and may be non compressible, as presented in Section 1.2.1.1. In our implementation, we do not manage those blocks differently than others because it may degrade the overall compression rate.

5.2.2 Overall approach: compute pre-selected vertices and manage underlying subparts

The objective is to cluster unknowns to increase compressibility. In practice, we rely on traces to pre-select some vertices that will increase the distance between blocks, and thus the compressibility of those interactions. Traces are used to cluster vertices at a coarse level and k-way is used to refine those clusters to obtain suitable blocking sizes. The approach consists of computing pre-selected vertices before extracting distinct connected components in the set of vertices that do not belong to traces.

First of all, connected components that contain too few vertices to form a compressible cluster are merged together in order to form supernodes which size is larger than the minimum compressible size. The threshold used, as presented in Figure 5.2, is simply the minimum size used to compress a supernode. For larger connected components, the number of vertices can be too large to obtain reasonable clusters, which is necessary to reduce the size of dense diagonal blocks. For this reason, those subgraphs are clustered one-by-one using k-way partitioning.

To improve the projection process, the maximum number of pre-selected vertices must be controlled. For a separator of size n, and considering a constant number of children projections, the number of pre-selected vertices should not exceed $\Theta(\sqrt{n})$, which is the size of a separator of the separator being reordered and also the size of the traces of direct children. It ensures that only a few number of blocks are not compressed.

In Figure 5.3, we present the clustering of the last separator of a $80 \times 80 \times 80$ Laplacian matrix. Six traces are considered, two for the first level and four for the second level. From pre-selection, four large clusters were exhibited since we consider only two levels of nested dissection. Depending on their size, each large cluster was again split into five or six clusters using a k-way partitioning.

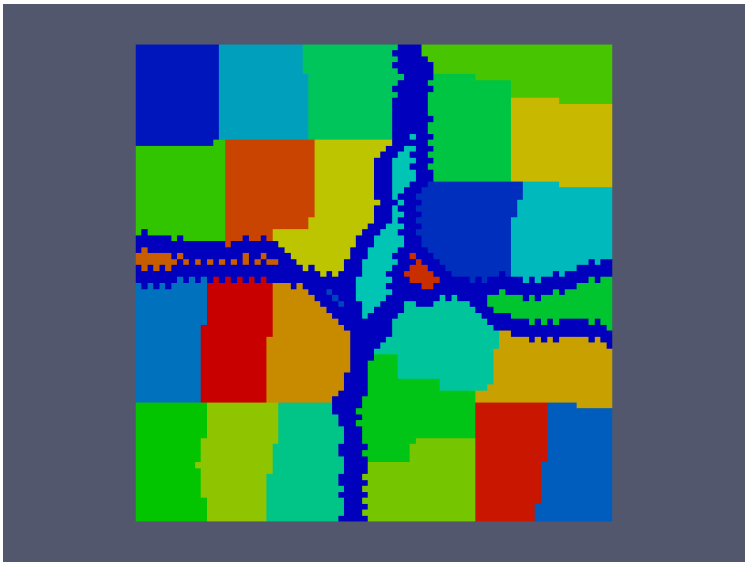


Figure 5.3: Two levels of trace for the last separator of a $80 \times 80 \times 80$ Laplacian matrix.

Using traces to cluster vertices seems to be straightforward for a regular 2D or 3D graph, as presented in Figure 5.3 for a relatively large case. However, such an approach is not appropriate for enhancing ordering of geometries where one dimension is larger than others. For instance, for a 2.5D graph where two dimensions are relatively large and the last one is much smaller, the first separators will be parallel plans. Thus, there is no connection between a separator and its direct children and obviously no vertex will be pre-selected. Such a pre-selecting algorithm is designed to enhance the clustering of graphs with a good aspect ratio.

5.2.3 Implementation details

The objective is to pre-process each separator before the block-symbolic factorization to form clusters and pre-select some non compressible vertices. In order to do so, each separator which is large enough is pre-processed with the heuristic relying on traces, and a k-way partitioning is performed if traces are not working, **i.e.**, if zero vertices were pre-selected.

First, separators computed by partitioning tools are not necessarily connected. Thus, reconnection of separators can be performed using paths of length 1 or 2 in the original graph. Then, we apply a clustering technique: either k-way, reordering, or the new heuristic introduced just before. Finally, for each cluster, the reordering is still performed to reduce the number of off-diagonal blocks contributing to the given cluster and thus the number of contributions. The reordering cost is even reduced, as the number of vertices considered in each block being reordered is much smaller and impacts the complexity by a quadratic factor (cf. Chapter 4). Thus, it is less expensive to perform reordering on clusters instead of performing reordering on the full separator.

5.2.3.1 Compute pre-selected vertices

We briefly described how traces are defined to obtain blue vertices in Figure 5.3. In practice, several parameters are considered to select vertices that are non-compressible and isolate suitable clusters.

The first parameter, named **levels of projections** (l), corresponds to the distance in the elimination tree for which children are considered. If this parameter is set to $l \geq 1$, the number of children considered will be 2^l (using nested dissection). This parameter has a large impact on the number of selected vertices, since increasing the number of children considered will increase the number of traces and by the same effect the number of distinct connected components. If too many children are considered, the connected components resulting from traces will be too small for being compressible. Note that we do not consider children that were not issued from the nested dissection process, for instance children that were obtained thanks to minimum-fill, but this type of ordering will only appear at the bottom of the elimination tree.

The second parameter, named **halo distance for projections** (d), corresponds to the distance from which a vertex from the separator being reordered and a vertex from children are considered as connected. In practice, for each vertex of the current separator, we are looking at his neighborhood at a distance d to see if the vertex has to be selected or not.

The third and last parameter, named **width of projections** (w), corresponds to the width of traces. After vertices of the separator have been selected thanks to **levels of projections** and **halo distance for projections** parameters, this third parameter will increase the width of bands to ensure a good separability.

Note that **halo distance for projections** and **width of projections** parameters are quite close in the sense that they both increase the width of selected vertices, but from a different point of view. Those parameters also increase the distances between clusters that were separated thanks to traces and then the compressibility of

interaction blocks between those clusters.

5.2.3.2 Control the number of pre-selected vertices

To control the number of pre-selected vertices, we introduced different parameters, depending on the blocking size b and the size of the separator n. First of all, a separator is clustered based on traces if its size is larger than $16b$. We expect to form four large connected components for a 2D separator of a 3D graph, as it would happen for a regular Laplacian with a constant number of traces. In addition, we want to form at least four k-way clusters in each connected component, which gives the 16 factor. K-way partitioning is always performed to obtain blocks of the maximum authorized blocking size, b.

Then, the number of children (and thus traces) considered is adapted level by level given a maximum limit. The objective is to select less than $\Theta(\sqrt{n})$ vertices such that the remaining number of vertices is larger than $4b$. Thus, we pre-select vertices level by level until reaching the maximum authorized number of pre-selected vertices and such that only the closest children in the elimination tree, given a maximum depth, are considered.

5.2.3.3 Graph algorithms

We now describe the different graph algorithms that are used to compute the clustering. Let us consider the graph of a separator $C = (V_C, E_C)$ made of V_C vertices and E_C edges for the complexity analysis.

OrderSupernode(C, l, d, w). This is the main routine (see Algorithm. 6) that orders unknowns of a separator C.

Algorithm 6 OrderSupernode(C, l, d, w): order unknowns within the separator C.

```
1:  ConnectSupernodeHalo( C )
2:  ComputeTraces( C, l, d, w )
3:  IsolateConnectedComponents( C )
4:  For each connected component C_i Do
5:      If |C_i| < threshold Then
6:          Merge C_i into small components vertices
7:      Else
8:          K-way( blocksize )
9:          For each k-way part K_j Do
10:             Reordering( K_j )
11:         End For
12:     End If
13: End For
14: Reordering(small components vertices)
15: Reordering(pre-selected vertices)
```

ConnectSupernodeHalo(C). This routine isolates the graph of the separator C being reordered. Connections through the original graph at distance 1 are turned into direct connections to obtain a connected graph and to better apply next partitioning algorithms. Reconnection at distance 2 was also used in MUMPS or STRUMPACK but it was shown in [110] that distance 1 is sufficient for most graphs.

In terms of complexity, this routine requires to explore, for each vertex of a separator, its neighborhood at distance 1. Given a bounded-degree graph where the larger degree of a vertex is $\Delta(C)$, the complexity of this routine is bounded by $\Theta(|V_C| \times \Delta(C))$.

ComputeTraces(C, l, d, w). This routine considers vertices of the separator C being reordered and search for direct connections with children from next l levels in the original graph. Connections that are issued from paths of length d can be computed with $\Theta(|V_C| \times \Delta(C)^d)$ operations. Finally, within the graph of the separator, some vertices at distance w from already pre-selected vertices are also marked as pre-selected.

IsolateConnectedComponents(C). This routine isolates each connected components of the separator C. It can be used either to isolate distinct components for a non-connected separator (for instance when partitioning a tore) or after traces have been computed to correctly identify each subpart.

This routine performs a Breadth-First Search (BFS) of the graph until each vertex has been visited once. When a BFS stops while all vertices have not been visited, a new connected component is created. This algorithm is linear in both the number of vertices and edges, its complexity is in $\Theta(|V_C| + |E_C|)$.

K-way(*blocksize*). K-way partitioning consists in partitioning a graph into a defined number of parts, such that each part has the same number of vertices and the number of edges (named cut) between parts is as low as possible. Note that k-way from METIS or SCOTCH try to minimize the overall edge-cut and not to balance edge-cut among different parts. For this routine, we directly call a SCOTCH strategy with an unbalance factor set to 5%.

The complexity of this routine used in a multilevel framework is in $\Theta(k|E_C|)$, where k is the number of parts in the k-way partitioning.

Reordering($part_i$). For each subpart, reorder vertices to enhance the sparsity pattern. A matrix of distances between the set of contributions for each unknown is computed and vertices are ordered using a traveling salesman algorithm. The algorithm and complexity study are presented in Chapter 4.

The complexity of the algorithm depends not only on the size and the connectivity of the separator graph, **i.e.**, average degree of nodes, but also on the parameters that are used. For instance, ComputeTraces(C, l, d, w) can be costly if d, **halo distance for projections** parameter, is too large. In Section 5.3.2.3, we study the

cost of clustering strategies and show that there is few or no overhead with the use of projections.

5.3 Experiments

In this section, we study the behavior of the different clustering strategies: k-way partitioning, named **K-way**, the reordering strategy together with smart splitting, named **Reordering**, and the newly introduced heuristic relying on traces, named **Projections**. In Section 5.3.1, we describe the parameters used in the solver to manage blocking size and control pre-selecting vertices. We study the behavior of the newly introduced heuristic with respect to **K-way** and **Reordering** strategies on a large set of matrices in Section 5.3.2. In Section 5.3.3, we detail results for a smaller set of matrices, to better describe the behavior of all heuristics.

5.3.1 Parameters and tuning of the solver

We use a large set of parameters to correctly tune our solver. All experiments are performed using 24 threads. Some parameters presented in Chapter 3 impact the solver by itself and not clustering strategies, studying their impact is out-of-scope of this section.

Blocking sizes and original ordering parameters are described in Section A.2. In order to obtain partitions with supernodes of similar width, the number of parts using k-way partitioning is defined to obtain clusters of size 256. The k-way partitioning method is the one from Scotch. As we will see in next experiments, the number of supernodes in the refined partition is almost invariant with the clustering method used.

Pre-selection is applied on separators which are large enough for being split. After pre-selecting vertices with traces, components of size lower than 128 are merged together, otherwise the corresponding blocks would be too small for being compressed. We use the newly introduced heuristic with $ComputeTraces(3, 1, 1)$, which provided in average the best results.

Finally, we switch to the **K-way** strategy if the number of pre-selected vertices is lower than $\alpha\sqrt{n}$, where α is set to 50, to correctly manage the number of pre-selected vertices as it was presented in Section 5.2.3.2.

5.3.2 Behavior on a large set of matrices

The objective of this section is to study the behavior of the three clustering techniques on a large set made of 33 matrices, presented in Table A.1.

In Figure 5.4 (respectively Figure 5.5), we present the performance profile for factorization (respectively for memory consumption) when using **K-way**, **Reordering** or **Projections** heuristics for both **Minimal Memory** and **Just-In-Time** factorizations using a 10^{-8} and a 10^{-12} tolerance. For each clustering heuristic, the percentage with respect to the optimal heuristic is computed (x axis) and accumulated for each matrix (y axis). It means that, on average, the best heuristics are

curves that remain close to $x = 1$. The objective of those figures is to give a general trend on a relatively large set of matrices.

5.3.2.1 Impact on factorization time

In Figure 5.4, one can observe that using the **K-way** strategy allows to reduce the factorization time with respect to the use of the **Reordering** strategy. When using either the **Minimal Memory** or the **Just-In-Time** strategy, **K-way** improves the factorization time by 10% for a 10^{-8} tolerance and 15% for a 10^{-12} tolerance.

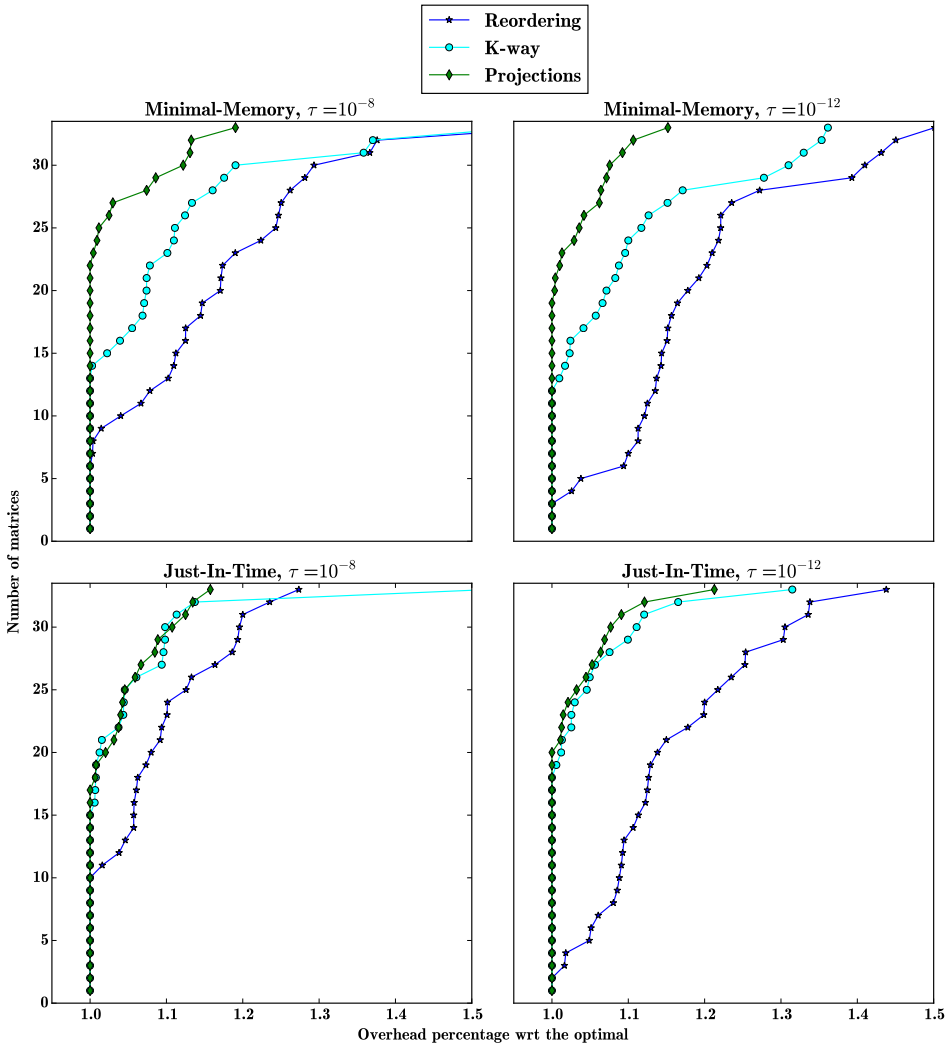


Figure 5.4: Performance profiles for the factorization time using three clustering strategies: **Reordering** (in blue star), **K-way** (in cyan circle), and **Projections** (in green diamond) on a set of 33 matrices for the **Minimal Memory** strategy on top, and the **Just-In-Time** strategy on bottom. On the left part, results with a 10^{-8} tolerance are presented and results with a 10^{-12} precision appear on the right.

Now, if we consider the new **Projections** heuristic, we observe different behavior

for both low-rank strategies. For the **Minimal Memory** strategy, the **Projections** heuristic allows to reduce the factorization time, as it was expected since it was designed to avoid updating blocks with a high rank. The gain is around 10% both for 10^{-8} and 10^{-12} tolerances. However, for the **Just-In-Time** strategy, there is almost no gain with respect to the **K-way** strategy. The burden on managing blocks with a high rank is less important, as it was shown in Figure 3.10(b). For both low-rank strategies, the **Projections** heuristic outperforms the **K-way** strategy with a larger factor for a 10^{-12} tolerance, since ranks are higher than using a 10^{-8} tolerance.

5.3.2.2 Impact on memory consumption

In Figure 5.5, we observe that the **K-way** strategy is the most suitable method for memory consumption. Using the **Reordering** strategy increases the memory consumption with a factor of 10%, while the **Projections** heuristic increases this metric by only a factor of 5%. The results favor the **K-way** strategy, especially for a more relaxed tolerance, such as 10^{-8}, as ranks are smaller.

Our new heuristic has only slight impact on memory consumption, while several blocks are not compressed and managed in a full-rank fashion all throughout the factorization.

5.3.2.3 Impact for preprocessing statistics

In Figure 5.6(a), we present the impact of clustering heuristics on the blocking sizes. We take the **Reordering** strategy as a reference, but it is only at a factor 2 from the optimal (cf. Table 4.1) and can eventually lead to a larger number of blocks than other clustering methods. We recall that for both **K-way** and **Projections** strategies, the same reordering strategy is still applied, but independently on each cluster of the separator and not the full separator. One can observe that both **K-way** and **Projections** strategies degrade the blocking sizes, since the number of off-diagonal blocks is larger. However, the average increase is of 5%, which should not impact much granularity. Note that for any clustering method, the number of supernodes in the refined partition is kept similar, by definition of our blocking sizes.

In Figure 5.6(b), we analyze the preprocessing cost of the three clustering strategies. The metric presented is the cost of clustering (including pre or post reordering) with respect to the cost of performing ordering using Scotch. All methods are performed in sequential. One can note that all methods have a similar behavior, with on average an extra cost of a factor 0.7 with respect to the ordering stage. If both **K-way** and **Projections** strategies require extra computations to compute clusters, this extra cost is masked by the reduction of the reordering cost. Indeed, using those methods, reordering is only performed within clusters, which reduces a lot its complexity. In addition, it can be easily parallelized, since each separator is pre-processed independently.

5.3.3 Detail analysis

In this section, we detail the behavior of the three clustering strategies. For the sake of simplicity, we will compare heuristics on the last separator only and rely on blocks

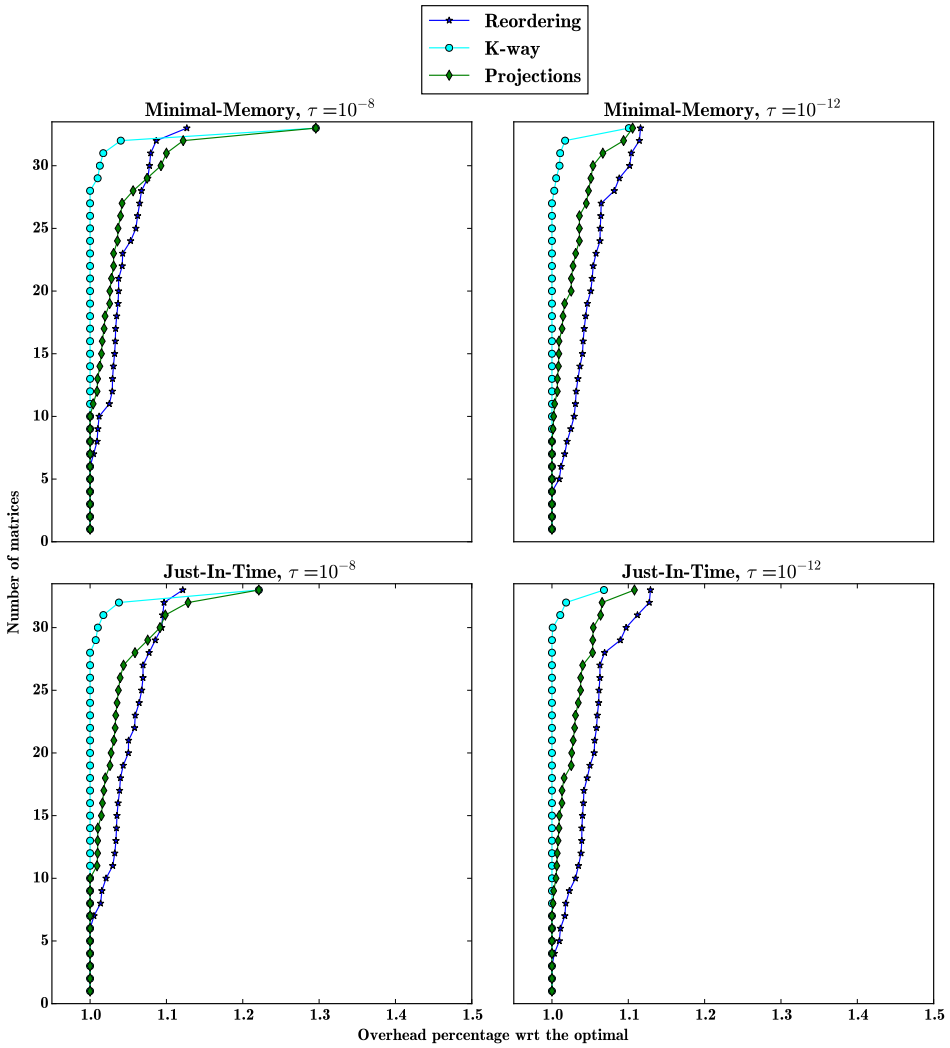
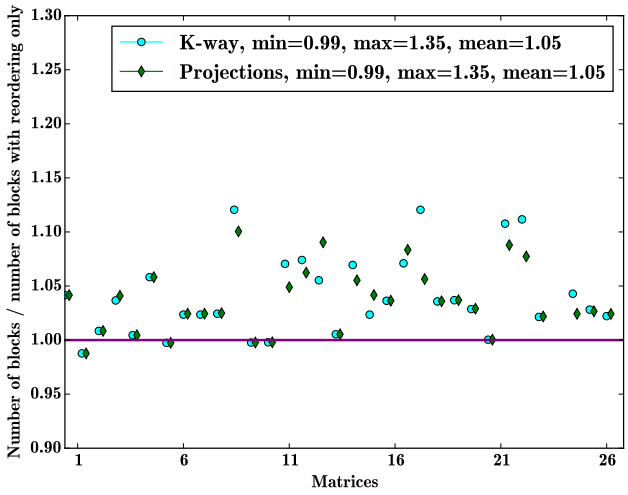
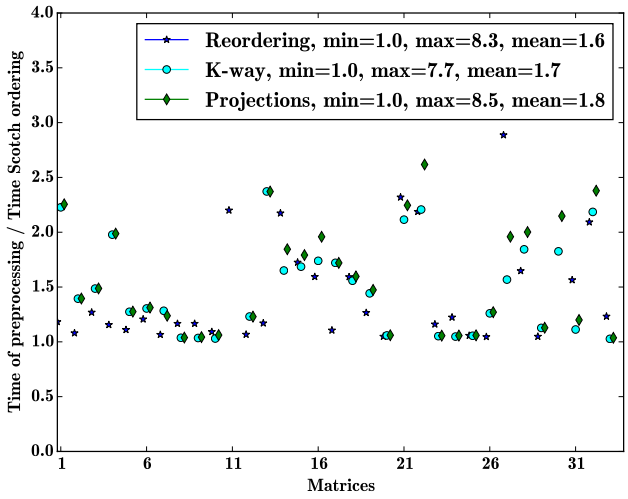


Figure 5.5: Performance profiles for the memory consumption time using three clustering strategies: **Reordering** (in blue star), **K-way** (in cyan circle), and **Projections** (in green diamond) on a set of 33 matrices for the **Minimal Memory** strategy on top, and the **Just-In-Time** strategy on bottom. On the left part, results with a 10^{-8} tolerance are presented and results with a 10^{-12} precision appear on the right.



(a) Impact on blocking sizes.



(b) Impact on preprocessing time.

Figure 5.6: Impact on preprocessing statistics for 33 matrices, for the number of blocks on top and for the preprocessing time on bottom. Three clustering strategies are studied: **Reordering** (in blue star), **K-way** (in cyan circle) and **Projections** (in green diamond). Those results are only structural and independent from the type of factorization or the tolerance used later.

introduced by Equation (5.1) to study the compression over different parts of the matrix. Note that to ease the reading we refer to A_{21} as the blocks belonging to both A_{12} and A_{21} in Equation (5.1). We start by discussing the behavior of both the **K-way** and the **Reordering** strategies before detailing results for the **Projections** strategy.

The intuition given in Section 5.1.2 is that, on the one hand, the **K-way** strategy will favor compression of A_{22} by correctly clustering vertices of the separators thanks to distances and diameters consideration. On the other hand, the **Reordering** strategy is supposed to be suitable for compression of A_{21} since it reduces the number of off-diagonal blocks.

In Table 5.1, we present the number of operations and the memory consumption for six representative matrices issued from various applications, as presented in Section A.2, with the full-rank, **Minimal Memory**, and **Just-In-Time** factorizations with a 10^{-8} tolerance. For low-rank strategies, we illustrate the memory consumption, the number of operations and the factorization time for the three considered clustering strategies. In this first analysis, we will only consider existing clustering methods and not the **Projections** heuristic.

The first observation is that, for all matrices, the **K-way** strategy leads to better factorization time with respect to the **Reordering** strategy. If we analyze how operations are split among different parts of the matrix, we observe that the **K-way** strategy does not only reduce the number of operations of A_{22}, but also the cost on the coupling part A_{21}, which is not intuitive. As k-way partitioning is only applied to the last separator to illustrate its behavior in a simpler case, there are only few differences for the number of operations corresponding to A_{11}. For this part, only the **TRSM**(A_{11}, A_{21}) kernel is impacted and it was shown in Chapter 3 that the corresponding operations do not represent a large percentage of the total number of operations.

This trend on the number of operations is reflected on the memory consumption, for which the **Reordering** strategy is worse than the **K-way** strategy not only for A_{22}, but also for A_{21} for all six matrices.

In order to better analyze the differences between the **K-way** and the **Reordering** strategies on the coupling part A_{21}, we introduce another metrics. In Table 5.2, we present, for the three clustering strategies, the distribution of off-diagonal blocks of A_{21} among three categories: 1) those which are compressed, 2) those which are compressible but have a high rank and thus are numerically incompressible and 3) those which are non-compressible as their height or their width is too small. This experiment was performed using a full-rank factorization followed by a compression of all blocks that are large enough using SVD with a 10^{-8} tolerance and without limiting the ranks to a maximum authorized rank. The objective is to illustrate the optimal compression rates attainable as well as the numerical properties of blocks which are incompressible (with a compression ratio lower than 1). We also integrate another metric, the number of compressed blocks that contain values from the original sparse matrix A, in order to evaluate how those values are split among blocks. Those interactions that already appear on the original matrix may be non-compressible.

The main observation, which confirms the asset of the **Reordering** strategy,

Matrix	Method	Strategy	HERK (A_{21}, A_{22}) (TFlops)	POTRF(A_{22}) (TFlops)	Total (TFlops)	Mem A_{11}	Mem A_{21}	Mem A_{22}	Fact(s)
lap120	Full-rank	-	3.71	0.95	14.44	8.66 GB	3.52 GB	805 MB	38.00
	Minimal-Memory	Reordering	0.21	0.04	2.63	3.77 GB	507 MB	151 MB	49.96
		K-way	0.16	0.02	2.62	3.77 GB	488 MB	123 MB	47.91
		Projections	0.07	0.25	2.78	3.77 GB	503 MB	445 MB	44.31
	Just-In-Time	Reordering	0.06	0.03	0.64	3.85 GB	527 MB	157 MB	13.77
		K-way	0.05	0.02	0.63	3.85 GB	504 MB	128 MB	13.58
		Projections	0.05	0.25	0.86	3.85 GB	521 MB	449 MB	13.95
atmosmodj	Full-rank	-	2.03	0.39	12.09	12.3 GB	3.42 GB	579 MB	35.69
	Minimal-Memory	Reordering	0.18	0.03	3.93	5.85 GB	651 MB	151 MB	69.72
		K-way	0.13	0.02	3.90	5.85 GB	599 MB	128 MB	65.03
		Projections	0.07	0.08	3.92	5.85 GB	658 MB	286 MB	62.64
	Just-In-Time	Reordering	0.04	0.02	0.83	5.99 GB	682 MB	159 MB	15.13
		K-way	0.04	0.02	0.80	5.99 GB	624 MB	133 MB	13.63
		Projections	0.04	0.08	0.88	5.99 GB	685 MB	289 MB	13.81
audi	Full-rank	-	0.32	0.02	5.23	8.62 GB	832 MB	52.6 MB	12.53
	Minimal-Memory	Reordering	0.08	0.01	3.82	5.98 GB	237 MB	31.6 MB	43.14
		K-way	0.05	0.00	3.78	5.98 GB	215 MB	25.4 MB	35.52
		Projections	0.05	0.00	3.78	5.98 GB	215 MB	25.4 MB	35.43
	Just-In-Time	Reordering	0.02	0.00	1.07	6.05 GB	243 MB	32.1 MB	9.17
		K-way	0.01	0.00	1.06	6.05 GB	221 MB	26.6 MB	7.97
		Projections	0.01	0.00	1.06	6.05 GB	221 MB	26.6 MB	7.95
Geo1438	Full-rank	-	3.23	0.70	18.40	15.7 GB	3.79 GB	658 MB	39.30
	Minimal-Memory	Reordering	0.78	0.18	9.22	10.6 GB	1.42 GB	266 MB	95.48
		K-way	0.68	0.14	9.00	10.6 GB	1.35 GB	242 MB	83.89
		Projections	0.40	0.32	8.97	10.6 GB	1.41 GB	460 MB	84.59
	Just-In-Time	Reordering	0.21	0.08	2.98	10.9 GB	1.49 GB	289 MB	22.69
		K-way	0.18	0.06	2.90	10.9 GB	1.41 GB	261 MB	22.41
		Projections	0.20	0.31	3.18	10.9 GB	1.48 GB	472 MB	22.66
Hook	Full-rank	-	0.94	0.10	8.82	10.8 GB	1.74 GB	183 MB	21.01
	Minimal-Memory	Reordering	0.17	0.02	4.75	6.95 GB	454 MB	74.4 MB	72.09
		K-way	0.12	0.01	4.68	6.95 GB	417 MB	61 MB	71.11
		Projections	0.07	0.04	4.66	6.95 GB	435 MB	118 MB	71.76
	Just-In-Time	Reordering	0.04	0.01	1.24	7.08 GB	475 MB	78.7 MB	14.01
		K-way	0.03	0.01	1.22	7.08 GB	436 MB	64.4 MB	13.55
		Projections	0.03	0.04	1.26	7.08 GB	455 MB	120 MB	13.33
Serena	Full-rank	-	5.67	1.18	29.04	15.8 GB	5.05 GB	944 MB	68.62
	Minimal-Memory	Reordering	0.73	0.20	8.97	9.4 GB	1.36 GB	311 MB	142.86
		K-way	0.66	0.15	8.86	9.4 GB	1.29 GB	271 MB	134.65
		Projections	0.44	0.36	8.82	9.4 GB	1.32 GB	532 MB	128.30
	Just-In-Time	Reordering	0.21	0.10	2.79	9.62 GB	1.43 GB	331 MB	32.18
		K-way	0.19	0.08	2.73	9.62 GB	1.35 GB	294 MB	31.52
		Projections	0.19	0.33	2.99	9.62 GB	1.38 GB	544 MB	30.59

Table 5.1: Number of operations and memory consumption for the factorization of six matrices with $\tau = 10^{-8}$ for the full-rank and both low-rank strategies. Three clustering strategies are studied: **Reordering**, **K-way** and **Projections**. To study the behavior on the last separator, we highlight three types of blocks: separator (A_{22}), its coupling (A_{21}) and the rest of the matrix (A_{11}), as presented in Equation (5.1).

Matrix	Strategy	Number of blocks				Compressible blocks containing values from A
		Total	Numerically compressible (ratio)	Numerically incompressible (ratio)	Non compressible, above threshold	
lap120	Reordering	46115	10137 (13.79)	0 (0)	35978	558
	K-way	51116	10809 (14.89)	1 (0.9)	40306	448
	Projections	50782	11259 (14.15)	2 (0.9)	39521	450
atmosmodj	Reordering	25653	4981 (10.33)	8 (0.9)	20664	306
	K-way	27486	4866 (12.11)	5 (0.9)	22615	244
	Projections	28678	5485 (10.88)	4 (0.9)	23189	234
audi	Reordering	4247	2187 (6.54)	74 (0.85)	1986	161
	K-way	4357	2048 (7.74)	59 (0.86)	2250	129
	Projections	4357	2048 (7.74)	59 (0.86)	2250	129
Geo1438	Reordering	17235	9955 (4.03)	314 (0.86)	6966	516
	K-way	18297	9958 (4.46)	292 (0.86)	8047	385
	Projections	18698	10454 (4.11)	331 (0.86)	7913	409
Hook	Reordering	11752	4617 (6.76)	56 (0.88)	7079	370
	K-way	12823	4835 (8.05)	35 (0.87)	7953	296
	Projections	13000	5185 (7.49)	46 (0.88)	7769	298
Serena	Reordering	25084	13250 (5.72)	206 (0.87)	11628	945
	K-way	27781	13537 (6.22)	180 (0.87)	14064	694
	Projections	27527	13731 (6.13)	178 (0.87)	13618	772

Table 5.2: Number of updates and compression rates for the coupling part A_{21} that represents interactions with the last separator. A full-rank factorization was performed and blocks were compressed afterwards using SVD with $\tau = 10^{-8}$ to illustrate the optimal compression rates attainable. Three clustering strategies are studied: **Reordering**, **K-way** and **Projections**. We distinguish compressible blocks, which sizes are large enough for compression and non compressible blocks. Among compressible blocks, numerically incompressible blocks are those whose ranks are too high for reducing memory consumption.

is that the total number of off-diagonal blocks is larger using the **K-way** strategy. However, the proportion of compressible blocks is quite similar and the compression rates obtained using the **K-way** strategy are better than the ones using the **Reordering** strategy. This trend can be linked with the number of compressible blocks that contain values from the original graph. As more blocks contain values from A with the **Reordering** strategy, it can explain the smaller compression rates. In practice, the **Reordering** strategy does not handle those blocks differently than others. However, as k-way partitioning clusters vertices that are close in the graph, it can order contiguously vertices that own edges connected with the same part of the original graph.

We now analyze low-level behavior of the **Projections** heuristic. The objective is to exhibit basic statistics about **how and when** compression rates are impacted.

In Table 5.2, one can see the **Projections** strategy has a behavior between the **K-way** and the **Reordering** strategies. Indeed, there are slightly less blocks, but more blocks that contain values from A. In addition, the compression rates for compressible blocks is slightly worse than the one of the **K-way** strategy but better than the one of the **Reordering** strategy.

In Table 5.1, we can observe the global behavior of the **Projections** strategy with respect to existing strategies. Firstly, one can note that for both **Minimal Memory** and **Just-In-Time** strategies, the **Projections** strategy allows reducing factorization time with respect to the use of **Reordering** strategy. Secondly, in terms of memory consumption, the consumption related to A_{22} is naturally increased since pre-selected blocks are managed in a full-rank fashion. For the coupling part A_{21}, memory consumption slightly increases with respect to the **K-way** strategy, but is better than the one of the **Reordering** strategy. Finally, the most relevant observation is the distribution of the number of operations. For both **Minimal Memory** and **Just-In-Time** strategies, the number of operations related to the factorization of A_{22} increases. However, as there is a gain on the factorization time even for the **Just-In-Time** strategy, this increase does not directly translate into a loss of time, since it concerns inefficient low-rank operations on high rank blocks. For the **Just-In-Time** strategy, the **Projections** strategy allows reducing the cost of **HERK**(A_{21}, A_{22}) corresponding to expensive low-rank updates between small and incompressible blocks. This gain directly turns into factorization time gain as those operations are not very efficient (cf. Section 3.3.6).

Some matrices do not take advantage of the **Projections** strategy, as it happens for the **audi** matrix. Indeed, with non regular geometries, the last separator may not be connected to its closest children, leading to zero pre-selected vertices. In such a case, the results obtained for the **Projections** strategy are identical with the **K-way** strategy, as it was shown in Table 5.1 and in Table 5.2.

5.4 Discussion

In this chapter, we analyzed the behavior of existing clustering strategies (k-way partitioning and the reordering strategy introduced in Chapter 4) and proposed a new heuristic to perform clustering and identify non-compressible contributions. We

demonstrated that it can reduce time-to-solution with only a slight memory increase.

In the experiments, we analyzed the advantages of such an approach for both **Minimal Memory** and **Just-In-Time** strategies. We studied this new heuristic on a large set of matrices to exhibit the general trend. We showed a reduction of the time-to-solution by a factor of 10% for the **Minimal Memory** strategy, while the memory consumption slightly increases by a factor of less than 5%.

For future work, we plan to better analyze which blocks are not compressible, for instance considering distances between clusters in the k-way partitioning. Another possibility would be to consider fill-in paths to cluster together unknowns that represent strong interactions, edges that exist in the original graph of A or edges corresponding to ILU(1) or ILU(2) factorizations, which are known to be numerically important for most matrices.

Chapter 6

Partitioning techniques to improve compressibility

In Chapter 4, a reordering strategy aiming at minimizing the number of off-diagonal blocks has been presented. Since the contributions to a separator that are coming from distinct branches of the elimination tree are not the same, it is impossible to order similar contributions from a branch contiguously without breaking ordering for another branch. Thus, we expressed the problem of minimizing the number of off-diagonal blocks as an optimization problem and turn it into a traveling salesman problem. In Chapter 5, a new clustering strategy has been presented, and we have shown that the number of distinct connected components grows quickly with the number of traces considered (cf. Figure 2.1). The heuristic we introduced handles irregular contributions to cluster unknowns, but is limited by the original partition furnished by Scotch. In order to reduce the burden on irregular structures, we propose in this chapter a modified nested dissection algorithm that aligns children separators with respect to their father to exhibit more symmetry in the recursive partitioning process.

In Section 6.1, we present the objective of fixed-vertices algorithms and their variants. Then, we present a new ordering strategy based on fixed vertices in Section 6.2, which intends to align separators to obtain more symmetric interactions between separators. We present some preliminary results in Section 6.3, and limitations of the approach in Section 6.4. Finally, we discuss how the proposed approach can be extended in Section 6.5.

6.1 Background on fixed vertices algorithms

In this section, we introduce fixed-vertices algorithms. We will distinguish the graph partitioning problem, which aims at splitting the vertices of a graph among different parts, and the graph ordering problem, which objective is to number each vertex, for instance to reduce the fill-in or to form a band graph. Typically, k-way takes part of graph partitioning methods while nested dissection belongs to graph ordering methods.

Constraining the partitioning or the ordering can favor several contexts. For

instance, some applications plug in together several components, as it can be the case in a multi-physics environment. In a parallel context, where each component is parallelized with inter-dependencies between components, exhibiting suitable data partitioning is crucial to reduce the communications between distinct components and enhance efficiency. A classic approach consists of dividing each component into a set of tasks and dependencies. Tasks are represented with vertices and dependencies with edges, and the resulting graph can be split using k-way partitioning to load balance computations. However, to avoid increasing a lot the number of communications, one have to express compatible partitioning among distinct components. In order to do so, initially fixed vertices can be used to restrict the partitioning of a graph depending on its coupled graph. Thus, algorithms have to be adapted such that the partitioning of the second graph keeps some vertices in a given part. In [96], Predari et al. proposed a k-way algorithm that manages fixed vertices for coupling applications. Two other examples that make use of initially fixed vertices are the load balancing of adaptive mesh refinement simulations [22] and the circuit design in the context of Very-Large-Scale Integration [26] (VLSI).

Partitioning or ordering graphs are generally realized in a multilevel [71, 77] approach with coarsening and uncoarsening stages, as presented in Figure 6.1. The objective is to contract the graph during coarsening phases into a smaller graph to apply expensive, but leading to good quality, routines such as greedy graph growing (GG) for bi-partitioning [20, 32, 68]. Then, the solution on the smaller graph is projected onto the original graph, with a refinement performed by Fiduccia and Mattheyses (FM) [41] algorithm at each step of the uncoarsening. Note that other methods may be used to perform either the initial partitioning or the refinement. However, in the context of this thesis, we will only consider GG, and its variants, and FM.

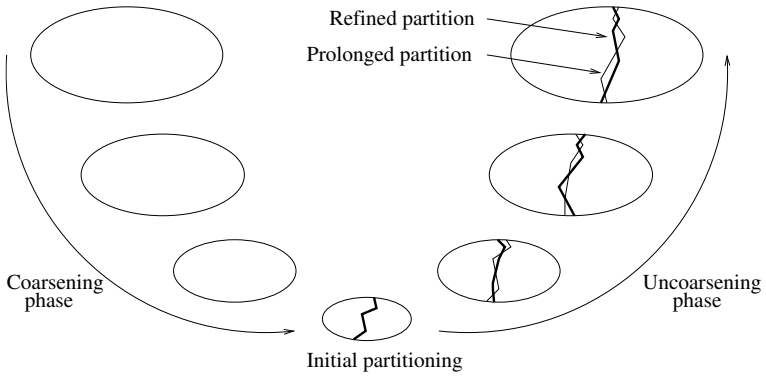


Figure 6.1: The multi-level partitioning process. The original graph is contracted during the coarsening phase, and an initial partitioning is performed on a smaller graph. Then, during the uncoarsening phase, the prolonged partition is refined using FM. This figure was extracted from [92].

In this chapter, we want to investigate the use of such algorithms to exhibit a nested dissection where separators are well aligned. More precisely, partitioning tools such as METIS or SCOTCH split a graph into $A \cup B \cup C$ where C is the separator,

before applying recursively and independently the same process on both A and B. The objective of the algorithm we propose is to fix some vertices of A and B such that their contribution to C will be as identical as possible.

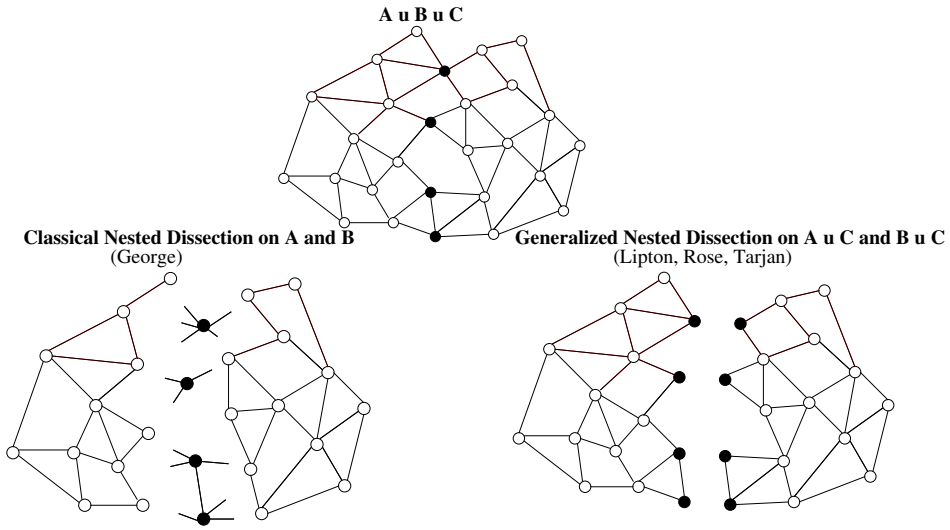


Figure 6.2: On the left, classical recursion is performed on A and B. On the right, recursion is performed on $A \cup C$ and $B \cup C$ to better balance halo vertices during recursion.

In Figure 6.2, we present the classical nested dissection [43] and the generalized nested dissection [80].

In the context of [27], the objective was to ensure good load balancing in a domain decomposition context by assigning to some vertices an initial part number. Contrary to approaches that equally distribute vertices among subdomains, this approach also intends to equally distribute interfaces, **i.e.**, halo vertices, between subdomains. In order to do so, generalized nested dissection was used, to perform bisection on a graph made of a subpart and of vertices belonging to ancestor separators, which correspond to the halo. Then, by partitioning first the halo vertices and using this partition as initial seeds for double greedy graph growing bisection, with two initial sets of seeds, it allows to better load balance interfaces. However, this method does not fix vertices, but only sets initial seeds which can still move from a part to another in the greedy graph algorithm used afterwards.

Contrary to the approach used in [27] that does not really constrain vertices to a given part, we expect to make use of fixed vertices algorithms such as the ones presented in [96].

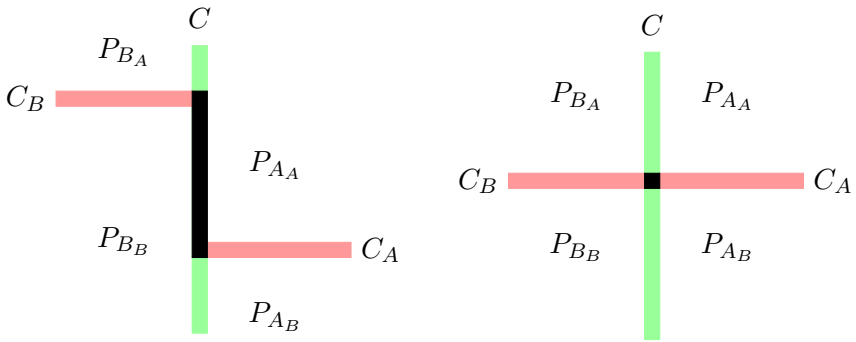
6.2 Modified nested dissection using fixed vertices

In this section, we present the algorithm to perform nested dissection with aligned separators. In Section 6.2.1, we describe one of the limitations of the nested dissection process in terms of data structures. In Section 6.2.2, we introduce graph algorithms that will be used afterwards by the new algorithm presented in Section 6.2.3.

6.2.1 A simple example

Let us present on a simple example why one could take advantage of aligning separators. This idea was already presented in Chapter 4, where non-symmetric contributions increase the number of off-diagonal blocks and in Chapter 5, where it impacts the clustering of unknowns for low-rank compression.

We recall that SCOTCH provides an ordering with unaligned sub-separators, as presented in Figure 6.3(a). We expect to build an ordering where separators are aligned, as presented in Figure 6.3(b). For the sake of simplicity, we consider that this 2D graph is a 5-point regular stencil.



(a) When nested dissection does not match on left and right.

(b) When nested dissection matches on left and right.

Figure 6.3: Two levels of nested dissection on a 2D graph. On the left, non-aligned separators are illustrated, which increases the sets of possible contributions on C. On the right, separators are aligned, which reduces the number of combinations of contributions on C.

With only two levels of nested dissection, one can note that original nested dissection approach that does not align children separators leads to four contribution schemes on the first separator C. When separators C_A and C_B are aligned with respect to their father C, only three contribution schemes remain: contributions from P_{A_A}/P_{B_A}, from C_A/C_B, and from P_{A_B}/P_{B_B}.

Such an ordering strategy is well suited for 1) reducing the irregularity of contributions and thus the number of off-diagonal blocks and 2) form large sets of unknowns receiving the same contribution pattern that will be suitable low-rank clusters (cf. Chapter 5). In addition, the number of vertices that will be pre-selected using projections will be naturally reduced. However, it may degrade the quality of the partition, since it introduces more constraints to compute separators, whose sizes can increase.

6.2.2 Graph algorithms and notations

Before presenting the algorithm to align separators, we describe low-level algorithms that will be used.

Compute a vertex separator. $(A, B, C) = \text{SEP}(G)$. computes a vertex separator C of the graph G such that any path from A to B goes through C.

Partition a graph. $P = \text{PART}(G, k)$ computes a k-part partition of the graph G and returns a table P, where each vertex is indexed by its part number.

Partition a graph with initially fixed vertices. $P' = \text{PARTFX}(G, k, P)$ computes a k-part partition of the graph G with P a set of initially fixed vertices. It returns a tabular P', where each vertex is indexed by its part number.

Transform an edge separator into a vertex separator. $(A, B, C) = \text{ES2VS}(G, P)$ transforms an edge separator **i.e.**, a two-part partition P, into a vertex separator C such that there is no connection between subparts A and B.

6.2.3 A modified nested dissection to align separators

The purpose is to connect separators of subparts A and B to obtain symmetric contributions on the father C. In order to do so, we used the generalized nested dissection to perform recursion on $A \cup C$ and $B \cup C$. Separators are aligned with respect to their direct father only and not with respect to separators higher in the elimination tree.

The approach consists of computing a bisection of the separator C, and used the two colors of those vertices to constrain the ordering of $A \cup C$ and $B \cup C$, with initially fixed vertices. As a partitioning method with fixed vertices is used (which is necessary with existing tools), edge separators obtained are then turned into vertex separators. The first step consists of performing a nested dissection $A \cup B \cup C$ on the original graph which does not contain halo vertices. Then, Algorithm 7 performs the recursion on both subparts.

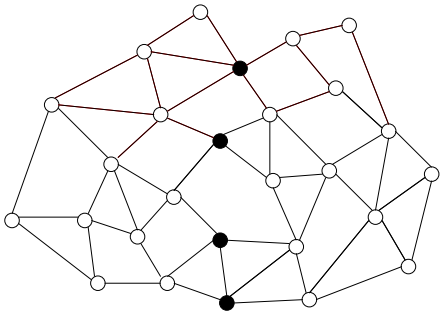
Algorithm 7 ORDER_SUBGRAPHS(A, B, C): ordering of subgraphs A and B separated by C.

$P_C = \text{PART}(C, 2)$ ▷ Bisection of the separator
$P'_A = \text{PARTFX}(A \cup C, 2, P_C)$ ▷ Partition $A \cup C$ with fixed vertices in C
$P'_B = \text{PARTFX}(B \cup C, 2, P_C)$ ▷ Partition $B \cup C$ with fixed vertices in C
$(A_{A_1}, B_{A_1}, C_{A_1}) = \text{ES2VS}(A \cup C, P'_A)$ ▷ Compute vertex separator for $A \cup C$
$(A_{B_1}, B_{B_1}, C_{B_1}) = \text{ES2VS}(B \cup C, P'_B)$ ▷ Compute vertex separator for $B \cup C$
$(A_{A_2}, B_{A_2}, C_{A_2}) = \text{SEP}(A)$ ▷ Classical separator on A
$(A_{B_2}, B_{B_2}, C_{B_2}) = \text{SEP}(B)$ ▷ Classical separator on B
If $(|C_{A_1}| + |C_{B_1}|) > \alpha(|C_{A_2}| + |C_{B_2}|)$ **Then** ▷ Keep classical separators
  ORDER_SUBGRAPHS(A_{A_2}, B_{A_2}, C_{A_2})
  ORDER_SUBGRAPHS(A_{B_2}, B_{B_2}, C_{B_2})
Else ▷ Keep aligned separators
  ORDER_SUBGRAPHS(A_{A_1}, B_{A_1}, C_{A_1})
  ORDER_SUBGRAPHS(A_{B_1}, B_{B_1}, C_{B_1})
End If

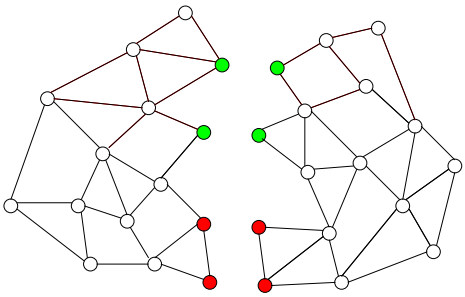
In Algorithm 7, we start by extending the partition of the separator C to either A or B. Then, we perform bi-partitioning with initially fixed vertices and convert the obtained edge separators into vertex separators. We also add a quality constraint

α, which represents the overhead in terms of the size of the separators that we authorize. As we perform partitioning under constraints, one may think that the resulting separators will be larger. Thus, we perform the aligned ordering, which gives separators C_{A_1} and C_{B_1}, and a regular ordering, that results into another separators C_{A_2} and C_{B_2}. We ensure the extra size of the separators is limited by moving back to regular nested dissection if $(|C_{A_1}| + |C_{B_1}|) > \alpha(|C_{A_2}| + |C_{B_2}|)$, with $\alpha > 1$. Note that both orderings may be computed in parallel to reduce preprocessing cost.

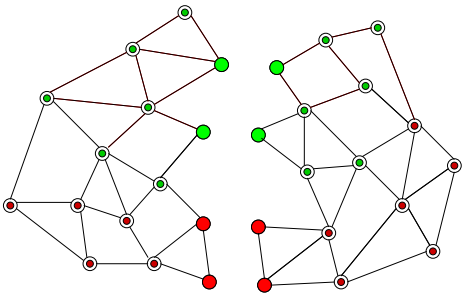
In Figure 6.4, we illustrate the different steps of the algorithm. We start with a classical nested dissection on G, before partitioning C and performing fixed-vertices partitioning of $A \cup C$ and $B \cup C$, which leads to edge separators.



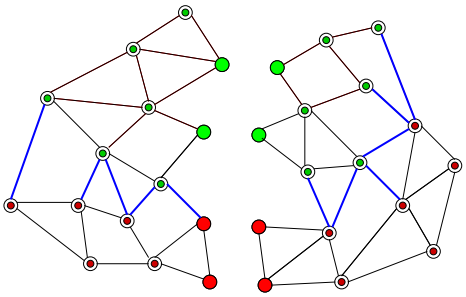
(a) Perform nested dissection on the original graph G. $G = A \cup B \cup C$ where C is the separator.



(b) Apply recursion on both $A \cup C$ and $B \cup C$. C vertices are bi-partitioned into green and red vertices.



(c) Perform partitioning of both $A \cup C$ and $B \cup C$ with initially fixed vertices.



(d) Extract both edge separators before turning them into vertex separators.

Figure 6.4: Different steps to compute aligned separators.

Several methods can be used to turn an edge separator into a vertex separator. Vertex cover [93] performs a maximal matching in the bipartite graphs associated with the edge cuts to compute a vertex separator. However, good vertex separators may not be extracted from good edge separators and this approach is not used in practice. FM can also be used to refine an edge separator into a vertex separator. However, FM cannot be directly applied on the graph made of vertices of the edge separator, as this algorithm requires the knowledge of other vertices to refine the separator by evaluating the cost of moving a vertex inside or outside the separator. Thus, it is applied on the full graph contrary to vertex cover than only works on the

edge separator.

6.3 Preliminary experiments

The objective is to experiment this new heuristic on low-rank strategies presented in Chapter 3. For the full-rank solver, such an approach could enhance the block-symbolic factorization, following the reordering strategy presented in Chapter 4. However, one can expect that a larger size for the separators will degrade at least memory consumption and the number of operations. The enhanced blocking may not be sufficient to increase the efficiency and thus to reduce factorization time. We name the new algorithm ALIGNATOR in the following of this chapter.

For low-rank strategies, there is more room for improvement, since larger separators may be better compressible. As it was shown in Chapter 5, exhibiting suitable low-rank clustering is not straightforward in an algebraic context. We can expect that aligned separators will facilitate this operation and thus increase compressibility. In addition, providing more regular structures may enhance blocking strategy, and it could enhance low-rank kernels which have poor efficiency.

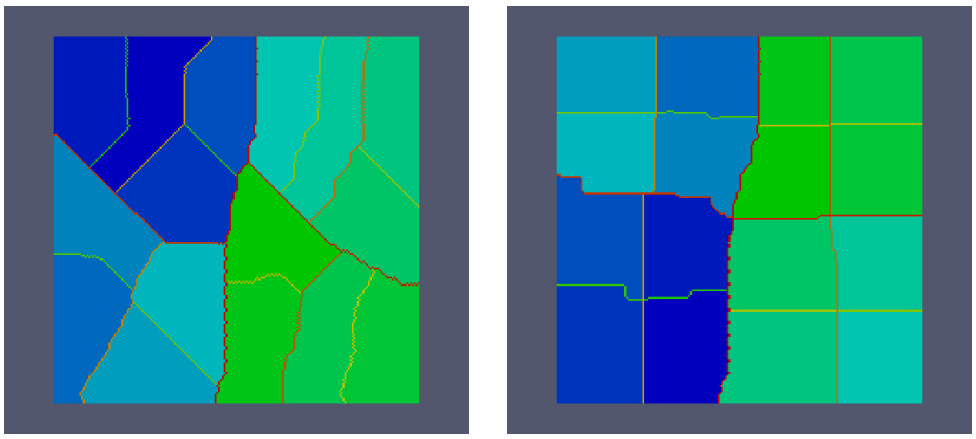
For the graph algorithms, we rely on the STARPART[1] library, which is able to combine together routines coming from different partitioning tools, such as METIS or SCOTCH. We used SCOTCH routines for being able to compare the resulting partition with the one that is classically used in the PASTIX solver. For performing partitioning with initially fixed vertices, we rely on K-Way Graph Greedy Growing algorithm (KGGGP), introduced in [96] and refine separators with a vertex-oriented FM with fixed vertices, which we implemented in SCOTCH.

In Figure 6.5(b), we present the ordering computed by ALIGNATOR on a 200×200 regular grid, which is more regular than the ordering provided by SCOTCH, presented in Figure 6.5(a). One can note that both orderings have the same first separator. ALIGNATOR succeeds into aligning separators with respect to their direct father. However, as it appears in Figure 6.5(b), the separators lower in the elimination tree are not aligned with respect to their common grand parent.

This work is still an ongoing work, and we perform an experiment in a regular 3D Laplacian of size $120 \times 120 \times 120$ to demonstrate the potential of the approach. In Figure 6.6, we present the impact on the sizes of separators using ALIGNATOR. In each node of the tree, the overhead in terms of size of separators is given. It corresponds to the size of the two sub-separators obtained with ALIGNATOR with respect to those obtained with SCOTCH. When this overhead exceeds 10%, separators obtained with regular nested dissection are kept, as highlighted with the red vertex in Figure 6.6. Note that when a single child appears (as in the right on the bottom of the figure), ALIGNATOR was not able to compute separators due to connectivity issues, and SCOTCH is used.

In Table 6.1, we present the statistics obtained for a $120 \times 120 \times 120$ Laplacian using **Minimal Memory** strategy with a 10^{-8} tolerance. We compare SCOTCH and ALIGNATOR for both **K-way** and **Projections** clustering techniques (cf. Chapter 5) in terms of factorization time, number of operations and memory consumption.

[1] https://gitlab.inria.fr/metapart/starpart



(a) Scotch. (b) ALIGnATOR.

Figure 6.5: Scotch versus ALIGnATOR on a 200×200 grid.

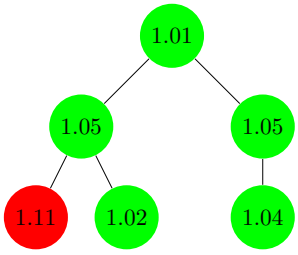


Figure 6.6: Tree of the overhead induced by ALIGnATOR. On each vertex, the overhead is highlighted for the couple of separators that are aligned with respect to their parent. This overhead corresponds to the ratio between the two sub-separators obtained with ALIGnATOR and those obtained with Scotch. For green vertices, aligned separators are kept, while for red vertices, classical nested dissection separators are kept.

We also highlight the behavior of the **Projections** strategy on the last separator, with the number of pre-selected vertices depending on the levels of projection on the elimination tree and the resulting number of connected components.

Ordering	Clustering	Fact (s)	Ops LR (TFlops)	Ops FR (TFlops)	Mem LR (GB)	Mem FR (GB)	Selected (lvl1/lvl2/lvl3)	Number of components
K-way	Scotch	45.09	2.67	14.18	4.08	13	-	1
	Alignator	43.25	2.89	16.16	4.45	14.4	-	1
Projections	Scotch	46.10	2.87	14.18	4.6	13	1230 / 1113 / 1541	33
	Alignator	40.95	2.90	16.16	4.64	14.4	816 / 508 / 487	7

Table 6.1: Factorization time, number of operations and memory consumption using either Scotch or ALIGnATOR to perform partitioning. Those results correspond to the use of the **Minimal Memory** strategy with a 10^{-8} tolerance for a $120 \times 120 \times 120$ Laplacian, with **K-way** and **Projections** clustering strategies. The number of selected vertices (level by level) as well as the number of connected components is shown for the last separator only. LR corresponds to low-rank operations and FR to full-rank operations.

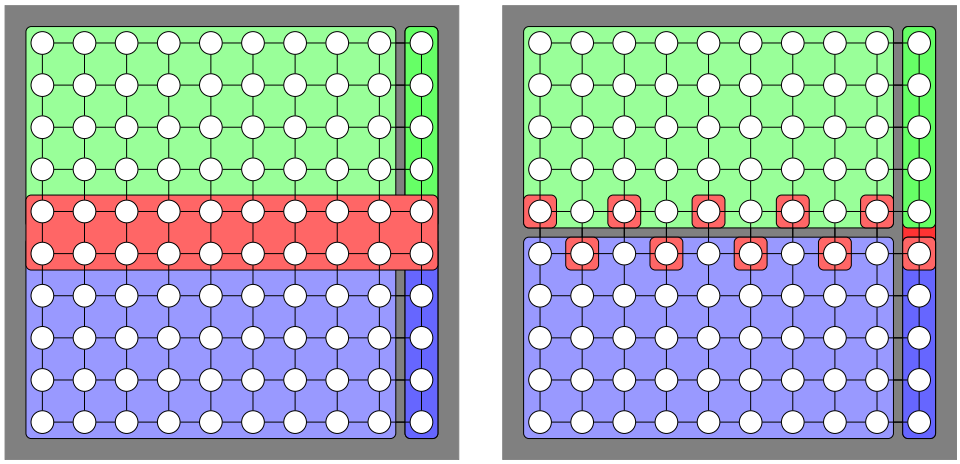
One can notice that ALIGnATOR is able to reduce the factorization time with respect to Scotch for both clustering strategies. In terms of overhead, both the number of operations and the memory consumption increase with respect to Scotch for the full-rank statistics. However, as data structures are more compressible, this overhead is hidden in the low-rank approach, for which both number of operations and memory consumption are similar when using Scotch. In terms of pre-selection, ALIGnATOR is, as expected, able to reduce by a large factor the number of pre-selected vertices, and thus the number of connected components, which explains the time-to-solution reduction.

This approach is particularly interesting together with the **Projections** heuristic as it helps to better classify contributions. We expect to evaluate the potential of the method on less regular geometries in future works.

6.4 Limitations

Several details limit the assets of the approach. First, there is a connectivity problem using FM, because the graph of the separator can be totally disconnected. For instance, the optimal edge separator of a 2D grid corresponding to a 5-point stencil can be totally disconnected, as it is presented in Figure 6.7(b), where FM was used to refine the edge separator presented in Figure 6.7(a). It limits the use of k-way partitioning to compute initial fixed vertices, as it is better suited for connected graphs. For this reason, we reconnect separators with a path of distance 1 (or 2 if it is not sufficient) in the original graph. Then, FM is not able to keep fixed vertices as it is executed on a larger graph and not only the graph of the separator being refined, and initially fixed vertices can still move to another part. For this reason, we implemented a vertex-oriented FM in Scotch with vertices that are fixed in the separator.

If using SEP() and PARTFX() should lead to subparts with a similar size, the



(a) Edge separator. (b) Vertex separator.

Figure 6.7: FM on a regular grid may lead to a disconnected vertex separator.

use of fixed vertices may increase the size of separators. The partitioning methods with fixed vertices used in the implementation of ALIGNATOR were developed in Predari's thesis [95]. An ongoing work is to implement properly the same method in a vertex-oriented fashion all throughout the process. One can expect that directly computing a vertex separator in the multilevel process will provide better results than those obtained by refining edge-oriented separators.

6.5 Discussion

In this chapter, we proposed a modified nested dissection process aiming at aligning separators. The approach consists of using generalized nested dissection and introducing fixed vertices to ensure that the contributions coming from direct children will be symmetric on their father. We demonstrated the impact of the approach on a regular graph and give some insights to generalize the algorithm for more cases.

This is still an ongoing work, since the algorithm could have been enhanced using vertex-oriented methods with initially fixed vertices. A first step would be to implement a vertex-oriented variant of KGGGP to avoid refining edge separators. In addition, separators are only aligned with respect to their father, while it may be possible to include more constraints by considering more levels in the recursion. An extended approach would consist of keeping the full halo all throughout computations. Then, when a subgraph is partitioned, some constraints can be defined to align the separator with respect to separators obtained previously on other branches of the elimination tree.

In future work, we expect to introduce some metrics to better control when aligning is worthwhile. In addition, it could be interesting to maintain connected separators, for instance using a max-flow algorithm [18]. Indeed, it allows to reduce the halo sizes and introduces more regularity. An idea would be to implement a FM algorithm with initially fixed vertices that maintain the separator connectivity.

Conclusion and perspectives

Solving sparse linear systems is a critical operation in many real-life applications. In this thesis, we focused on sparse direct solvers, which provide accurate solutions but have high time and memory complexities. Several solvers have recently introduced the use of low-rank compression techniques to reduce those complexities.

In this thesis, we introduced low-rank compression in the PaStiX solver and developed several ordering methods to better manage granularity and low-rank approximations in the sparse context. This approach allows to obtain a solution close to the prescribed accuracy, which outperforms most of existing approaches that are used as preconditioners and cannot provide a very high accuracy solution. The developments are available publicly in the PaStiX solver. Both the level of parallelism and the use of runtime systems have been kept untouched with the integration of low-rank kernels, so the low-rank strategies can benefit to all available features of the solver. Using Parsec runtime system, we are able to reduce both time-to-solution and memory consumption for relatively small problems, made of several millions of unknowns (cf. Table B.1).

The supernodal context makes this approach different with many other solvers that rely on the multifrontal method, due to the differences in the update process. In addition, the low-rank compression strategies are new in the sense that low-rank assembly is performed in a fully algebraic context, which was not done in the literature for the sparse case. However, it introduces new challenges to correctly manage sparse structures together with low-rank compression. For this reason, we studied in this thesis various ordering algorithms that allow to reduce time-to-solution by increasing block compressibility.

Contributions

In Chapter 3, we introduced low-rank compression in the PaStiX solver with two new strategies: **Minimal Memory** and **Just-In-Time**. We demonstrated that the **Just-In-Time** strategy reduces the time-to-solution on a large set of real-life matrices. The **Minimal Memory** strategy may slightly increase time-to-solution, especially for relatively small matrices, but reduces memory consumption, allowing to solve problems that would not have fit in memory with the original solver. With only 128 GB of memory, a 3D Laplacian of size 330^3 has been solved while the original solver was limited to 200^3 unknowns. While we have seen it has limitations to reduce the time-to-solution of "average" problems, the **Minimal Memory** strategy reduces time-to-solution of larger problems.

In Chapter 4, we proposed a reordering strategy that reduces the number of off-diagonal blocks appearing in the block-symbolic factorization. We succeeded to reduce this number by a large factor and showed that it can enhance the solver by up to 20% when using accelerators. In addition, this approach both reduces the number of updates for low-rank strategies and increases compression rates.

In Chapter 5, we introduced a new clustering heuristic in an algebraic fashion and showed how it enhances the low-rank strategies in the PaStiX solver. We studied its behavior on a large set of matrices to illustrate the time-to-solution gain with only a slight memory increase. For instance, it reduces time-to-solution by 10% on average for the **Minimal Memory** strategy while the memory consumption overhead is only at a factor of 5%. Such a method is not only dedicated to the supernodal approach but may enhance multifrontal solvers too, especially with the use of low-rank assembly.

In Chapter 6, we proposed a new algorithm to perform nested dissection with aligned separators. It allows to obtain more regular sparse patterns and thus to compute a better clustering thanks to the heuristic developed in Chapter 5. This is still an ongoing work, but results on a regular cube showed better compression rates. Such an approach can be promising to better control "where and when" compression occurs and to increase the overall block compressibility.

Around those main contributions, we also studied the behavior of low-rank strategies when using the Parsec runtime system in Appendix B. It shows that using a runtime system, that handles dynamically load balancing, reduces time-to-solution, especially for irregular kernels such as those appearing with low-rank compression. The low-rank strategies have also been studied in the domain decomposition solver Horse in Appendix C. It demonstrates the impact of the solver on a large real-life experiment.

The introduction of low-rank strategies together with the different ordering, clustering and partitioning strategies succeeded into reducing the time-to-solution and/or the memory consumption of the PaStiX solver for real-life matrices. The contributions of Chapter 3 and Chapter 4 have been integrated into the last release of the solver, while the contributions presented in Chapter 5 will be integrated in the next release. The algorithm presented in Chapter 6 is still an on-going work and is developed into the StarPart library.

Perspectives

A challenging problem is to extend the low-rank compression strategies for distributed architectures. If the basic kernels will be kept untouched, controlling the number and the size of communications is an open problem, especially in a fully-structured approach where low-rank updates are used. Different strategies can be developed for communicating, for instance compressing data blocks before or after communications. We are convinced that using low-rank updates is a good solution to reduce as much as possible memory consumption and the volume of communications.

Another perspective is to replace BLR compression by the use of hierarchical formats. Still, low-rank updates make this problem challenging since small blocks contribute to larger blocks. In existing solvers using hierarchical formats, this prob-

lem is hidden with the knowledge of the geometry or the use of randomization techniques. The clustering strategies proposed may help with this problem as well as the new algorithm to perform nested dissection. If these studies were conducted for BLR compression, they could be extended for hierarchical compression. Those algorithms consider several levels in the elimination tree and a hierarchy can be exhibited by the natural recursion over levels.

In [74], the use of accelerators has been introduced in the PaStiX solver. Using this type of computational unit in a sparse context is a challenging problem, due to the irregularity of operations. However, it demonstrated significant speedup over the multi-threaded version of the PaStiX solver. Low-rank compression kernels can also be performed on the GPU, but may require to find a compromise between accuracy and efficiency. For instance, using fixed ranks or ranks as a multiple of a given blocking size may allow better management on the GPUs.

Concluding remarks

The HiePACS team at Inria Bordeaux - Sud-Ouest will continue to investigate the algorithms introduced in this thesis. In particular, hierarchical matrices will be introduced in the PaStiX solver in future works.

The different ordering, clustering and partitioning methods are not dedicated to the PaStiX solver, but can be used in other sparse direct solvers, or even for hybrid solvers, such as the ones relying on domain decomposition. Thus, it could be interesting to implement those methods in an external tool that provides a partition such as the one provided by Metis or Scotch, but with extra information to better manage low-rank compression.

Appendix A

Experimental environment

In this appendix, we present the context in which experiments were performed. We describe the platforms on which experiments were conducted, as well as the parameters used to tune the PaStiX solver. In addition, we present the different set of matrices that have been used. In Section A.1, we describe the environmental conditions of Chapter 4, which was the first work conducted in this thesis. Some machines were upgraded afterwards, and we switched to a new platform for other chapters. In Section A.2, we present the environmental conditions used for Chapter 3, Chapter 5 and Chapter 6.

The PaStiX version used for experiments of Chapter 3 and Chapter 4 is available on the public git repository[1] as the tag `6.0.1`. For Chapter 5, it will be integrated in the next release. The multi-threaded version used is the static scheduling version presented in [74].

A.1 Context reordering

Experiments for Chapter 4 were performed on the **Plafrim**[2] supercomputer. For the performance experiments on a heterogeneous environment, we used the `mirage` cluster, where nodes are composed of two Intel `Westmere Xeon X5650` hexa-core CPUs running at 2.67 GHz with 36 GB of memory, and enhanced by three Nvidia GPUs, `M2070`. For the scalability experiments in a multi-threaded context, we used the `miriel` cluster, where each node is equipped with two Intel `Xeon E5-2680 v3` 12-cores running at 2.50 GHz with 128 GB of memory. We used Intel MKL `2016.0.0` for the BLAS kernels on the CPUs, and we used the Nvidia Cuda `7.5` development kit to compile the GPU kernels.

The PaStiX version used for those experiments is the one implemented on top of the Parsec [25] runtime system and presented in [76].

For the initial ordering step, we used Scotch 5.1.11 with the configurable strategy string from PaStiX to set the minimal size of non-separated subgraphs, *cmin*, to be 20, as it appears in [74]. We also set the *frat* parameter to 0.08, meaning that column aggregation is allowed by Scotch as long as the fill-in introduced does not

[1] https://gitlab.inria.fr/solverstack/pastix
[2] https://plafrim.bordeaux.inria.fr

exceed 8% of the original matrix. It is important to use such parameters to increase the width of the column blocks and reach a good level of performance using accelerators. Even if it increases the fill-in, the final performance gain is usually more important and makes the memory overhead induced by the extra fill-in acceptable.

In order to validate the behavior of the reordering heuristic presented in Chapter 4, we used a set of large matrices arising from real-life applications originating from the SuiteSparse Matrix Collection [33]. More precisely, we took all matrices from this collection with a size between $500,000$ and $10,000,000$. From this large set, we extracted matrices that are applicants for solving linear systems. Thus, we removed matrices originating from the Web and from DNA problems. This final set is composed of 104 matrices, sorted by families. In the same chapter, we also conduct some experiments with a matrix of 10^7 unknowns, taken from a CEA simulation, an industrial partner in the context of the PaStiX project.

A.2 Context BLR and clustering

Experiments for Chapter 3, Chapter 5 and Chapter 6 were conducted on the **Plafrim** supercomputer, and more precisely on the `miriel` cluster. Each node is equipped with two Intel `Xeon E5-2680 v3` 12-cores running at 2.50 GHz with 128 GB of memory. The Intel MKL `2017` library is used for BLAS and SVD kernels. The RRQR kernel is issued from the BLR-MUMPS solver [8], and is an extension of the block rank-revealing QR factorization subroutines from LAPACK 3.6.0 (xGEQP3) to stop the factorization when the precision is reached.

For the initial ordering step, we used Scotch 6.0.4 with the configurable strategy string from PaStiX to set the minimal size of non-separated subgraphs, *cmin*, to 15. We also set the $frat$ parameter to 0.08, meaning that column aggregation is allowed by Scotch as long as the fill-in introduced does not exceed 8% of the original matrix.

In experiments, blocks that are larger than 256 are split into blocks of size within the range $128 - 256$ to create more parallelism while keeping sizes that are large enough to reach good efficiency. The same 128 criteria is used to define the minimal width of the column blocks that are compressible. An additional limit on the minimal height to compress an off-diagonal block is set to 20. We set the rank ratio (cf. Section 3.3.6.3) to 1 to illustrate the results when the rank is as strict as possible to obtain Flops and/or memory gains. CGS orthogonalization is also used. Note that in Section 3.3.6 we try different blocking sizes to experiment the impact on the solver, relax the maximum ranks authorized, and compare the orthogonalization strategies.

However, those parameters are kept invariant in Chapter 5, and we set the blocking sizes to 128 and 256. We also use CGS orthogonalization and set the rank ratio to 1.

Note that when results showing numerical precision are presented, we used the backward error on b, given by $\frac{||Ax-b||_2}{||b||_2}$.

Experiments for Chapter 3 and Chapter 5 were computed on a set made of five 3D matrices issued from The SuiteSparse Matrix Collection [33], highlighted with stars in Table A.1. We also used 3D Laplacian generators (7-point stencils), and defined $lap120$ as a Laplacian of size 120^3.

In Chapter 6, we used a larger set of 32 matrices described in Table A.1, as well as *lap*120 to highlight the behavior of different clustering strategies.

Kind		Matrix	Arith.	Fact.	N	NNZ_A	TFlops	Memory (GB)
2d/3d		PFlow_742	d	LL^t	742793	18940627	1.4	4.3
		Bump_2911	d	LL^t	2911419	65320659	204.9	78.3
Computational fluid dynamics		StocF-1465	d	LL^t	1465137	11235263	3.6	8.7
		atmosmodl	d	LU	1489752	10319760	10.1	16.7
		atmosmodd	d	LU	1270432	8814880	12.1	16.3
	*	atmosmodj	d	LU	1270432	8814880	12.1	16.3
		RM07R	d	LU	381689	37464962	15.7	16.0
Dna electrophoresis		cage13	d	LU	445315	7479343	356.2	76.3
Electromagnetics		dielFilterV3clx	z	LU	420408	16653308	1.3	5.2
		fem_hifreq_circuit	z	LU	491100	20239237	1.6	6.0
		dielFilterV2clx	z	LU	607232	12958252	2.1	7.0
Magnetohydrodynamics		matr5	d	LU	485597	24233141	8.4	10.5
Materials		3Dspectralwave2	z	LDL^h	292008	7307376	6.5	6.3
Model reduction		boneS10	d	LL^t	914898	28191660	0.3	2.5
		CurlCurl_3	d	LDL^t	1219574	7382096	3.8	6.5
		bone010	d	LL^t	986703	36326514	4.4	9.4
		CurlCurl_4	d	LDL^t	2380515	14448191	13.7	15.7
Optimization		nlpkkt80	d	LDL^t	1062400	14883536	27.3	17.9
Structural		ldoor	d	LL^t	952203	23737339	0.1	1.2
		inline_1	d	LL^t	503712	18660027	0.1	1.5
		Flan_1565	d	LL^t	1564794	59485419	3.7	12.3
		ML_Geer	d	LU	1504002	110879972	4.2	17.2
	*	audikw_1	d	LL^t	943695	39297771	5.5	9.5
		Fault_639	d	LL^t	638802	14626683	7.7	9.0
	*	Hook_1498	d	LL^t	1498023	31207734	8.6	12.7
		Transport	d	LU	1602111	23500731	10.2	20.8
		Emilia_923	d	LL^t	923136	20964171	12.7	13.5
	*	Geo_1438	d	LL^t	1437960	32297325	18.0	20.1
	*	Serena	d	LL^t	1391349	32961525	28.6	21.7
		Long_Coup_dt0	d	LDL^t	1470152	44279572	47.1	31.9
		Cube_Coup_dt0	d	LDL^t	2164760	64685452	87.2	51.6
		Queen_4147	d	LL^t	4147110	166823197	251.8	110.0

Table A.1: Set of real-life matrices issued from The SuiteSparse Matrix Collection, sorted by family and number of operations. The set of five matrices used in most experiments is highlighted with stars.

Appendix B

PaStiX BLR over runtime systems

Exploiting modern architectures made of heterogeneous computational units is a challenging problem. A crucial criterion for being able to run an application over various machines is the ability to adapt dynamically parallelism and reduce the constraints introduced by performing statically load balancing before any computations. As presented in Section 1.3.2, the parallelism inside PaStiX can be exploited through different scheduling policies. Most of the developments performed in the context of this thesis have been experimented with the static scheduling strategy.

Over the past years, the use of runtime systems have gained lot of interest since it allows taking advantage of generic scheduling strategies without directly managing parallelism inside each library. It demonstrated outstanding results for dense linear algebra on multicore architectures for the Plasma [73] library and for distributed architectures for the Dplasma [24] library.

Runtime systems have been introduced recently in the PaStiX solver [74, 76]. More precisely, both Parsec [25] or StarPU [19] can be used on distributed heterogeneous architectures. It is especially interesting when using GPUs or other type of accelerators such as Intel Xeon Phi, as computational kernels are independent with scheduling constraints. For instance, in Chapter 4, we evaluated the impact of the reordering strategy on heterogeneous architectures, using Parsec runtime system to split computations among CPUs and GPUs. Using runtimes systems is an active research area, especially for irregular computations such as those appearing in sparse arithmetic. Note that the use of runtime systems has also been studied for the sparse QR factorization in [4].

In this appendix, we study the impact of using runtime systems together with low-rank compression kernels. Since the level of parallelism exhibited with both **Minimal Memory** and **Just-In-Time** strategies follows the original, full-rank, version of the solver, adding extra dependencies is not required, and the code corresponding to task submission for both Parsec and StarPU is kept untouched. One can also expect that using runtime systems will provide better load balancing, since it is impossible to predict algebraically the distribution of ranks before any numerical operation. In this appendix, we will only study the use of the Parsec runtime system.

Static scheduling. With the static scheduling strategy, the set of supernodes obtained after splitting is divided regularly (in terms of number of operations) among

the set of working threads. In addition, some locks are used to avoid simultaneous updates of a block by several threads. The low-rank strategies introduced in Chapter 3 use the same approach, but fail to equally distribute computations among threads due to the irregularity of low-rank computations.

PARSEC. With the Parametrized Task Graph (PTG) approach of PARSEC, each task is explicitly described together with its dependencies. A language, named JDF, allows describing tasks and their dependencies.

STARPU. With the Sequential Task Flow (STF) approach of STARPU, the dependencies between tasks are not explicit and follow the order in which tasks are submitted. Each task is described with a codelet, which can point to several computational kernels (for the CPU, for the GPU...) when using heterogeneous architectures. Then, STARPU can choose on-the-fly on which type of computational unit the task will be scheduled. Handlers on supernodes or blocks (depending if a 1D or a 2D level of parallelism are used) are used to control dependencies between tasks.

Experiments. We perform experiments for the set of five matrices presented in Table A.1, as well as for a 3D Laplacian of size 120^3, with the static scheduling and PARSEC runtime system. Note that both scheduling strategies lead to the same accuracy, and the same memory consumption for the factors.

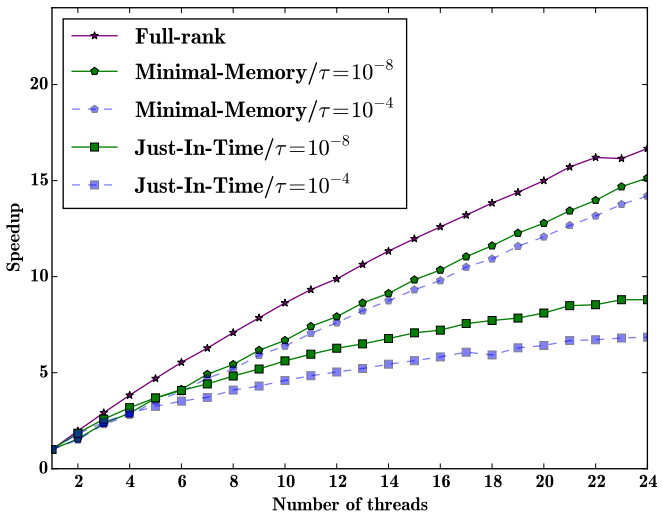


Figure B.1: Speedup of the factorization for the **atmosmodj** matrix with 1 to 24 threads for full-rank and both low-rank strategies with $\tau = 10^{-4}$ and $\tau = 10^{-8}$, using PARSEC.

Figure B.1 presents the speedup of the full-rank factorization and low-rank strategies using tolerances of 10^{-4} and 10^{-8} for the **atmosmodj** matrix, using PARSEC. The reference taken to compute the speedup is the sequential run using PARSEC. This figure is to compared with Figure 3.9, which presented the same experiments,

but with the use of static scheduling. One can note that the speedup is enhanced using Parsec. For the full-rank strategy, the speedup increases from 12.9 to 16.7 when using Parsec. For the low-rank strategies using a 10^{-8} tolerance, it goes from 11.1 to 15.1 for the **Minimal Memory** strategy, and from 9.1 to 8.8 for the **Just-In-Time** strategy. Thus, we can expect that time-to-solution is reduced with respect to static scheduling.

In order to analyze different heuristics and parameters we studied all throughout this thesis, we perform experiments to see how the performance (presented in Figure 3.5(b) for **Minimal Memory** strategy and in Figure 3.5(a) for **Just-In-Time** strategy) is impacted by the use of Parsec and several parameters. In order to ensure that reduction of time-to-solution does not increase memory consumption, we also study memory consumption, to compare with initial results that were presented in Figure 3.6.

In next figures, we present the impact of using parameters or heuristics described in this thesis on low-rank strategies for static scheduling and Parsec. In Figure B.2, we present the results when off-diagonal blocks touching the diagonal are not compressed (strong admissibility for those blocks). In Figure B.3, we add k-way partitioning to show how it impacts the solver, before presenting in Figure B.4 the assets of using projections. Finally in Figure B.5 (respectively in Figure B.6), we keep using projections, and we relax the maximum rank ratio (cf. Section 3.3.6.3) to 0.5 (respectively 0.25).

In terms of memory consumption, different variants of parameters have only a slight impact. However, performance is enhanced using projections, especially together with a maximum rank ratio relaxed to 0.5, as it can be shown in Figure B.5(b). Overall, Parsec outperforms static scheduling for most cases. It is the case in full-rank arithmetic and for the **Minimal Memory** strategy. For the **Just-In-Time** strategy, it also reduces time-to-solution, except for the **lap120** matrix.

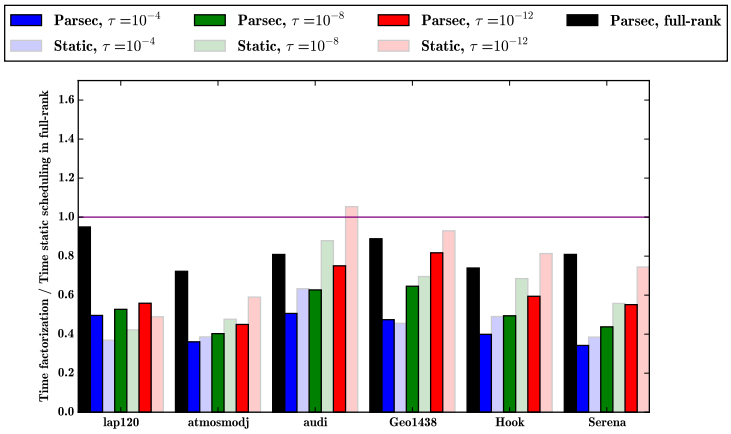
In Table B.1 we summarize the average gain of the five variants over the six matrices that are studied for both **Minimal Memory** and **Just-In-Time** strategies using a 10^{-8} tolerance with respect to the full-rank factorization performed with static scheduling. For Parsec and static scheduling, the average gain corresponds to the average time-to-solution with respect to the full-rank version used together with static scheduling. We also show the impact on memory consumption for the **Minimal Memory** strategy. We recall that the Parsec scheduling strategy leads to the same memory consumption as the static scheduling. Thus, the memory consumption results are to be considered only depending on the variant used and not the scheduling strategy. One can notice that there is only few impacts for the **Just-In-Time** strategy. However, **Minimal Memory** strategy is better using projections together with a maximum rank factor relaxed to 0.5. For this strategy, using Parsec leads to much better results.

For the variant using projections together with a maximum rank factor relaxed to 0.5 and Parsec runtime system, we succeed to reduce both time-to-solution and memory consumption on average on a set of relatively small matrices. It is particularly interesting since we have shown in Section 3.3.6 that the efficiency of low-rank kernels is not that good and it is difficult to reduce time-to-solution for small matrices, despite the reduction of the number of operations. Thus, one can

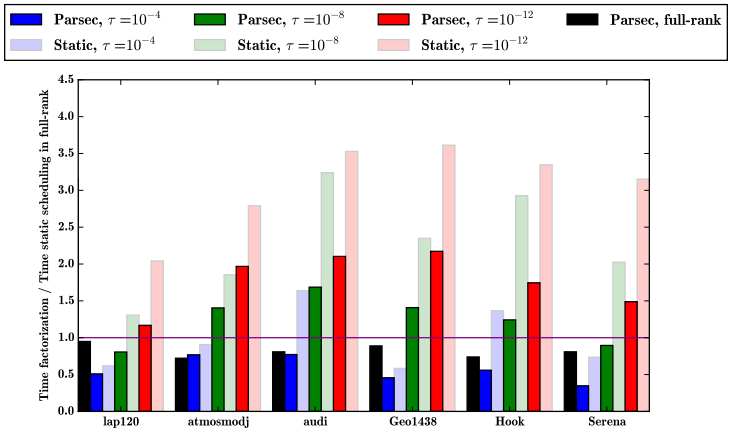
expect higher time-to-solution gains with larger matrices, as it was illustrated in Figure 3.7(b).

Variant	**Just-In-Time**		**Minimal Memory**		
	Time PARSEC	Time static	Time PARSEC	Time static	Memory
strong admissibility	0.52	0.62	1.24	2.28	0.52
+ k-way	**0.49**	0.55	1.17	1.94	**0.48**
+ projections	0.50	0.55	1.19	1.90	0.50
+ ratio set to 0.5	0.51	0.56	**0.96**	1.43	0.54
+ ratio set to 0.25	0.50	0.55	1.19	1.89	0.50

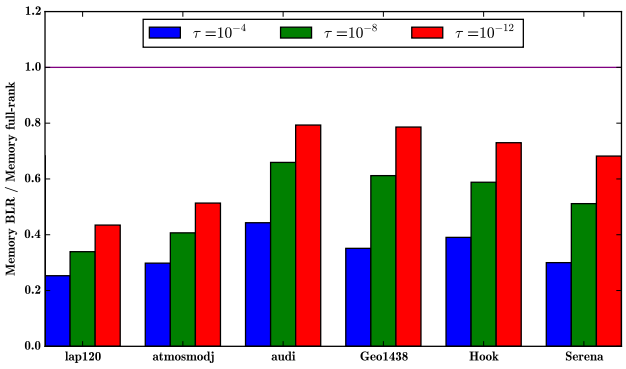
Table B.1: Average gain on factorization time and memory consumption for **Minimal Memory** and **Just-In-Time** strategies on six matrices using a 10^{-8} tolerance with respect to the full-rank factorization performed with static scheduling. Several variants are presented to highlight how parameters impact the solver.



(a) Factorization for **Just-In-Time** scenario using RRQR.

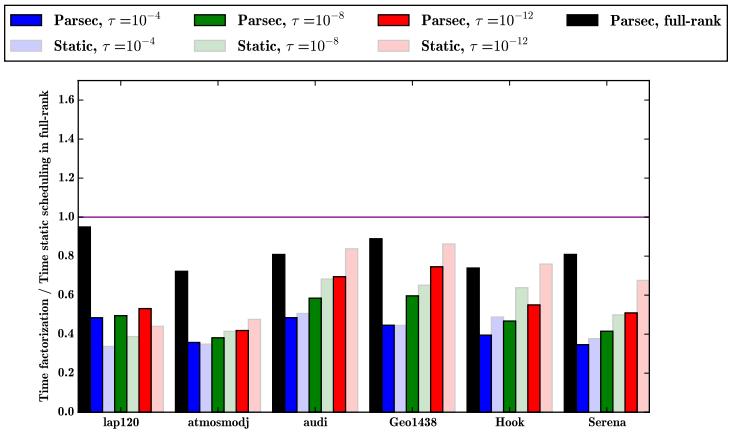


(b) Factorization for **Minimal Memory** scenario using RRQR.

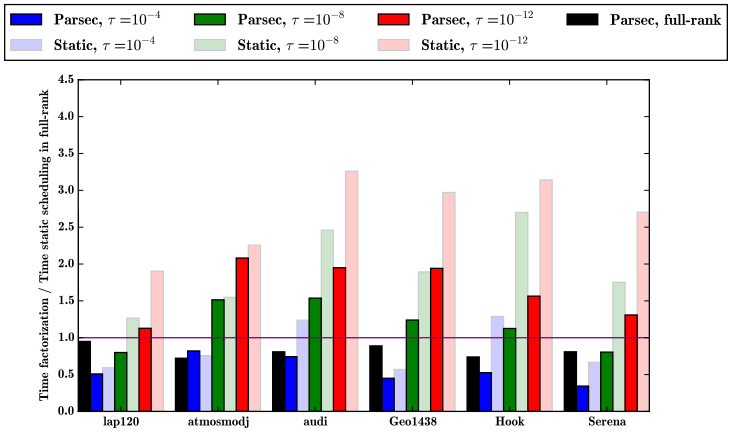


(c) Memory consumption for **Minimal Memory** scenario using RRQR.

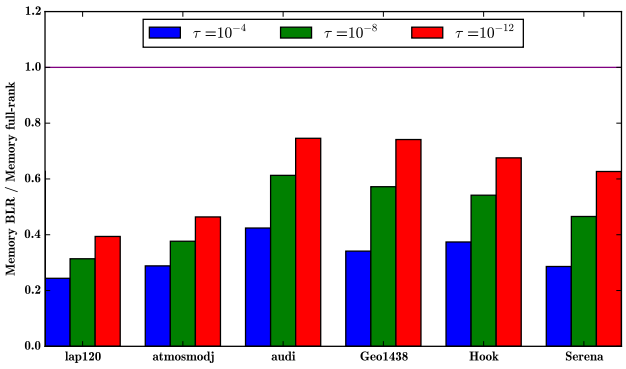
Figure B.2: Performance and memory consumption of both low-rank strategies with three tolerance thresholds, using static scheduling or Parsec. In this variant, off-diagonal blocks touching the diagonal are not compressed.



(a) Factorization for **Just-In-Time** scenario using RRQR.

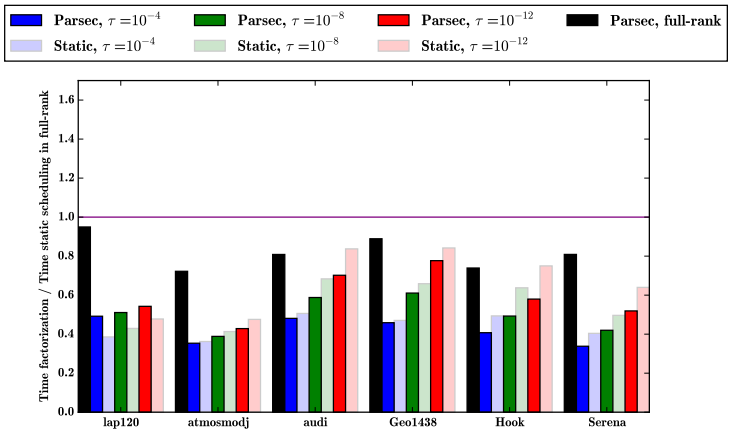


(b) Factorization for **Minimal Memory** scenario using RRQR.

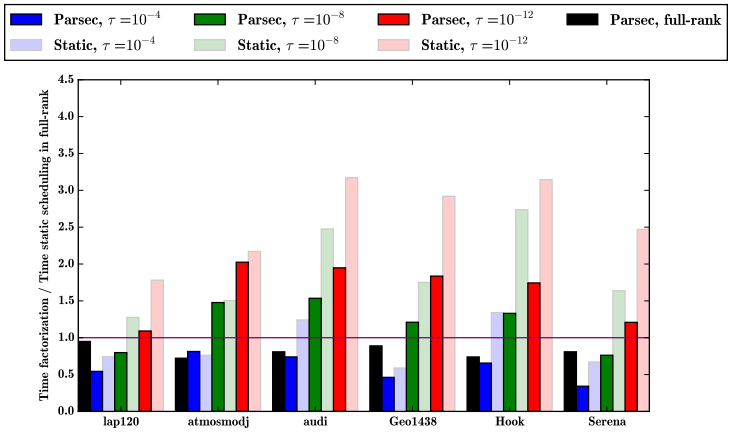


(c) Memory consumption for **Minimal Memory** scenario using RRQR.

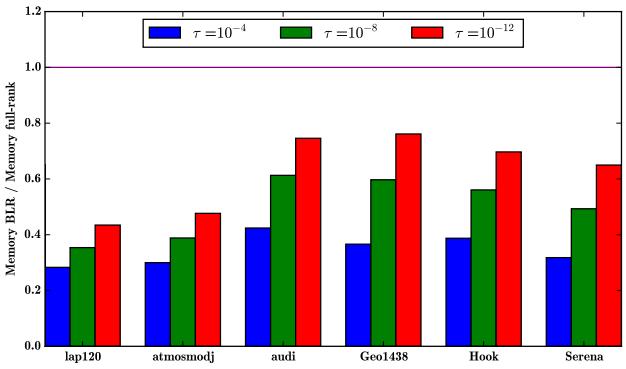
Figure B.3: Performance and memory consumption of both low-rank strategies with three tolerance thresholds, using static scheduling or PARSEC. In this variant, off-diagonal blocks touching the diagonal are not compressed and k-way partitioning is used for the clustering.



(a) Factorization for **Just-In-Time** scenario using RRQR.

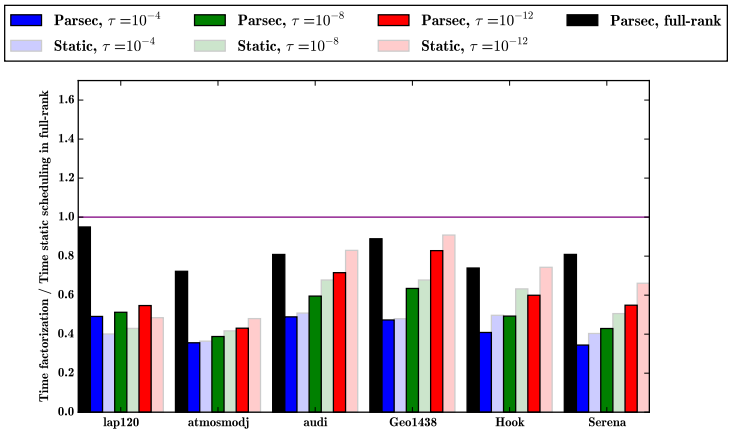


(b) Factorization for **Minimal Memory** scenario using RRQR.

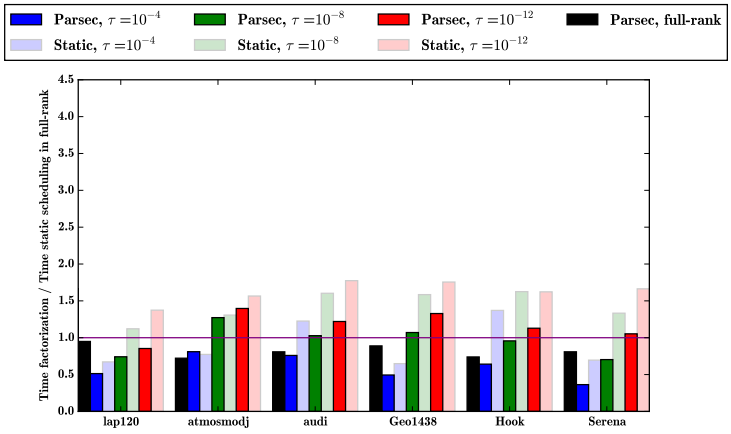


(c) Memory consumption for **Minimal Memory** scenario using RRQR.

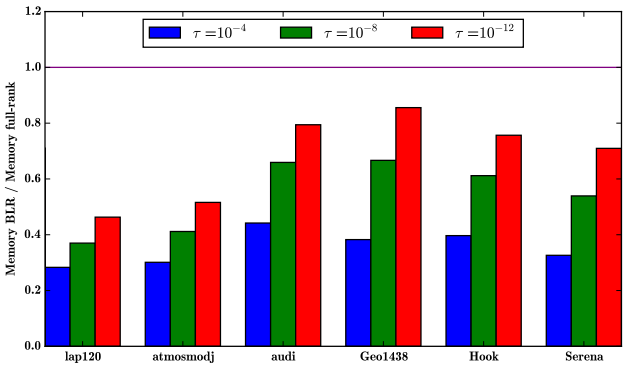
Figure B.4: Performance and memory consumption of both low-rank strategies with three tolerance thresholds, using static scheduling or Parsec. In this variant, off-diagonal blocks touching the diagonal are not compressed and projections heuristic is used for the clustering.



(a) Factorization for **Just-In-Time** scenario using RRQR.

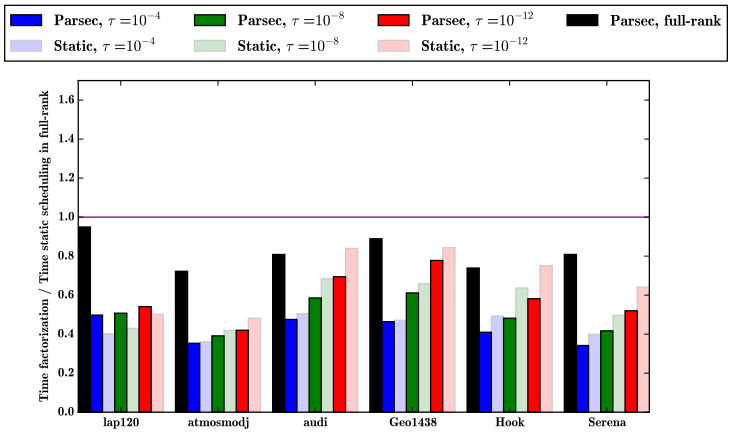


(b) Factorization for **Minimal Memory** scenario using RRQR.

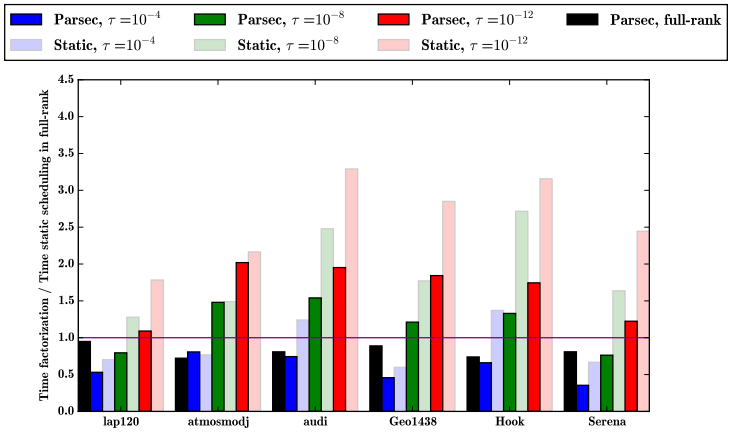


(c) Memory consumption for **Minimal Memory** scenario using RRQR.

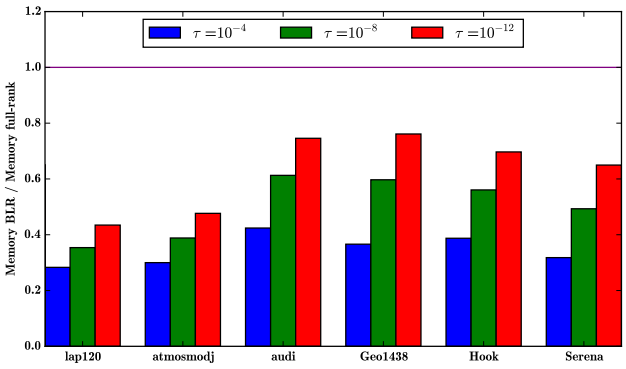
Figure B.5: Performance and memory consumption of both low-rank strategies with three tolerance thresholds, using static scheduling or PARSEC. In this variant, off-diagonal blocks touching the diagonal are not compressed and projections heuristic is used for the clustering. In addition, the maximum rank ratio is relaxed to 0.5.



(a) Factorization for **Just-In-Time** scenario using RRQR.



(b) Factorization for **Minimal Memory** scenario using RRQR.



(c) Memory consumption for **Minimal Memory** scenario using RRQR.

Figure B.6: Performance and memory consumption of both low-rank strategies with three tolerance thresholds, using static scheduling or Parsec. In this variant, off-diagonal blocks touching the diagonal are not compressed and projections heuristic is used for the clustering. In addition, the maximum rank ratio is relaxed to 0.25.

Appendix C

PaStiX BLR inside the domain decomposition solver Horse

In this appendix, we study the behavior of the PaStiX solver using block low-rank compression into a real-life simulation performed with Horse [78]. Horse solves the frequency-domain Maxwell equations discretized by a high order Hybrid Discontinuous Galerkin (HDG) method. Its approach consists of solving a reduced system on faces instead of elements, which leads to smaller systems than those obtained through continuous or discontinuous Galerkin (CG and DG) methods. Then, a Schwarz additive domain decomposition is used, where a sparse system is solved on each subdomain using a sparse direct solver. The objective of this study is to analyze the behavior when replacing the full-rank version of the PaStiX solver by the low-rank strategies we introduced in Chapter 3.

Method. Horse solves the electromagnetic (EM) field by introducing a hybrid variable named $\mathbf{\Lambda}$ that forms a system on faces instead of elements. This hybrid variable reduces the number of elements of the system with respect to other existing approaches, as we present in the following:

- Classical DG method with $\mathbb{P}_p$ interpolation:

$$(p+1)(p+2)(p+3)N_e, N_e \text{ is the number of elements}$$

- HDG method with $\mathbb{P}_p$ interpolation:

$$(p+1)(p+2)N_f, N_f \text{ is the number of faces}$$

- Continuous finite element formulation based on Nedelec's first family of face/edge elements in a simplex (tetrahedron):

$$\frac{p(p+2)(p+3)}{2}N_e$$

Table C.1 gives the number of unknowns for an unstructured tetrahedral mesh with $1,645,874$ elements and $3,521,251$ faces with different levels of interpolation.

Table C.2 presents the contour line of the electromagnetic field depending on the degree of interpolation.

Table C.1: Number of unknowns of the EM and the $\mathbf{\Lambda}$ fields for an unstructured tetrahedral mesh with $1,645,874$ elements and $3,521,251$ faces.

HDG method	# DoF $\mathbf{\Lambda}$ field	# DoF EM field
HDG-$\mathbb{P}_1$	21,127,506	39,500,976
HDG-$\mathbb{P}_2$	42,255,012	98,752,440
HDG-$\mathbb{P}_3$	70,425,020	197,504,880

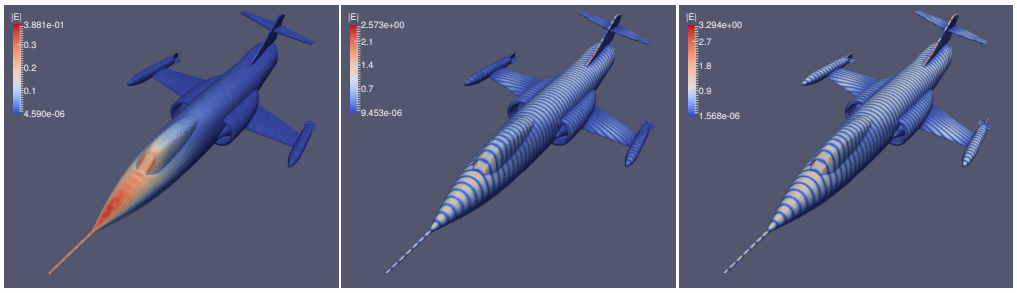


Table C.2: Contour line of $|\mathbf{E}|$ from HDG$-\mathbb{P}_1$ to HDG$-\mathbb{P}_3$ for an unstructured tetrahedral mesh with $1,645,874$ elements and $3,521,251$ faces.

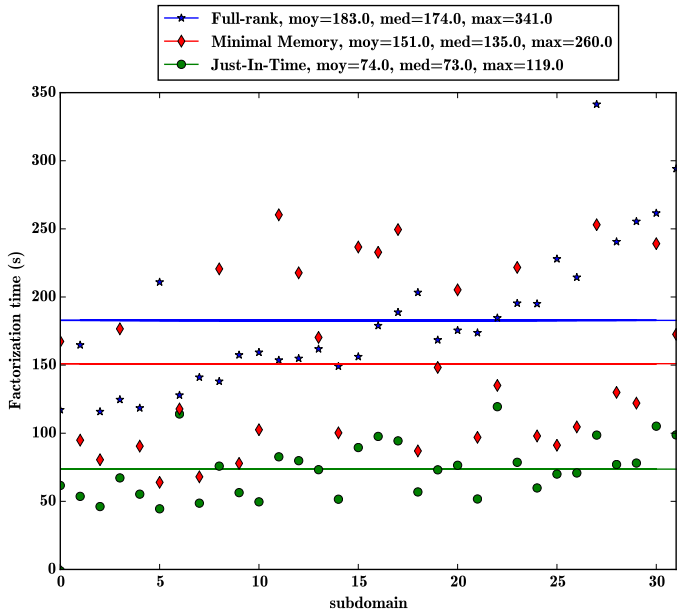
Experiments. We performed experiments for the mesh presented in Table C.1 with $\mathbb{P}_2$ interpolation. In this study, we consider only the $\mathbf{\Lambda}$ field, which leads to a sparse system with $42,255,012$ unknowns. This system is solved using a Schwarz additive domain decomposition solver, where a single factorization is performed on each subdomain. However, several solves can be performed for iterative refinement between subdomains, using BiCGSTAB.

We performed experiments on the Occigen[1] supercomputer, with 24-cores nodes with 128 GB. In next experiments, we use either 32, 48 or 64 subdomains. Each subdomain, corresponding to a MPI process, is hold on a socket with 12 cores. The full-rank version of PaStiX is studied, as well as both **Minimal Memory** and **Just-In-Time** strategies with 10^{-4} and 10^{-8} tolerances.

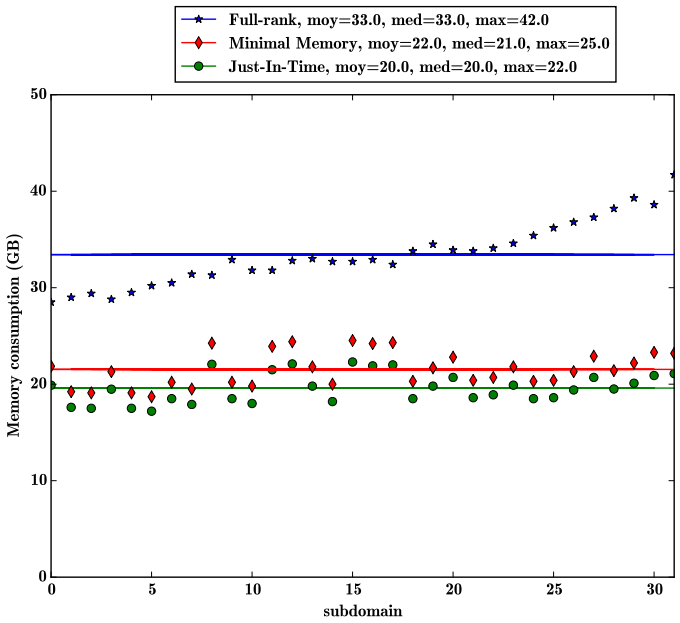
In Figure C.1(a) (respectively Figure C.1(b)), we present the factorization time (respectively the memory consumption) on each subdomain with a domain decomposition into 32 subdomains, using the full-rank factorization and both low-rank strategies with a 10^{-4} tolerance. Subdomains are ordered accordingly to the number of operations for the full-rank factorization, to illustrate the imbalance among subdomains.

One can note that for both factorization time and memory consumption, low-rank strategies reduce time-to-solution. For the factorization time, the average gain of the **Just-In-Time** strategy with respect to the full-rank factorization is of 2.5 and 1.6 for the **Minimal Memory** strategy. However, the real gain appears with respect to the subdomain which is the longer to be factorized, as all subdomains have

[1] https://www.cines.fr/calcul/materiels/occigen



(a) Factorization time.



(b) Memory consumption.

Figure C.1: Factorization time and memory consumption on each subdomain for the full-rank approach and both low-rank strategies with a 10^{-4} tolerance, for an unstructured tetrahedral mesh leading to a system with $42,255,012$ unknowns. The 32 subdomains are sorted by increasing number of operations.

to be factorized before next operations in the HORSE solver. For both methods, it still enhances the factorization step, since low-rank strategies reduce the maximum factorization time. In terms of memory consumption, low-rank strategies are better of a factor of 1.6 in average with respect to the full-rank factorization. If it is slightly in advantage of the **Just-In-Time** strategy, we recall that this approach only reduces the final size of the factors and not the memory peak achieved during the factorization. Thus, it can enhance the solver (meaning allow to solve larger problems), if and only if the memory peak is not achieved during the factorization of subdomains, but somewhere else in HORSE, when the factors are still used.

However, if subdomains are not solved with a direct solver, but with low-rank strategies that provide an approximation of the solution, more iterative refinement steps may be performed. In Table C.3, we present detailed statistics to analyze the effects of using low-rank compression for the full application. We first present the factorization time, which corresponds to the time to factorize all subdomains. Then, the number of iterations to compute a suitable solution is shown, together with this refinement cost. Those factorization and refinement steps are included in the total time to solve the system. Finally, the memory consumption, corresponding to the maximum consumption (for the factors only) achieved on one subdomain is shown.

The **Just-In-Time** always outperforms the full-rank factorization, while the **Minimal Memory** strategy is only better for a 10^{-4} tolerance. Low-rank strategies may increase the number of refinement steps, especially when using a 10^{-4} tolerance. However, the cost of the refinement is not necessarily higher, since low-rank solves used in the refinement are faster than full-rank solves. For memory consumption, **Minimal Memory** strategy is the only method that allows to reduce the memory peak achieved during the factorization of subdomains. As said before, **Just-In-Time** strategy can also enhance the solver for memory consumption if the peak is achieved when using the factors, but not during the factorization. For the global behavior, **Just-In-Time** strategy reduces time-to-solution, especially when using a 10^{-4} tolerance. **Minimal Memory** strategy computes the solution in the same order of time than the full-rank factorization, but can reduce the memory consumption of the solver, allowing solving larger systems.

Discussion: limitations for domain decomposition solvers. In order to enhance the factorization of all subdomains in a domain decomposition approach, one could use a single instance of the PASTIX solver to perform all factorizations. Thus, it should allow better load balancing, as more computational resources can be feed until all factorizations are done. It can be performed using runtime systems, to initially factorize each subdomain on a set of workers and to perform work stealing dynamically for balancing computations. Thus, one can expect that the behavior of the full-rank and both low-rank factorizations will be enhanced. It can be particularly interesting for low-rank strategies, since it is impossible to predict the ranks and thus the number of operations before any numerical operation.

Subs	τ	Method	Fact(s)	Nb. of iters	Refinement (s)	HDGM (s)	Memory (GB)
32	-	Full-rank	526.0	9	254.5	814.0	41.7
	1e-4	Just-In-Time	209.3	9	164.8	**405.8**	41.7 (22.3)
		Minimal-Memory	516.7	15	273.2	822.0	**24.5**
	1e-8	Just-In-Time	325.2	9	192.9	550.4	41.7 (29.4)
		Minimal-Memory	600.1	9	193.2	825.8	30.5
48	-	Full-rank	237.3	8	118.6	374.6	24.9
	1e-4	Just-In-Time	112.2	9	106.1	**237.1**	24.9 (14.1)
		Minimal-Memory	229.3	13	159.7	407.5	**15.3**
	1e-8	Just-In-Time	171.5	8	105.8	296.1	24.9 (18.1)
		Minimal-Memory	319.4	8	109.8	447.9	18.8
64	-	Full-rank	179.8	9	104.7	298.3	17.4
	1e-4	Just-In-Time	79.7	10	91.1	**184.1**	17.4 (10.0)
		Minimal-Memory	138.1	13	120.7	272.2	**11.0**
	1e-8	Just-In-Time	124.6	9	90.0	228.2	17.4 (12.9)
		Minimal-Memory	239.2	9	91.5	344.5	13.7

Table C.3: Statistics for Horse on an unstructured tetrahedral mesh leading to a system with $42,255,012$ unknowns with either 32, 48 or 64 subdomains, using full-rank factorization or both low-rank strategies. The factorization time on subdomains is illustrated, as well as the refinement cost and the total time to solve the problem. The number of iterations of BiCGSTAB and the memory consumption are also presented.

References

[1] S. Abdulah, H. Ltaief, Y. Sun, M. G. Genton, and D. E. Keyes. Tile low-rank approximation of large-scale maximum likelihood estimation on manycore architectures. *arXiv preprint arXiv:1804.09137*, 2018. [Cited on page 25]

[2] E. Agullo, P. R. Amestoy, A. Buttari, A. Guermouche, J-Y. L'Excellent, and F-H. Rouet. Robust memory-aware mappings for parallel multifrontal factorizations. *SIAM Journal on Scientific Computing*, 38(3):C256–C279, 2016. [Cited on page 46]

[3] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. Thibault. Achieving high performance on supercomputers with a sequential task-based programming model. *IEEE Transactions on Parallel and Distributed Systems*, 2017. [Cited on page 20]

[4] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems. *ACM Transactions on Mathematical Software (TOMS)*, 43(2):13, 2016. [Cited on pages 30 and 117]

[5] E. Agullo, L. Giraud, A. Guermouche, A. Haidar, and J. Roman. Parallel algebraic domain decomposition solver for the solution of augmented systems. *Advances in Engineering Software*, 60(Supplement C):23 – 30, 2013. [Cited on page 6]

[6] K. Akbudak, H. Ltaief, A. Mikhalev, and D. E. Keyes. *Tile Low Rank Cholesky Factorization for Climate/Weather Modeling Applications on Manycore Architectures*, pages 22–40. Springer International Publishing, Cham, 2017. [Cited on page 25]

[7] J. I. Aliaga, R. Carratalá-Sáez, R.R Kriemann, and E. S. Quintana-Ortí. Task-parallel lu factorization of hierarchical matrices using ompss. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*, pages 1148–1157. IEEE, 2017. [Cited on page 24]

[8] P. R. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J-Y. L'Excellent, and C. Weisbecker. Improving Multifrontal Methods by Means of Block Low-Rank Representations. *SIAM Journal on Scientific Computing*, 37(3):A1451–A1474, 2015. [Cited on pages 16, 25, and 114]

[9] P. R. Amestoy, A. Buttari, I. S. Duff, A. Guermouche, J.Y. L'Excellent, and B. Uçar. MUMPS. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 1232–1238. Springer, 2011. [Cited on pages 25 and 78]

[10] P. R.. Amestoy, A. Buttari, J-Y. L'Excellent, and T. Mary. On the complexity of the block low-rank multifrontal factorization. *SIAM Journal on Scientific Computing*, 39(4):A1710–A1740, 2017. [Cited on pages 16 and 55]

[11] P. R. Amestoy, A. Buttari, J-Y. L'Excellent, and T. Mary. Bridging the gap between flat and hierarchical low-rank matrix formats: the multilevel BLR format. Research report, University of Manchester, April 2018. [Cited on page 25]

[12] P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, 1996. [Cited on page 8]

[13] A. Aminfar, S. Ambikasaran, and E. Darve. A fast block low-rank dense solver with applications to finite-element matrices. *Journal of Computational Physics*, 304:170–188, 2016. [Cited on pages 16, 18, 24, and 25]

[14] A. Aminfar and E. Darve. A fast, memory efficient and robust sparse preconditioner based on a multifrontal approach with applications to finite-element matrices. *International Journal for Numerical Methods in Engineering*, 107(6):520–540, 2016. [Cited on page 24]

[15] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, et al. LAPACK Users' Guide. SIAM, Philadelphia, PA, 1992, 2007. [Cited on page 20]

[16] J. Anton, C. Ashcraft, and C. Weisbecker. A Block Low-Rank Multithreaded Factorization for Dense BEM Operators. In *SIAM Conference on Parallel Processing for Scientific Computing (SIAM PP 2016)*, Paris, France, April 2016. [Cited on pages 25, 37, 40, and 55]

[17] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook. *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*. Princeton University Press, Princeton, NJ, USA, 2007. [Cited on page 64]

[18] C. Ashcraft and I. Duff. Maxflow, min-cuts and multisectors of graphs. In *CSC14: The Sixth SIAM Workshop on Combinatorial Scientific Computing*, page 17, 2014. [Cited on page 108]

[19] C. Augonnet, S. Thibault, R. Namyst, and P-A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011. [Cited on pages 20 and 117]

[20] R. Battiti and A. Bertossi. Differential greedy for the 0-1 equicut problem. In *in Proceedings of the DIMACS Workshop on Network Design: Connectivity and Facilities Location*, pages 3–21. American Mathematical Society, 1997. [Cited on page 100]

[21] M. Bebendorf. *Hierarchical matrices*. Springer, 2008. [Cited on page 27]

[22] J. T. Betts and W. P. Huffman. Mesh refinement in direct transcription methods for optimal control. *Optimal Control Applications and Methods*, 19(1):1–21, 1998. [Cited on page 100]

[23] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J.W. Demmel, I. Dhillon, J.J. Dongarra, S. Hammarling, G. Henry, A. Petitet, et al. *ScaLAPACK users' guide*. SIAM, 1997. [Cited on page 20]

[24] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, et al. Flexible development of dense linear algebra algorithms on massively parallel architectures with dplasma. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1432–1441. IEEE, 2011. [Cited on pages 20 and 117]

[25] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013. [Cited on pages 20, 113, and 117]

[26] A. E. Caldwell, A. B. Kahng, A. A. Kennings, and I. L. Markov. Hypergraph partitioning for VLSI CAD: methodology for heuristic development, experimentation and reporting. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 349–354. ACM, 1999. [Cited on page 100]

[27] A. Casadei, P. Ramet, and J. Roman. An improved recursive graph bipartitioning algorithm for well balanced domain decomposition. In *21st IEEE International Conference on High Performance Computing (HiPC)*, pages 1–10, Goa, India, December 2014. [Cited on page 101]

[28] J. N. Chadwick and D. S. Bindel. An Efficient Solver for Sparse Linear Systems Based on Rank-Structured Cholesky Factorization. *CoRR*, abs/1507.05593, 2015. [Cited on pages 24, 25, and 80]

[29] P. Charrier and J. Roman. Algorithmic study and complexity bounds for a nested dissection solver. *Numerische Mathematik*, 55(4):463–476, July 1989. [Cited on pages 10, 13, 67, and 68]

[30] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam. Algorithm 887: Cholmod, supernodal sparse cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software (TOMS)*, 35(3):22, 2008. [Cited on page 24]

[31] N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, DTIC Document, 1976. [Cited on page 64]

[32] P. Ciarlet and F. Lamour. On the validity of a front-oriented approach to partitioning large sparse graphs with a connectivity constraint. *Numerical Algorithms*, 12(1):193–214, 1996. [Cited on page 100]

[33] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011. [Cited on page 114]

[34] T. A. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar. A survey of direct methods for sparse linear systems. *Acta Numerica*, 25:383–566, 2016. [Cited on page 6]

[35] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, 1990. [Cited on page 10]

[36] J. Duersch and M. Gu. Randomized qr with column pivoting. *SIAM Journal on Scientific Computing*, 39(4):C263–C291, 2017. [Cited on page 18]

[37] I. S. Duff, A. M. Erisman, and J. K. Reid. Direct methods for sparse matrices. *Oxford University Press*, London 1986. [Cited on page 6]

[38] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear. *ACM Transactions on Mathematical Software (TOMS)*, 9(3):302–325, 1983. [Cited on page 10]

[39] C. Eckart and G. Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218, 1936. [Cited on pages 14 and 17]

[40] M. Faverge. *Ordonnancement hybride statique-dynamique en algèbre linéaire creuse pour de grands clusters de machines NUMA et multi-coeurs*. PhD thesis, LaBRI, Université Bordeaux, Talence, France, December 2009. [Cited on page 20]

[41] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th design automation conference*, pages 175–181. IEEE Press, 1982. [Cited on page 100]

[42] J. Gaidamour and P. Hénon. A parallel direct/iterative solver based on a Schur complement approach. In *IEEE 11th International Conference on Computational Science and Engineering*, pages 98–105, Sao Paulo, Brésil, July 2008. [Cited on page 6]

[43] A. George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973. [Cited on pages 8 and 101]

[44] A. George, M. T. Heath, J. W. H. Liu, and E. G. Ng. Sparse Cholesky factorization on a local memory multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 9:327–340, 1988. [Cited on page 6]

[45] A. George and J. W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981. [Cited on pages 6 and 59]

[46] A. George and D. R. McIntyre. On the application of the minimum degree algorithm to finite element systems. *SIAM Journal on Numerical Analysis*, 15(1):90–112, 1978. [Cited on page 26]

[47] P. Ghysels, X. S. Li, C Gorman, and F-H. Rouet. A robust parallel preconditioner for indefinite systems using hierarchical matrices and randomized sampling. In *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*, pages 897–906, 2017. [Cited on page 24]

[48] P. Ghysels, X.S. Li, F-H. Rouet, S. Williams, and A. 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. [Cited on page 24]

[49] L. Giraud, A. Haidar, and Y. Saad. Sparse approximations of the Schur complement for parallel algebraic hybrid linear solvers in 3D. Rapport de recherche RR-7237, INRIA, March 2010. [Cited on page 6]

[50] L. Giraud and J. Langou. A Robust Criterion for the Modified Gram-Schmidt Algorithm with Selective Reorthogonalization. *SIAM Journal on Scientific Computing*, 25(2):417–441, 2003. [Cited on page 39]

[51] L. Grasedyck, W. Hackbusch, and R. Kriemann. Performance of H-LU preconditioning for sparse matrices. *Computational methods in applied mathematics*, 8(4):336–349, 2008. [Cited on page 24]

[52] L. Grasedyck, R. Kriemann, and S. Le Borne. Parallel black box $\mathcal{H}$-LU preconditioning for elliptic boundary value problems. *Computing and Visualization in Science*, 11(4-6):273–291, 2008. [Cited on page 24]

[53] L. Grasedyck, R. Kriemann, and S. Le Borne. Domain decomposition based $\mathcal{H}-lu$ preconditioning. *Numerische Mathematik*, 112(4):565–600, 2009. [Cited on page 23]

[54] W. Hackbusch. A Sparse Matrix Arithmetic Based on $\mathcal{H}$-Matrices. Part I: Introduction to $\mathcal{H}$-Matrices. *Computing*, 62(2):89–108, 1999. [Cited on pages 16 and 23]

[55] W. Hackbusch. *Hierarchical Matrices: Algorithms and Analysis*, volume 49. Springer Series in Computational Mathematics, 2015. [Cited on pages 24, 25, and 55]

[56] W. Hackbusch and S. Börm. Data-sparse Approximation by Adaptive $\mathcal{H}^2$-Matrices. *Computing*, 69(1):1–35, 2002. [Cited on pages 16 and 25]

[57] N. Halko, P-G. Martinsson, and A. J. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, 53(2):217–288, 2011. [Cited on page 18]

[58] R. W. Hamming. Error Detecting and Error Correcting Codes. *Bell System Technical Journal*, 26(2):147–160, 1950. [Cited on page 63]

[59] 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. [Cited on pages 10 and 19]

[60] P. Hénon, P. Ramet, and J. Roman. On finding approximate supernodes for an efficient block-ILU (k) factorization. *Parallel Computing*, 34(6-8):345–362, 2008. [Cited on pages 6 and 13]

[61] K. L. Ho and L. Ying. Hierarchical Interpolative Factorization for Elliptic Operators: Differential Equations. *Communications on Pure and Applied Mathematics*, 8(69):1415–1451, 2016. [Cited on page 25]

[62] T. Hofmann, B. Schölkopf, and A. J. Smola. Kernel methods in machine learning. *The annals of statistics*, pages 1171–1220, 2008. [Cited on page 27]

[63] K. D. Hogg, J. K. Reid, and J. A. Scott. Design of a multicore sparse Cholesky factorization using dags. *SIAM Journal on Scientific Computing*, 32(6):3627–3649, 2010. [Cited on page 26]

[64] G. W. Howell, J. W. Demmel, C. T. Fulton, S. Hammarling, and K. Marmol. Cache efficient bidiagonalization using blas 2.5 operators. *ACM Trans. Math. Softw.*, 34(3):14:1–14:33, May 2008. [Cited on page 51]

[65] A. Ida. Lattice h-matrices on distributed-memory systems. In *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018*, May 2018. [Cited on page 25]

[66] A. Ida, H. Nakashima, and M. Kawai. Parallel hierarchical matrices with block low-rank representation on distributed memory computer systems. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, pages 232–240. ACM, 2018. [Cited on page 25]

[67] M. Jacquelin, E. G. Ng, and W. B. Peyton. Fast and effective reordering of columns within supernodes using partition refinement. In *2018 Proceedings of the Seventh SIAM Workshop on Combinatorial Scientific Computing*, pages 76–86. SIAM, 2018. [Cited on page 78]

[68] S. Jain, C. Swamy, and K. Balaji. Greedy algorithms for k-way graph partitioning. In *the 6th international conference on advanced computing*, 1998. [Cited on page 100]

[69] D. S. Johnson and L. A. Mcgeoch. *The Traveling Salesman Problem: A Case Study in Local Optimization*, volume 1, pages 215–310. Princeton University Press, 1997. [Cited on page 64]

[70] G. Karypis and V. Kumar. METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices, 1995. [Cited on page 9]

[71] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998. [Cited on page 100]

[72] R. Kriemann. H-LU factorization on many-core systems. *Computing and Visualization in Science*, 16(3):105–117, 2013. [Cited on page 24]

[73] J. Kurzak, P. Luszczek, A. Yarkhan, M. Faverge, J. Langou, H. Bouwmeester, and J. J. Dongarra. Multithreading in the PLASMA Library. In Sanguthevar Rajasekaran Mohamed Ahmed, Reda A. Ammar, editor, *"Handbook of Multi and Many-Core Processing: Architecture, Algorithms, Programming, and Applications*. Chapman and Hall/CRC, March 2014. [Cited on pages 20 and 117]

[74] X. Lacoste. *Scheduling and memory optimizations for sparse direct solver on multi-core/multi-gpu cluster systems*. PhD thesis, Bordeaux University, Talence, France, February 2015. [Cited on pages 27, 30, 57, 81, 111, 113, and 117]

[75] X. Lacoste, C. Augonnet, and D. Goudin. Designing An Efficient and Scalable Block Low-Rank Direct Solver for Large Scale Clusters. In *SIAM Conference on Parallel Processing for Scientific Computing (SIAM PP 2016)*, Paris, France, April 2016. [Cited on page 25]

[76] X. Lacoste, M. Faverge, P. Ramet, S. Thibault, and G. Bosilca. Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes. In *HCW'2014 workshop of IPDPS*, Phoenix, AZ, May 2014. IEEE. [Cited on pages 20, 113, and 117]

[77] R. Leland and B. Hendrickson. A multilevel algorithm for partitioning graphs. In *1995 ACM/IEEE conference on Supercomputing*, 1995. [Cited on page 100]

[78] L. Li, S. Lanteri, and R. Perrussel. A hybridizable discontinuous galerkin method combined to a schwarz algorithm for the solution of 3d time-harmonic maxwell's equation. *Journal of Computational Physics*, 256:563–581, 2014. [Cited on page 127]

[79] X. S.. Li and J. W. Demmel. SuperLU_DIST: A Scalable Distributed-memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM Trans. Math. Softw.*, 29(2):110–140, June 2003. [Cited on page 19]

[80] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM journal on numerical analysis*, 16(2):346–358, 1979. [Cited on pages 13 and 101]

[81] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36:177–189, 1979. [Cited on pages 56 and 67]

[82] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11(1):134–172, 1990. [Cited on page 10]

[83] J. W. H. Liu, E. G. Ng, and B. W. Peyton. On finding supernodes for sparse matrix computations. *SIAM Journal on Matrix Analysis and Applications*, 14(1):242–252, 1993. [Cited on page 10]

[84] B. Lizé. *Résolution directe rapide pour les éléments finis de frontière en électromagnétisme et acoustique : H-matrices. parallélisme et applications industrielles.* PhD thesis, École Doctorale Galilée, June 2014. [Cited on page 24]

[85] R. Luce and E. G. Ng. On the minimum flops problem in the sparse Cholesky factorization. *SIAM Journal on Matrix Analysis and Applications*, 35(1):1–21, 2014. [Cited on page 8]

[86] P-G. Martinsson. A fast randomized algorithm for computing a hierarchically semiseparable representation of a matrix. *SIAM Journal on Matrix Analysis and Applications*, 32(4):1251–1274, 2011. [Cited on page 24]

[87] P-G. Martinsson. Blocked rank-revealing qr factorizations: How randomized sampling can be used to avoid single-vector pivoting. *arXiv preprint arXiv:1505.08115*, 2015. [Cited on page 18]

[88] T. Mary. *Block Low-Rank multifrontal solvers: complexity, performance, and scalability.* PhD thesis, Toulouse University, Toulouse, France, November 2017. [Cited on pages 25, 55, and 56]

[89] G. L. Miller, S.-H. Teng, and S. A. Vavasis. A unified geometric approach to graph separators. In *Proc. 31st Annual Symposium on Foundations of Computer Science*, pages 538–547, 1991. [Cited on pages 13 and 67]

[90] G. L. Miller and S. A. Vavasis. Density graphs and separators. In *Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 331–336, 1991. [Cited on pages 13, 56, and 67]

[91] University of Waterloo. Concorde TSP solver. `http://www.math.uwaterloo.ca/tsp/concorde.html`. [Cited on page 65]

[92] F. Pellegrini. Scotch and libScotch 5.1 User's Guide, August 2008. User's manual, 127 pages. [Cited on pages 9 and 100]

[93] A. Pothen and C-J. Fan. Computing the block triangular form of a sparse matrix. *ACM Transactions on Mathematical Software (TOMS)*, 16(4):303–324, 1990. [Cited on page 104]

[94] H. Pouransari, P. Coulier, and E. Darve. Fast hierarchical solvers for sparse matrices using extended sparsification and low-rank approximation. *SIAM Journal on Scientific Computing*, 39(3):A797–A830, 2017. [Cited on page 25]

[95] M. Predari. *Load Balancing for Parallel Coupled Simulations*. PhD thesis, Université de Bordeaux, LaBRI ; Inria Bordeaux Sud-Ouest, December 2016. [Cited on page 108]

[96] M. Predari, A. Esnard, and J. Roman. Comparison of initial partitioning methods for multilevel direct k-way graph partitioning with fixed vertices. *Parallel Computing*, 2017. [Cited on pages 100, 101, and 105]

[97] S. Rajamanickam, E.G. Boman, and M.A. Heroux. Shy*LU*: A hybrid-hybrid solver for multicore platforms. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 631–643, May 2012. [Cited on page 6]

[98] E. Rebrova, G. Chavez, Y. Liu, P. Ghysels, and X. S. Li. A study of clustering techniques and hierarchical matrix formats for kernel ridge regression. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2018, Vancouver, BC, Canada, May 21-25, 2018*, pages 883–892, 2018. [Cited on page 27]

[99] S. Rjasanow. Adaptive cross approximation of dense matrices. In *Int. Association Boundary Element Methods Conf., IABEM*, pages 28–30, 2002. [Cited on page 18]

[100] D. J. Rose and R. E. Tarjan. Algorithmic aspects of vertex elimination in directed graphs. *SIAM Journal on Applied Mathematics*, 34(1):176–197, 1978. [Cited on page 8]

[101] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis II. An analysis of several heuristics for the traveling salesman problem. *SIAM J. Comput.*, 6(3):563–581, 1977. [Cited on page 65]

[102] Y. Saad. *Iterative methods for sparse linear systems*, volume 82. siam, 2003. [Cited on page 6]

[103] E. Schmidt. Über die auflösung linearer gleichungen mit unendlich vielen unbekannten. *Rendiconti del Circolo Matematico di Palermo (1884-1940)*, 25(1):53–77, 1908. [Cited on page 39]

[104] Science and Technology Facilites Council. The HSL Mathematical Software Library. A collection of Fortran codes for large scale scientific computation. http://www.hsl.rl.ac.uk/. [Cited on page 26]

[105] M. Sergent, D. Goudin, S. Thibault, and O. 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, IL, May 2016. [Cited on pages 20 and 57]

[106] W. M. Sid-Lakhdar. *Scaling the solution of large sparse linear systems using multifrontal methods on hybrid shared-distributed memory architectures*. PhD thesis, École normale supérieure de lyon - ENS LYON, December 2014. [Cited on pages 26 and 78]

[107] D. A. Sushnikova and I. V. Oseledets. "Compress and eliminate" solver for symmetric positive definite sparse matrices. *arXiv preprint arXiv:1603.09133v3*, 2016. [Cited on page 25]

[108] W. F. Tinney and J. W. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proceedings of the IEEE*, 55(11):1801–1809, Nov. 1967. [Cited on page 8]

[109] S. Wang, X. S. Li, F-H. Rouet, J. Xia, and M. V. De Hoop. A Parallel Geometric Multifrontal Solver Using Hierarchically Semis-Separable Structure. *ACM Trans. Math. Softw.*, 42(3):21, 2016. [Cited on page 24]

[110] C. Weisbecker. *Improving multifrontal solvers by means of algebraic block low-rank representations*. PhD thesis, Institut National Polytechnique de Toulouse-INPT, 2013. [Cited on page 88]

[111] J. L. Xia. Randomized sparse direct solvers. *SIAM Journal on Matrix Analysis and Applications*, 34(1):197–227, 2013. [Cited on page 24]

[112] J. L. Xia, S. Chandrasekaran, M. Gu, and XS Li. Superfast Multifrontal Method For Large Structured Linear Systems of Equations. *SIAM Journal on Matrix Analysis and Applications*, 31:1382–1411, 2009. [Cited on pages 16, 24, 25, and 56]

[113] J. Xiao, M. Gu, and J. Langou. Fast parallel randomized qr with column pivoting algorithms for reliable low-rank matrix approximations. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 233–242, Dec 2017. [Cited on pages 17 and 18]

[114] I. Yamazaki and X. S. Li. On techniques to improve robustness and scalability of a parallel hybrid linear solver. In *International Conference on High Performance Computing for Computational Science*, pages 421–434. Springer, 2010. [Cited on page 6]

[115] K. Yang, H. Pouransari, and E. Darve. Sparse Hierarchical Solvers with Guaranteed Convergence. *arXiv preprint arXiv:1611.03189*, 2016. [Cited on page 25]

[116] C. D. Yu, J. Levitt, S. Reiz, and G. Biros. Geometry-oblivious fmm for compressing dense spd matrices. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 53. ACM, 2017. [Cited on page 27]

Publications

International journals

[117] G. Pichon, E. Darve, M. Faverge, P. Ramet, and J. Roman. Sparse supernodal solver using block low-rank compression: Design, performance and analysis. *International Journal of Computational Science and Engineering*, 27:255 – 270, July 2018.

[118] G. Pichon, M. Faverge, P. Ramet, and J. Roman. Reordering strategy for blocking optimization in sparse linear solvers. *SIAM Journal on Matrix Analysis and Applications*, 38(1):226–248, 2017.

International conferences with proceedings

[119] G. Pichon, E. Darve, M. Faverge, P. Ramet, and J. Roman. Sparse Supernodal Solver Using Block Low-Rank Compression. In *PDSEC'2017 workshop of IPDPS*, Orlando, United States, May 2017.

[120] G. Pichon, A. Haidar, M. Faverge, and J. Kurzak. Divide and Conquer Symmetric Tridiagonal Eigensolver for Multicore Architectures. In *29th IEEE International Parallel & Distributed Processing Symposium*, Hyderabad, India, May 2015.

French conferences with proceedings

[121] G. Pichon. Utilisation de la compression Block Low-Rank pour accélérer un solveur direct creux supernodal. In *Conférence d'informatique en Parallélisme, Architecture et Système (ComPAS'17)*, Sophia Antipolis, France, June 2017.

International conferences without proceedings

[122] M. Faverge, G. Pichon, and P. Ramet. Exploiting Kepler architecture in sparse direct solver with runtime systems. In *9th International Workshop on Parallel Matrix Algorithms and Applications (PMAA'2016)*, Bordeaux, France, July 2016.

[123] M. Faverge, G. Pichon, P. Ramet, and J. Roman. Blocking strategy optimizations for sparse direct linear solver on heterogeneous architectures. In *Sparse Days*, Saint Girons, France, June 2015.

[124] M. Faverge, G. Pichon, P. Ramet, and J. Roman. On the use of H-Matrix Arithmetic in PaStiX: a Preliminary Study. In *Workshop on Fast Solvers*, Toulouse, France, June 2015.

[125] G. Pichon, E. Darve, M. Faverge, P. Ramet, and J. Roman. Exploiting H-Matrices in Sparse Direct Solvers. In *SIAM Conference on Parallel Processing for Scientific Computing*, Paris, France, April 2016.

[126] G. Pichon, E. Darve, M. Faverge, P. Ramet, and J. Roman. On the use of low rank approximations for sparse direct solvers. In *SIAM Annual Meeting (AN'16)*, Boston, United States, July 2016.

[127] G. Pichon, E. Darve, M. Faverge, P. Ramet, and J. Roman. Sparse Supernodal Solver Using Hierarchical Compression. In *Workshop on Fast Direct Solvers*, Purdue, United States, November 2016.

[128] G. Pichon, E. Darve, M. Faverge, P. Ramet, and J. Roman. Sparse Supernodal Solver exploiting Low-Rankness Property. In *Sparse Days 2017*, Toulouse, France, September 2017.

[129] G. Pichon, E. Darve, M. Faverge, P. Ramet, and J. Roman. Sparse Supernodal Solver Using Hierarchical Compression over Runtime System. In *SIAM Conference on Computation Science and Engineering (CSE'17)*, Atlanta, United States, February 2017.

[130] G. Pichon, M. Faverge, and P. Ramet. Exploiting Modern Manycore Architecture in Sparse Direct Solver with Runtime Systems. In *SIAM Conference on Computation Science and Engineering (CSE'17)*, Atlanta, United States, February 2017.

[131] G. Pichon, M. Faverge, P. Ramet, and J. Roman. Impact of blocking strategies for sparse direct solvers on top of generic runtimes. In *SIAM Conference on Parallel Processing for Scientific Computing*, Paris, France, April 2016.

[132] G. Pichon, M. Faverge, P. Ramet, and J. Roman. Impact of Blocking Strategies for Sparse Direct Solvers on Top of Generic Runtimes. In *SIAM Conference on Computation Science and Engineering (CSE'17)*, Atlanta, United States, February 2017.

French conferences without proceedings

[133] G. Pichon, E. Darve, M. Faverge, S. Lanteri, P. Ramet, and J. Roman. Sparse supernodal solver with low-rank compression for solving the frequency-domain Maxwell equations discretized by a high order HDG method. Journées jeunes chercheur-e-s - Résolution de problèmes d'ondes harmoniques de grande taille, November 2017.