Improving the memory and time overhead of low-rank parallel linear sparse direct solvers

Esragul Korkmaz

LaBRI

université de BORDEAUX

THÈSE PRÉSENTÉE

POUR OBTENIR LE GRADE DE

DOCTEUR
DE L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

SPÉCIALITÉ: INFORMATIQUE

Par **Esragul KORKMAZ**

Improving the memory and time overhead of low-rank parallel linear sparse direct solvers

Sous la direction de : **Pierre RAMET**
Co-directeur : **Mathieu FAVERGE**

Soutenue le 21 Septembre 2022

Membres du jury :

M. Alfredo Buttari	Directeur de recherche	IRIT, CNRS	Rapporteur
M. Mathieu Faverge	Maître de conférence	Inria, Bordeaux INP, CNRS, UB	Invité
Mme. Laura Grigori	Directrice de recherche	Inria Paris, LJLL	Membre du Jury
M. Hatem Ltaief	Senior research scientist	KAUST	Rapporteur
M. François Pellegrini	Professeur	Labri, UB	Président
M. Gregoire Pichon	Maître de conférence	Université Claude Bernard Lyon 1	Membre du Jury
M. Pierre Ramet	Maître de conférence	Inria, Bordeaux INP, CNRS, UB	Directeur de thèse

Amélioration de la mémoire et du temps des solveurs directs creux linéaires de rang faible

Résumé :

Grâce aux récentes améliorations apportées par les nouveaux supercalculateurs exaflopiques, des simulations de grande envergure peuvent être effectuées dans des délais raisonnables en utilisant des opérations massivement parallélisées. Malheureusement, l'augmentation du nombre d'unités de calcul dans ces systèmes n'est pas accompagnée d'une augmentation de la mémoire disponible par cœur. Par conséquent, cette limitation de la mémoire oblige les scientifiques et les ingénieurs à non seulement paralléliser efficacement les opérations, mais aussi à minimiser la mémoire utilisée.

De nombreuses applications scientifiques et d'ingénierie doivent résoudre de grands systèmes linéaires creux du type $Ax = b$. Bien que les méthodes directes soient les solutions les plus robustes numériquement, elles sont coûteuses en termes d'utilisation de la mémoire et de temps de résolution. À cet égard, les techniques de compression de rang faible ont été récemment introduites dans ces solveurs afin de réduire leur complexité et leur empreinte mémoire.

Dans ce travail, notre objectif est d'améliorer les représentations de rang faible par bloc dans le solveur direct supernodal creux PaStiX. Pour cela, nous comparons quelques méthodes de compression pour déterminer le noyau le plus rapide, qui conserve les données représentatives avec le plus petit rang possible. Nous nous concentrons sur l'amélioration du solveur supernodal en réduisant le nombre de recompressions lors des opérations de mise à jour. Tout d'abord, nous étudions les stratégies de renumérotation des séparateurs pour identifier les blocs peu compressibles impliqués dans ces mises à jour et réduire leurs occurrences. Ensuite, nous proposons une solution orthogonale pour prédire la compressibilité des blocs avant la factorisation numérique. Cette dernière approche s'appuie sur l'utilisation de la notion de niveau de remplissage pour une factorisation incomplète symbolique par blocs.

Grâce à ces optimisations, l'utilisation de la mémoire a été fortement réduite, par rapport aux solveurs à l'état de l'art, tout en améliorant le temps de résolution. Cette thèse est un premier pas nécessaire vers un solveur direct creux performant utilisant des schémas de compression hiérarchiques.

Mots-clés : solveur linéaire creux direct, compression à rang faible, numérotation

Improving the memory and time overhead of low-rank parallel linear sparse direct solvers

Abstract:

Through the recent improvements toward exascale supercomputer systems, huge computations can be performed in reasonable times by using massively parallelized operations. Unfortunately, the increase of the computational units in these systems does not lead to a rise in the memory available per core. Therefore, this memory limitation forces the scientists/engineers to not only efficiently parallelize the operations but also minimize the memory used.

Many scientific and engineering applications have to solve large sparse linear systems of the type $Ax = b$. Although the direct methods are the most robust solutions for these systems, they are costly in terms of their memory usage and time-to-solution. In this respect, the low-rank representations have been recently introduced into these solvers to reduce the time and memory footprint.

In this work, our goal is to improve the low-rank feature of the block low-rank (BLR) sparse supernodal direct solver PaStiX. For this purpose, we compare some compression methods to determine the fastest kernel, which keeps the representative data with the smallest rank possible. Then, we focus on improving supernodal solver by reducing the number of re-compression during the updates. Firstly, we study the separator reordering strategies to identify the poorly compressible blocks involved in these updates and reduce their occurrences. Secondly, we propose an orthogonal solution to predict the compressibility of the blocks before the numerical factorization. This last approach relies on the use of the level of fill of a symbolic block incomplete factorization.

Thanks to these optimizations, the memory usage has been reduced more effectively compared to the state of the art solvers while also improving the time to solution. This thesis is a requested first step toward a advanced sparse direct solver using hierarchical compression schemes.

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

Acknowledgement

Writing this thesis was a long journey with a lot of ups and downs for me. I would not succeed it without the help of many people. Firstly, I would like to thank to my director Pierre Ramet for always supporting me both in terms of my work and motivating me psychologically. Secondly, I want to thank to my co-director Mathieu Faverge for always being available for me. I learned a lot from him. Both my directors supported me a lot in terms of work and personal problems. I really feel lucky to work with them. Here, I also want to thank to Gregoire Pichon who is involved in all parts of my work as much as my directors. Anytime I have trouble understanding something, he was my savior :) Among all his responsibilities, he could always find a time to help me.

I want to express my gratitude to all my jury members. My discussions with my reviewer Alfredo Buttari on my thesis has contributed me a lot and it was very precious for me. I would like to thank to my second reviewer Hatem Ltaief for providing me very nice references and calming me down just before my defense. I thank to both of my jury members Laura Grigori and Francois Pellegrini for their interesting questions which let me to think more broad in my field.

I feel very lucky to meet/work with all the Inria people (especially the HiePACS members). They contributed me to improve myself both in the field and personally. I thank to all my colleagues Alena Shilova, Martina Iannacito, Romain Peressoni, Marek Felsoci, Mathieu Verite, Yanfei Xiang, Tony Delarue, Xunyi Zhao, Jean-Francois David, Mahmoud Shouman, Pierre Esterie and Alycia Lisito for all the nice conversations we had and always being supportive.

I really appreciate Romain Peressoni for always supporting me for all my problems and preparing the food for my defense day. In addition, Alena Shilova and Martina Iannacito deserve a special appreciation since they were always by my side whenever I feel stress. And last but not least, I thank to Emmanuel Agullo who taught me how to play fairly in babyfoot :)

I am grateful to have my brother Koray Korkmaz as my life-long best friend and supporter. At this point of my life, I am looking forward to continue my journey with Olivier Beaumont, Lionel Eyraud-Dubois and Laercio Lima Pilla who were always very warm and helping since I met them.

Contents

Résumé long

Grâce aux avancées récentes en calcul scientifique, nous pouvons désormais remplacer les expériences coûteuses et/ou dangereuses de la vie réelle par des simulations numériques. Les domaines dans lesquels la simulation est utilisée sont vastes et incluent, par exemple, la météorologie, la biologie, la fusion nucléaire ou l'aérodynamique. Bien qu'il soit encore possible d'optimiser ces codes de calcul, les simulations les plus récentes sont déjà capables de fournir des solutions étonnamment précises et rapides.

La simulation de problèmes physiques dans le domaine de la science computationnelle exige que l'espace réel soit discrétisé. Les équations mathématiques continues sont également discrétisées sur ces points d'échantillonnage du domaine entier. L'ensemble des équations qui en résulte peut être représenté comme une matrice. Dans de nombreuses applications, seules les interactions entre les points proches de la discrétisation sont prises en compte. En effet, les interactions entre les points éloignés sont souvent faibles, et les valeurs correspondantes dans la matrice sont donc négligeables. Par conséquent, la matrice résultante comprend de nombreux termes nuls et est appelée une matrice creuse. Plus le nombre de points discrétisés et la précision attendue sont importants, plus le coût de calcul de la solution du système sera élevé. Ainsi, les améliorations dans ce domaine visent à résoudre des systèmes de plus grande taille, tout en réduisant le temps de restitution, pour une précision fixée.

Grâce aux récentes améliorations apportées par les nouveaux supercalculateurs exaflopiques, des simulations de grande envergure peuvent être effectuées dans des délais raisonnables en utilisant des opérations massivement parallélisées. Malheureusement, l'augmentation du nombre d'unités de calcul dans ces systèmes n'est pas suivie par une augmentation de la mémoire disponible par cœur. Par conséquent, cette limitation de la mémoire oblige les scientifiques et les ingénieurs à non seulement paralléliser efficacement les opérations, mais aussi à minimiser la mémoire utilisée.

Lors de la résolution de grands systèmes linéaires creux, la structure de la matrice peut être exploitée pour réduire la consommation de mémoire et la complexité des calculs. Parmi toutes les approches permettant de résoudre des systèmes linéaires creux, les méthodes directes représentent les approches les plus robustes au détriment d'exigences élevées en matière de mémoire et de complexité calculatoire. Ces solveurs factorisent la matrice initiale en un produit de matrices triangulaires. Ces systèmes triangulaires résultants peuvent ensuite être résolus pour obtenir la solution du problème.

Selon le type d'applications, il existe plusieurs variantes de ces méthodes directes. Selon les caractéristiques de la matrice, ces méthodes peuvent être basées sur différents

types de factorisation comme LL^T, LDL^T, ou LU. Ici, L, U et D représentent respectivement une matrice triangulaire inférieure, une matrice triangulaire supérieure et une matrice diagonale. De plus, selon la manière dont les contributions (mises à jour) sont appliquées pendant la factorisation, on peut considérer une approche supernodale ou multifrontale. La première stratégie permet d'appliquer directement toutes les mises à jour et d'éviter les surcoûts mémoires pour leur stockage. La seconde utilise un stockage temporaire pour les contributions afin de pouvoir les appliquer plus tard pendant la factorisation.

Il existe de nombreuses études sur la réduction du coût des solveurs directs en considérant divers types d'approximation. Par exemple, on peut considérer des factorisations incomplètes. L'approche consiste alors à abandonner certaines contributions dans la matrice, soit en utilisant des critères numériques, soit en considérant la structure uniquement (avec la notion de niveaux de remplissage). Alternativement, l'utilisation de techniques de compression de rangs faibles permet généralement d'obtenir de meilleures approximations que la factorisation incomplète. Bien qu'ils aient été introduits plus récemment dans le cadre des solveurs creux, de nombreuses études ont déjà prouvé leur efficacité en termes de réduction du temps et de la mémoire pour une précision donnée.

Les techniques de compression de rang faible visent à regrouper les inconnues de la matrice et à approximer les blocs résultants en fonction d'un critère d'admissibilité. Certaines approches regroupent les inconnues de la matrice de manière hiérarchique comme les schémas H, HODLR, $\mathcal{H}^2$ ou HSS, tandis que d'autres adoptent un regroupement non hiérarchique comme l'approche "block low-rank" (BLR). Bien que les schémas hiérarchiques offrent de meilleures complexités asymptotiques, un regroupement non hiérarchique est plus simple à mettre en œuvre et plus flexible dans les environnements parallèles.

Dans cette thèse, nous avons réalisé toutes nos expériences dans le cadre parallèle du solveur PaStiX qui implémente l'approche non hiérarchique BLR. Comme dans la plupart des solveurs directs, la renumérotation des inconnues de la matrice est obtenue en utilisant la méthode de dissection emboîtée. Cette méthode permet de préserver une grande partie de la structure creuse lors de sa factorisation et permet d'appliquer les opérations en parallèle.

Dans le cadre des solveurs utilisant une méthode supernodale, le temps de résolution et l'utilisation de la mémoire sont dépendants du moment où les blocs sont compressés. Lors de travaux antérieurs, deux variantes du solveur PaStiX ont été implémentées : *Minimal Memory* et *Just-In-Time*. Dans la première variante, tous les blocs admissibles sont compressés avant de commencer la factorisation. Ainsi, l'utilisation de la mémoire du solveur est réduite de manière optimale. Cependant, l'application des contributions de rangs faibles est coûteuse et peut conduire le solveur à être plus lent que la version sans compression. Dans la seconde variante, les blocs sont compressés seulement après avoir reçu toutes leurs contributions. De cette façon, le temps de résolution est amélioré sensiblement, mais sans réduire de manière significative la contrainte mémoire. Il est alors nécessaire de trouver une solution intermédiaire entre les deux stratégies *Minimal Memory* et *Just-In-Time*.

Les contributions de cette thèse consistent à déterminer le noyau de compression le plus rapide pour obtenir l'approximation la plus efficace des blocs et à fournir une solution hybride entre les stratégies *Minimal Memory* et *Just-In-Time* existantes. Grâce à ces améliorations, nous visons à optimiser les besoins en temps et en mémoire des solveurs directs utilisant des approximations de rangs faibles.

Dans le chapitre 1, nous discutons du contexte et des notions nécessaires pour introduire les contributions de cette thèse. Nous commençons le chapitre en expliquant en détail le fonctionnement des solveurs directs creux. Ensuite, nous présentons la condition d'admissibilité des blocs et discutons des représentations hiérarchiques et non-hiérarchiques. Nous mettons l'accent sur l'utilisation du format BLR dans les solveurs creux. Ensuite, nous expliquons le problème de l'application des mises à jour pendant la factorisation dans le contexte d'une méthode supernodale, et plus spécifiquement dans le solveur PaStiX, avec les approches *Minimal Memory* et *Just-In-Time*. Nous concluons ce chapitre en discutant des approches développées dans d'autres solveurs et de notre positionnement.

Dans le chapitre 2, nous étudions tout d'abord quelques noyaux d'approximation (compression) de blocs afin de déterminer le plus efficace en termes de stockage et de temps de compression. Ainsi, la décomposition en valeurs singulières (SVD) permet d'atteindre le stockage optimal à une précision donnée. Cependant, elle est très coûteuse à utiliser en pratique. Nous nous concentrons donc sur des méthodes plus rapides qui fournissent des résultats proches de la SVD. Bien qu'il existe de nombreuses alternatives pour cet objectif, nous ne considérerons que ceux qui répondent à nos besoins et qui sont basés sur la factorisation QR. Ici, nous implémentons toutes ces méthodes de manière homogène pour une meilleure comparaison. En particulier, nous appliquons un critère d'arrêt pour déterminer numériquement la qualité de la compression, ce qui est appliqué de manière originale pour deux des méthodes étudiées dans la littérature. Cette compression basée sur un seuil numérique garantit ainsi une précision prédéfinie. Dans ce chapitre, nous nous intéressons à la compression de matrices dont la taille cible correspond aux tailles des blocs apparaissant dans notre solveur BLR. De plus, comme PaStiX utilise des noyaux séquentiels pour la compression, nous considérons un environnement également séquentiel. Les résultats numériques montrent que les noyaux de compression utilisant des techniques de projections aléatoires peuvent être compétitifs par rapport à la méthode SVD. En moyenne, toutes les méthodes ont fourni des résultats assez proches de la SVD. Comme la stabilité et la précision sont assurées par la factorisation QR et notre critère d'arrêt, nous déterminons les noyaux les plus appropriés en comparant les temps de compression. Nous avons observé qu'il est nécessaire d'adapter les noyaux de compression en fonction de la taille de la matrice. Ainsi, nous avons proposé d'utiliser la méthode QRCP pour les matrices de petite taille, tandis que la méthode TQRCP est plus appropriée pour les matrices plus grandes.

Dans le chapitre 3, nous avons amélioré une heuristique préexistante, qui réordonne

les inconnues de la matrice après le processus de dissection emboîtée afin d'améliorer la compressibilité et permet de réduire le nombre total de mises à jour dans lors de la factorisation. Grâce à ce réordonnancement, une identification des blocs peu compressibles est également effectuée. Dans ce chapitre, nous avons optimisé cette heuristique en utilisant une configuration déterminée empiriquement. Plus précisément, nous vérifions le besoin de compresser les blocs peu compressibles juste après l'application de toutes les mises à jour (comme avec la stratégie *Just-In-Time*). Ainsi, cette heuristique conduit à une solution intermédiaire entre les stratégies *Just-In-Time* et *Minimal Memory* puisque les blocs hautement compressibles restent compressés avant de commencer la factorisation (comme avec la stratégie *Minimal Memory*). La compressibilité par rapport à l'heuristique originale est donc augmentée sensiblement. Dans les expériences, nous avons pu mettre en évidence des gains en termes de flops et de temps par rapport à l'heuristique originale.

Dans le chapitre 4, nous avons étudié une autre heuristique pour identifier les blocs peu compressibles afin de tirer profit des stratégies *Just-In-Time* et *Minimal Memory* simultanément. Ici, nous utilisons la notion de niveaux de remplissage utilisée pour les factorisations incomplètes. Dans cette heuristique, nous attribuons une valeur de niveau à chaque bloc. On suppose alors que les blocs présentant des niveaux élevés sont fortement compressibles, tandis que les autres représentent des blocs à forte interaction et sont peu compressibles. Les expériences que nous avons menées sur un large ensemble de matrices issues d'applications réelles ont démontré que notre heuristique parvient à identifier efficacement les blocs les plus compressibles. La solution proposée s'exécute jusqu'à $5,2$ fois plus rapidement que *Minimal Memory* avec seulement une augmentation d'un facteur $1,38$ sur l'utilisation de la mémoire, en considérant la plus haute précision dans les environnements séquentiels et multithreads. Dans l'environnement multithread, elle est même jusqu'à $1,84$ fois plus rapide et surpasse la version originale dans la plupart des cas. Nous avons donc montré que, si le critère sur le niveau de remplissage est correctement réglé, notre heuristique améliore systématiquement les stratégies existantes. Elle améliore la factorisation numérique en termes de mémoire et de temps, et elle améliore la scalabilité pour les environnements parallèles. Nous avons observé qu'un critère utilisant les premiers niveaux de remplissage est suffisant pour obtenir un gain significatif, et qu'une tendance claire apparaît pour régler ce critère en fonction de la précision souhaitée.

Dans cette thèse, nous avons ainsi amélioré l'utilisation de la mémoire et réduit le temps de résolution pour l'implémentation "Bloc Low Rank" dans le contexte d'un solveur direct creux supernodal, et c'est une première étape nécessaire avant d'implémenter un schéma hiérarchique. Grâce à toutes ces améliorations, le solveur PaStiX peut ainsi être utilisé soit comme un solveur direct ou soit comme un préconditionneur, pour lequel il est possible de réduire la complexité de l'algorithme en jouant le critère de compression de la matrice.

Introduction

Thanks to the rapid advances in computational science, we can now avoid costly and/or dangerous real-life experiments by simulating them. The fields where simulation is used include but are not limited to meteorology, biology, nuclear fusion and aerodynamics. Despite the need of further improvements in computational science, the up-to-date simulations are already able to provide amazingly accurate and fast solutions.

Simulating real life problems in computational science requires the continuous domain to be discretized into points. Then the continuous mathematical equations are discretized at these sample points of the whole domain. The resulting set of equations can be represented into a matrix form, where the coefficients stand for the entries of the matrix. In many applications, like when discretizing a heat equation, only the interactions between close points of the discretization are taken into account. Indeed far point interactions are frequently weak, thus the corresponding values in the matrix are negligible. Therefore, the resulting matrix includes many zero terms and is called a sparse matrix. Then, the larger the number of discretized points and the accuracy expectations, the more expensive the cost of computing the solution of the system. Thus, the improvements in this field aim at solving larger systems in shorter times at a given accuracy.

Through the recent improvements toward exascale supercomputer systems, huge computations can be performed in reasonable times by using massively parallelized operations. Unfortunately, the increase of the computational units in these systems does not lead to a rise in the memory available per core. Therefore, this memory limitation forces the scientists/engineers to not only efficiently parallelize the operations but also minimize the memory footprint.

When solving large sparse systems, the existing zero blocks of the sparse matrix can be exploited to reduce the memory and computational complexities. Among all the solutions to solve sparse linear systems, direct solvers serve as the most robust approaches despite their expensive memory and computational requirements. These solvers factorize the matrix into a multiplication of triangular matrices. Then, the resulting triangular systems can be solved to obtain the solution.

Depending on the application requirements, direct methods can be performed differently. According to the matrix features, these methods can be based on different factorization types like LL^T, LDL^T, and LU, for example. Here, L, U, and D represent a lower triangular, an upper triangular, and a diagonal matrix, respectively. Additionally, depending on how the contributions (updates) are applied during the factorization, they can adopt a supernodal or a multifrontal approach. The former strategy allows to directly

apply all the updates and avoid using extra memory to store those updates. On the other hand, the latter one uses temporary storage for the contributions to be able to apply them later during the factorization. Thus, this approach leads to a larger memory usage compared to the supernodal version.

There are many studies on reducing the cost of direct solvers by approximating them. For instance, incomplete factorization is proposed for this purpose. The approach consists of dropping some data in the matrix, either by using the geometry of the structure (using the fill-in levels concept) or in a numerical way (using a threshold). Alternatively, low-rank compression schemes are offered as better approximation approaches than incomplete factorization. Although they were recently introduced, there are already many studies that have proven their efficiency in reducing time and memory at a given precision.

Low-rank schemes aim to cluster the matrix and approximate the resulting admissible blocks according to an admissibility criterion. Some approaches cluster the matrix unknowns in a hierarchical way like H, HODLR, $\mathcal{H}^2$ or HSS schemes, while some of them adopt a non-hierarchical clustering like block low-rank (BLR). Although the hierarchical ones provide better asymptotical complexities, a non-hierarchical clustering is simpler to implement and more flexible in parallel environments.

In this thesis, we conduct all our experiments through the parallel framework of the sparse supernodal BLR solver PaStiX. In this solver, the unknowns numbering (ordering) of the matrix is obtained by using the nested dissection method. This method ensures a good sparsity in the factorized matrix and allows parallel operations. We use the existing block low-rank (BLR) features of the solver to compress suitable dense blocks for an approximated solution.

In the context of supernodal solvers, the time to solution and memory usage is affected by the decision on when the blocks are compressed. In this respect, the sparse supernodal direct solver PaStiX offers two solutions: *Minimal Memory* and *Just-In-Time*. In the former one, all the admissible blocks are compressed before starting the factorization. Thus, the memory usage of the solver is highly reduced. However, the costly low-rank contributions can lead the solver to be even slower than the classical one. In the latter one, the blocks are compressed just after they have received all their contributions. In this way, time-to-solution is improved, but it does not reduce memory usage.

We aim to compare the hierarchical schemes to an efficient implementation of the BLR version in the context of a supernodal solver. Thus, in this thesis, we focus on improving the BLR scheme in terms of memory usage and/or time-to-solution. This improvement can be seen as a first step before the hierarchical implementation.

In Chapter 1, we discuss all the necessary background for the remainder of the thesis. Here, we first explain the sparse direct solvers in detail. Then, we present the low-rank representations with a special emphasis on the use of the BLR format within sparse solvers. Afterward, we explain the problem of applying the updates during the factorization. Here, we specifically focus on the solutions within the supernodal PaStiX solver. We conclude the chapter by discussing other solvers and our positioning.

In Chapter 2, we firstly study some block approximation (compression) kernels to determine the most efficient one in terms of storage and compression time. Although

there are a lot of different candidates for this aim, we compare only a few that meet our needs, which are based on stable QR factorization. Here, we implement all these existing methods similarly for a better comparison. In addition, we apply a stopping criterion to numerically determine the compression amount, which is the first time applied for two of the studied methods in the literature. This threshold-based compression guarantees a predefined precision. In this chapter, we are interested in compressing matrices with a similar size of those arising in our BLR solver. Moreover, as PaStiX uses sequential kernels for the compression purpose, we work in a sequential environment.

In Chapter 3, we improve an existing heuristic, which reorders the matrix entries after the nested dissection process to improve the compressibility and reduce the total number of updates in the solver. Through this reordering, an identification of the poorly compressible blocks is also performed. In this chapter, we first optimize the original heuristic through an empirically determined configuration. More specifically, we verify the necessity of compressing the poorly compressible blocks just after all updates are applied to them (like in *Just-In-Time*). Thus, this heuristic leads to an intermediate solution between *Just-In-Time* and *Minimal Memory* strategies as the highly compressible blocks are compressed before starting the factorization (like in *Minimal Memory*). For this reason, we propose to further improve the compressibility compared to the original heuristic because we compress both the highly and the poorly compressible blocks. Then, we observe our improvements in terms of flops and time.

In Chapter 4, we focus on another heuristic to identify the poorly compressible blocks to take advantage of *Just-In-Time* and *Minimal Memory* strategies at the same time. Here, we use the fill-in levels concept of incomplete factorization. In this heuristic, the blocks with high levels are well compressible, while others represent strong interaction blocks and are poorly compressible. We prove the correctness of our heuristic through the compressibility results of different fill-in level blocks. Then, in both parallel and sequential environments, we empirically validate the efficiency of our heuristic in terms of memory, flops, and time.

Finally, we conclude our work and discuss our perspectives.

This work is supported by the Agence Nationale de la Recherche, under grant ANR-18-CE46-0006 (SaSHiMi). 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 (https://www.plafrim.fr/).

Chapter 1

State of the Art

Many scientific applications need to solve large sparse linear systems of the form

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

where the matrix A is of size $n \times n$. Similarly, the unknown vector (x) and the right-hand-side vector (b) are of size n. Here, we can informally define a sparse matrix to be "any matrix with enough zeros that it pays to take advantage of them" [Wil71].

Sparse linear systems can be solved through many different approaches, which are divided into two main categories [GvL13]: direct and iterative solvers. Additionally, there are some hybrid methods [GH08, AGG+13, YL10, RBH12], which benefit from both approaches at the same time.

Iterative methods [Saa03] are usually used for very large systems thanks to their lower memory usage and computational cost compared to the direct methods. However, they frequently need a convenient preconditioner to converge to the solution. Therefore, finding an iterative method suitable for arbitrary systems is very challenging. Alternatively, direct methods [DER86, GL81, GHLN88] provide the most robust and reliable solutions. Yet, they are very costly in terms of memory usage and computational complexity. From now on, we only focus on improving the high cost of direct solvers.

All the applications have their own precision requirements and computations at the machine precision is not necessary. As we frequently need to deal with matrices that can be approximated into much lower dimensional blocks, some cheaper (approximated) direct solver alternatives are proposed. For example, incomplete factorization is proposed to neglect some values of the matrix to reduce the solver cost. In this approach, the dropped values can be chosen according to a numerical criterion (threshold) [KK97a] or based on the geometrical structure (fill-in levels) [DD97]. These solvers are mostly suitable as a preconditioner for the iterative solvers. Recently, low-rank approaches were introduced with better approximation criteria compared to the incomplete factorization. As low-rank approaches are convenient for high precision approximations, they can be used as a cheaper direct solver alternative than the classical version (with some accuracy trade-off). In this thesis, we focus on a sparse low-rank solver, which can be used either as a cheap direct solver, or as a good preconditioner for the iterative methods.

In Section 1.1, we explain the sparse direct solvers. In Section 1.2, we discuss the background on the low-rank schemes. In Section 1.3, we explain the problem of applying the contributions within the low-rank sparse solvers. Then, we provide the solutions of the PaStiX solver [Pic18,PDF+18] for this problem in Section 1.4. Finally, in Section 1.5, we discuss other solvers to show our positioning compared to them and state our objectives in this thesis.

1.1 Sparse Direct Solvers

A straightforward way of finding the x vector in Equation 1.1 is to compute it as $x = A^{-1}b$. However, not only does this approach requires A to be invertible, but also inverting the sparse matrix A does not guarantee to conserve its sparsity. In this respect, direct methods can be used to solve these systems in high accuracy.

In this work we focus on Gaussian elimination based direct solvers. These solvers firstly factorize the matrix into a product of triangular matrices depending on the matrix features. For example, symmetric positive definite (SPD) matrices can be factorized through LL^T (Cholesky), while symmetric matrices can use LDL^T factorization. For the general cases the LU factorization is adopted. Here, L, U and D represent respectively a lower triangular, an upper triangular and a diagonal matrix. Then, following the factorization of the matrix, the resulting triangular systems can be solved to obtain the original solution.

Sparse direct solvers follow four main steps:

1. Find the ordering
   - This step aims to increase the number of zeros in the factorized matrix structure (L and U matrices for the general case) and improve the parallelism.
2. Perform symbolic factorization
   - This step allows to allocate the factorized matrix structures before starting the numerical operations.
3. Factorize
   - This step factorizes the matrix in-place as the factorized matrix structure is known.
4. Solve
   - This step solves the triangular systems.

Let us explain each step in more detail in the following sections.

1.1.1 Ordering

After the factorization of a sparse matrix, A, some structural zero values become non-zero in the factorized matrix. These non-zero values that are zeroes in the original matrix are called fill-in. It is trivial to see that reducing the fill-in amount in the sparse matrices improves the storage and computational complexity. For this purpose, an ordering step is performed before the factorization in sparse direct solvers.

Let us introduce some necessary background through Figure 1.1 to understand the ordering step better. In the left side of the figure, we can see the original matrix, A, of order 5 on the left. This matrix can be represented with an adjacency graph, which can be expressed as $G(A) = (V, E)$. Here, V and E represent the graph vertices and edges, respectively. Each variable i in the matrix illustrates the vertex i of the graph. Note that for the unsymmetric matrices, we use the symmetric pattern $A + A^T$ for more efficient operations. Then, for each $a_{ij} \neq 0$ or $a_{ji} \neq 0$ for $i > j$, there is an edge in E, which is represented as a pair (i, j).

On the right side of Figure 1.1, we can see the information about the factorized matrix. Here, the filled elements on the factorized matrix, L, are represented in red. Similarly, the filled graph G^* illustrates both the original and fill-in edges after the factorization, where red color stands for the fill-in edges. The right-most structure, T, illustrates the dependencies when eliminating the unknowns during the factorization. This structure is called elimination tree [Sch82]. In this tree, the elimination of the unknowns is performed from bottom to top. Here, the siblings are eliminated in parallel since the fill-in occurs only from a node to the ancestors.

Now let us explain the importance of the unknown numbering (ordering) through Figure 1.1. As seen at the top row of the figure, the numbering leads to a dense matrix after factorization due to the introduced fill-in. In addition, we can see that each node is eliminated in sequential from bottom to top in the corresponding elimination tree, T. In the bottom row of the figure, the fill-in is avoided through the new numbering of the unknowns. This clever numbering also allows parallelism when eliminating the unknowns, as seen in its elimination tree. That is, the nodes from 1 to 4 can be eliminated in parallel, since there is no dependency between them. On the other hand, the node 5 is eliminated after the other unknowns since it depends on them.

Through Figure 1.1, we can observe the importance of the ordering since the fill-in introduced for large systems can be very crucial and an efficient usage of parallel architectures is necessary. Therefore, we need to order the matrix conveniently, before the factorization. In this respect, the ordering can be represented by a permutation matrix P. Then, instead of solving Equation 1.1, we can equivalently solve $(PAP^T)Px = Pb$.

We can order a sparse matrix through some methods like approximate minimum degree (AMD) [ADD96], minimum local fill (MF) [LN14, TW67] or nested dissection [Geo73] to successfully reduce the fill-in. The nested dissection method is a classical method in sparse solvers as it is also convenient for providing parallelism. It can be performed through well-known partitioning libraries like METIS [KK97b] or SCOTCH [Pel08]. We adopt this method from now on.

In Figure 1.2, the nested dissection partitioning is illustrated on an adjacency graph. This method recursively divides the graph (and sub-graphs) into two balanced sub-parts through a separator of minimal size. At each recursion, this method separates the sub-parts well to avoid direct interaction between them, so that they are only connected through the separator. As a result, the sub-part interaction blocks become zero and these sub-parts do not depend on each other.

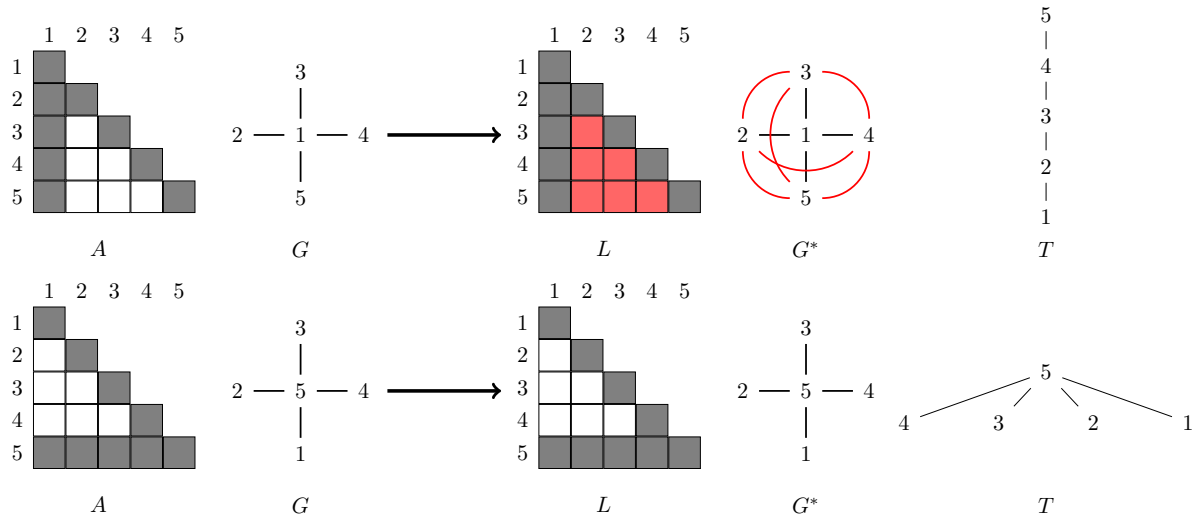


Figure 1.1: The impact of ordering on a sparse matrix with five unknowns. On the left, we see the matrices to be factorized (A) with the corresponding graphs (G). On the right, we see the factorized matrix structures (L) with the corresponding filled graph (G^*) and elimination tree (T). The red color represent the fill-in. The ordering at the top generates no fill-in, while the bottom ordering causes the matrix to be fully filled.

1.1.2 Block symbolic factorization

After the nested dissection partitioning, the symbolic factorization is performed to know the data structure of the factorized matrix before starting any numerical operations. For example, in the right part of Figure 1.2, we can see the data structure of the factorized matrix after the nested dissection ordering. Here, the fill-in which is caused by the separator interactions with the other sub-parts are shown in red and green. The upper red fill-in represents the interaction of the first and second sub-parts with the third sub-part (separator), while the bottom red fill-in illustrates the interaction of the forth and fifth sub-parts with the sixth sub-part (separator). Similarly, the green fill-in represents the interaction of all the sub-parts with the seventh sub-part (largest separator). In practice, this information is obtained through the matrix graph. When a node of the graph is eliminated during the factorization, its neighbors are connected to each other (clique). These new connections are represented as new edges in the filled graph. Then, this information about the factorized matrix allows to allocate the data structures before factorization and collect the data in the blocks for efficient BLAS Level 3 [DDCHD90] operations.

Each sub-graph, including the separators, obtained after the nested dissection is called supernode. We know that each supernode resulting from the nested dissection process is designed in a way that the unknowns which have similar non-zero structures in the factorized matrix are gathered together.

In Figure 1.2, we can see each supernode and corresponding intra-supernode interactions in each diagonal matrix (in gray). Here, there are seven supernodes resulting

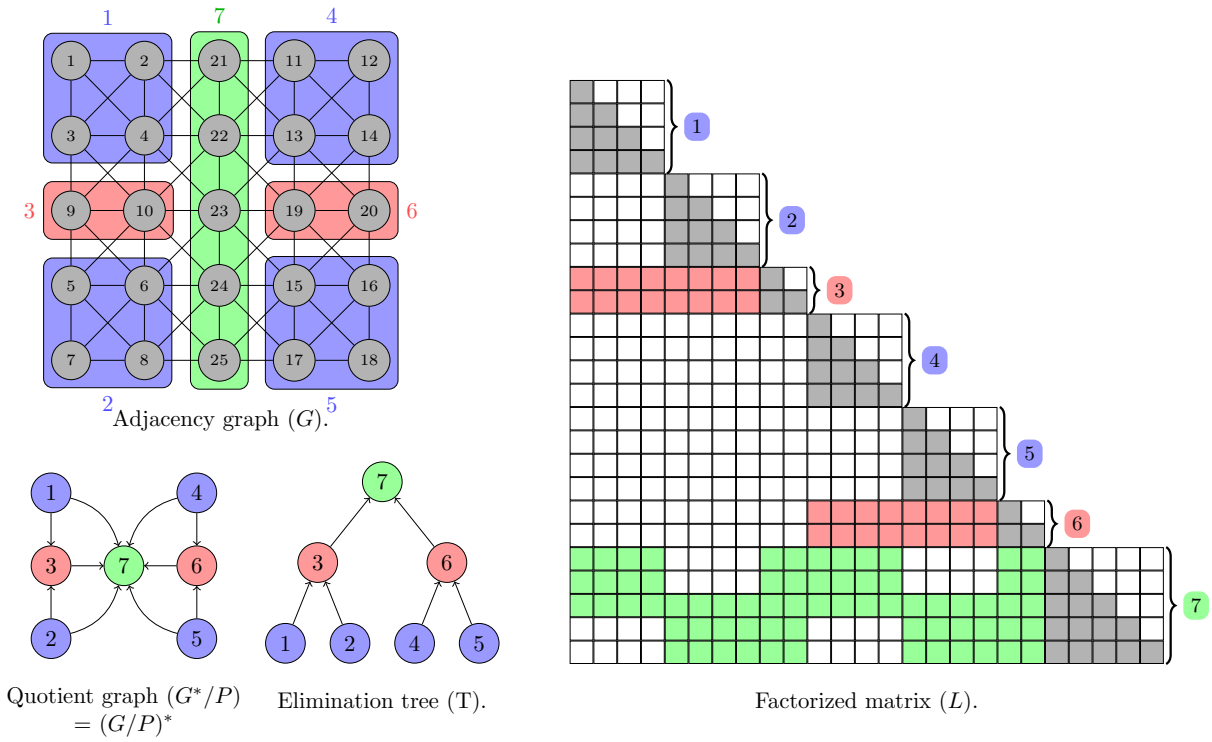


Figure 1.2: Nested dissection applied on the adjacency graph. The factorized matrix structure is represented on the right. The corresponding quotient graph and elimination trees are shown on the left bottom.

from the nested dissection partition. The red and green dense blocks represent the inter-supernode interactions. Here, each vertical panel consisting of one supernode's diagonal block and its interactions is called column block (or simply panel). Then, both the quotient graph and the supernode elimination tree can be obtained through the symbolic structure of L in Figure 1.2. Here, the quotient graph illustrates the supernode interactions, while the latter one shows the dependencies. Note that the top-most separators of large matrices can cause huge computations. Thus, in practice, they are clustered into smaller sub-graphs (so column blocks) before each panel is assigned to the processors [HRR02]. In this way the parallelism is increased.

1.1.3 Numerical factorization

The numerical factorization is performed after the preprocessing steps that are explained previously. In order to take advantage of the efficient BLAS Level 3 operations, this factorization works on the blocks of each column block of the sparse matrix. That is, the factorization basically iterates on each panel, where three main steps are applied at each iteration: factorize, solve and update. In the first step, the diagonal block of the current panel (green in Figure 1.3) is factorized. In the second step, other blocks of the current panel (off-diagonal blocks) are solved. The affected blocks of this step is colored

in blue in Figure 1.3. Lastly, the contributions of the current panel to other column blocks are performed. The updated blocks of other panels are colored in red in the figure.

Numerical factorization can be performed in two different ways: following the multifrontal [DR83] or the supernodal method. These approaches differ in terms of data locality and parallelism.

In the multifrontal method, the children update only their direct parents in the elimination tree. Therefore, they use some temporary storage (called fronts) for accumulating the contributions to be applied to the red blocks of Figure 1.3.

Opposite to the multifrontal method, supernodal approaches apply the update operations to any level of ancestors in the elimination tree. The supernodal method can be applied in two different ways depending on when the updates are performed: left-looking and right-looking approaches. The former approach updates the current supernode blocks just before eliminating it through the contributions from any level of descendants. On the other hand, the right-looking strategy allows to directly apply all the updates to any level of ancestors in the elimination tree, as soon as the contributions are computed. Supernodal methods avoid the extra storage used in the multi-frontal approach for storing the fronts, thus reduce the memory peak compared to it.

In this work, we focus on the right-looking supernodal approach. The right-looking panel-wise Cholesky algorithm can be seen in Algorithm 1. Note that we compute the numerical factorization in-place since memory is a problem for large matrices.

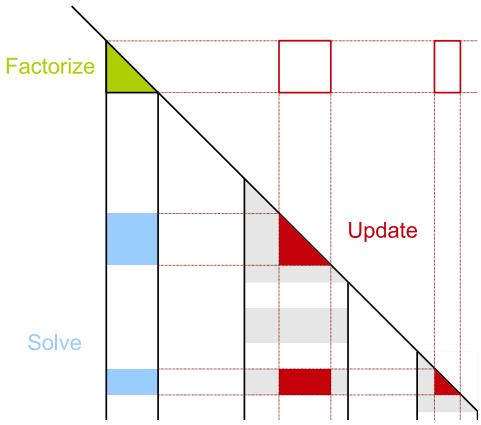


Figure 1.3: The three steps that are performed during the factorization of an arbitrary column block.

1.1.4 Triangular solves

After the factorization of the matrix into two triangular matrices, we have two steps to solve the resulting system. That is, considering a LU factorization on the matrix A, we end up with $LUx = b$. Therefore, we first solve $Ly = b$ (forward substitution) and then we solve $Ux = y$ (backward substitution). Note that in this thesis we do not focus on the triangular solve steps.

Algorithm 1 Sequential Cholesky factorization

```
1: for all column block A_*k in A do                    ▷ Numerical factorization
2:     Factorize(A_kk)
3:     for all block A_ik in A_*k do
4:         Solve( A_kk, A_ik )
5:         for all block A_jk in A_*k (with j <= i) do
6:             Update( A_ik, A_jk, A_ij )
```

1.2 Low-rank representations

As mentioned before, sparse direct solvers are very costly in terms of memory and time consumption. Nowadays, the introduction of low-rank representations has enabled the use of these solvers for larger systems. These representations basically aim to approximate the matrix. They represent matrices as multiplication of two matrices as $A_{m\times n} \approx U_{m\times r}V^T_{r\times n}$, where r (called rank) is much smaller than $min(m,n)$. As a result, the memory and computational complexity of the solver can be reduced. The rank can be determined through a predefined criterion based on either a fixed-rank or a fixed-precision to satisfy

$$\frac{||A - UV^T||}{||A||} < \epsilon, \tag{1.2}$$

where ϵ is the precision. At a given precision, the optimal approximation can be computed through the singular value decomposition (SVD). However, this method is too costly to be applied in practice. Therefore, some QR factorization based approaches are offered to reduce this cost, while providing close approximations to SVD. Especially, through the randomized methods, computations on large matrices are avoided by using small representative data [HMT11]. As a more detailed explanation on the approximation (compression) kernels is provided in Chapter 2, we skip further information here.

Many recent studies have tackled the problem of reducing the memory consumption of linear solvers with low-rank compression. For the sparse solvers, low-rank techniques are more challenging than for dense solvers [CPA+20, ALS+18, ALMK17] as the low-rank representations on the full matrix do not exploit the existing sparse structure. That is, after the nested dissection procedure, large zero blocks are introduced. Then, these zero blocks should be ignored to reduce memory and computational complexity. Therefore, in this thesis we focus on a low-rank solver that takes advantage of the existing sparsity of the matrix after the nested dissection and compresses the dense matrix parts independently.

In order to obtain the low-rank representation of a matrix, some hierarchical or non-hierarchical (flat) block clustering formats can be used on the dense parts of the matrix. Then, according to a block admissibility criterion, the compressible off-diagonal blocks can be approximated, while the non-admissible ones are kept uncompressed. Let us first explain the block admissibility condition in 1.2.1. Afterward, we provide the necessary background on the low-rank compression formats in Section 1.2.2, and continue with a more specific case (a flat representation, which is called block low-rank) in Section 1.2.3.

1.2.1 Block compression admissibility

In the applications (like the ones with PDE discretization), when the matrix is represented in blocks, some of the blocks can be represented in low-rank form [CDGS10]. For this purpose, some block admissibility criteria can be used to determine the compressibility of the blocks.

To decide which blocks to compress, Hackbusch [Hac15] defined some admissibility conditions. More specifically, he proposed both a weak and a strong admissibility condition.

In order to explain the admissibility criteria, let us first define a block with the unknown set σ as row indices and τ as column indices. Then, this block stands for the interaction of the sets σ and τ. Therefore, if $\sigma = \tau$, this block is a dense diagonal block, which represents the interactions within the same set. If $\sigma \neq \tau$ then the block can be represented in low-rank form if the admissibility criterion is respected.

According to the weak admissibility criterion, all the off-diagonal blocks are approximated:

$$\sigma \times \tau \; is \; admissible \quad iff \quad \sigma \neq \tau. \tag{1.3}$$

The strong admissibility condition relies on the problem geometry and the definition of the diameter of a set of unknowns ($diam(\sigma)$), as well as the distance between two sets ($dist(\sigma, \tau)$). The interaction between two sets of unknowns is then considered admissible if the distance between the sets is sufficiently larger than both of the diameters of the sets

$$\sigma \times \tau \; is \; admissible \quad iff \quad max(diam(\sigma), diam(\tau)) \leq \eta dist(\sigma, \tau), \tag{1.4}$$

where η represents a fixed value. As we can see, as the distance between the clusters increases, the rank gets smaller since their interaction gets weaker. The least restrictive strong admissibility criterion considers that blocks are admissible only if their distance is larger than 0 (i.e. all blocks except the close neighbors).

In order to exploit the low-rank features, it is necessary to correctly determine the blocks that are worth to be approximated. Thus, in a fully-algebraic context, we need to adopt an algebraic alternative to the admissiblity criteria defined by Hackbusch [Hac15]. In this respect, we will use the adjacency graph information (for diameter and distance) [GKLB08] to determine the block admissibility, instead of using the underlying geometry of the problems. Now let us discuss the clustering approaches to obtain the blocks on which we check the admissibility conditions.

1.2.2 Low-rank compression formats

As mentioned before, in order to approximate the matrices, the matrix unknowns are clustered, so that only the admissible blocks can be compressed. In Figure 1.4, we illustrate an example of the matrix unknown clustering on the left, while we see the corresponding cluster tree on the right. Here, $\mathcal{I}$ stands for all matrix unknowns to be clustered.

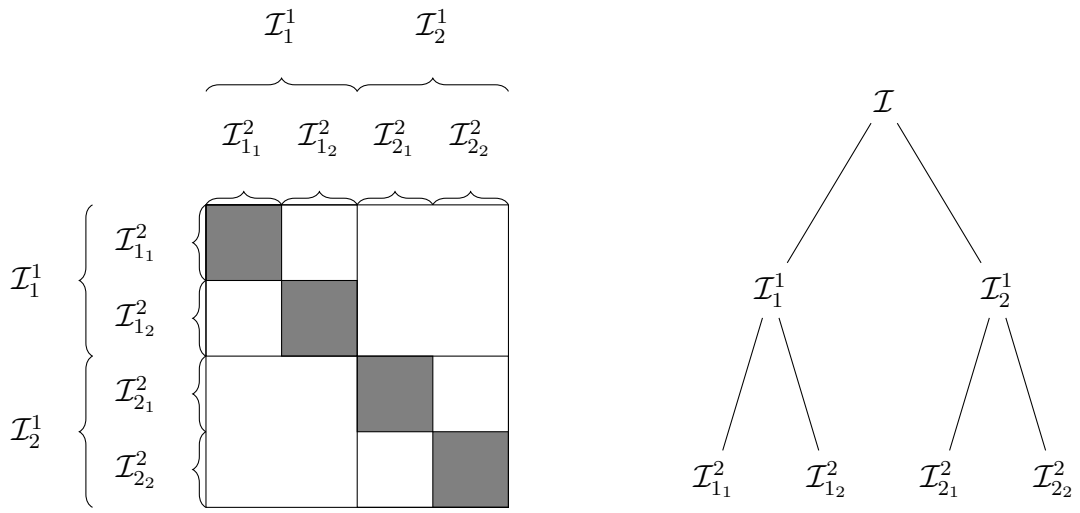


Figure 1.4: Clusters obtained (on the left) and the corresponding cluster tree (on the right).

A matrix clustering can be performed by using different formats. For example, Hierarchical (H) matrices [Hac99] are introduced to cluster the unknowns in a hierarchical way. Since this format adopts a strong admissibility, another hierarchical format called HODLR [AD16,CB15] is proposed as a weak admissibility criterion based alternative to it. In addition, other hierarchical formats like $\mathcal{H}^2$ [HB02] and Hierarchically Semi-Separable (HSS) [GLR+16,Xia13a] are offered to take advantage of the nested bases. That is, let us consider a low-rank representation of a block in the form UBV^T, where U and V are the orthogonal bases of this block. Then, in these solvers, the bases of a block are based on the bases of the descendants in the corresponding cluster tree. In this way, the storage cost of the solvers is improved. However, as $\mathcal{H}^2$, which uses a strong admissibility, and HSS, which adopts a weak admissibility, depend on the existence of the nested bases, they are more restrictive compared to H and HODLR formats.

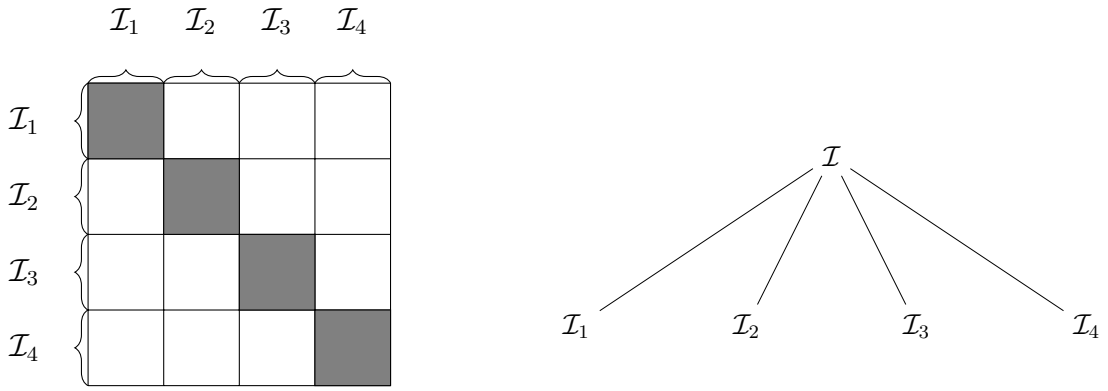


Figure 1.5: Four clusters obtained by a flat clustering (on the left) and the corresponding cluster tree (on the right).

As an alternative to the hierarchical formats, some non-hierarchical (flat) formats are also proposed. These formats can be seen as a special case of the hierarchical formats (illustrated in Figure 1.4), where all the unknown clusters are the direct children of $\mathcal{I}$ as seen in Figure 1.5. Here, we see the cluster tree (on the right) of a simple flat block clustering example (on the left).

Flat representations have larger asymptotical complexities in terms of flops and memory compared to the hierarchical schemes. However, they are simpler to implement and more convenient to parallelize. For example, arranging the block sizes small enough for a shared memory implementation or having good load balance in a distributed environment is easier with these formats compared to the hierarchical ones. Therefore, in this thesis we focus on these flat representations, more specifically the block low-rank (BLR) format, and explain it in the following section.

1.2.3 Block Low-Rank (BLR) format

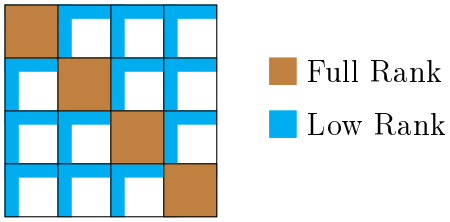


Figure 1.6: Block Low-Rank (BLR) compression on a dense matrix. Brown color represents the dense blocks, blue color represents the compressed matrices and white color stands for the sparsity generated thanks to the compression.

The BLR representation [AAB+15] of a dense matrix is illustrated in Figure 1.6. Here, we adopt a weak admissibility criterion for the sake of simplicity. In the figure, the dense diagonal blocks are kept in full-rank (in brown) as they represent strong interactions. However, the off-diagonal blocks are compressed into two lower dimensional blocks (in blue) to generate sparsity of the matrix. If strong admissibility criterion was adopted, some of the off-diagonal blocks could be in full-rank. As seen in the figure, BLR format provides similar size blocks.

As mentioned before, to take advantage of the existing zero blocks of the sparse matrices, the BLR representation can be applied only on the dense parts of the matrix, instead of the full matrix. In this way, unnecessary memory usage and computational complexity of the zero blocks are avoided.

We already explained that we obtain the column blocks of the factorized matrix through the nested dissection process, where each column block represents a supernode. Then, the largest separators are also clustered to reduce their sizes as explained in Chapter 3. This procedure results in the BLR representation of the dense parts of the matrix, where the blocks are of similar sizes. In addition, the large supernode interaction blocks are also suitable to be represented in low-rank. In practice, small

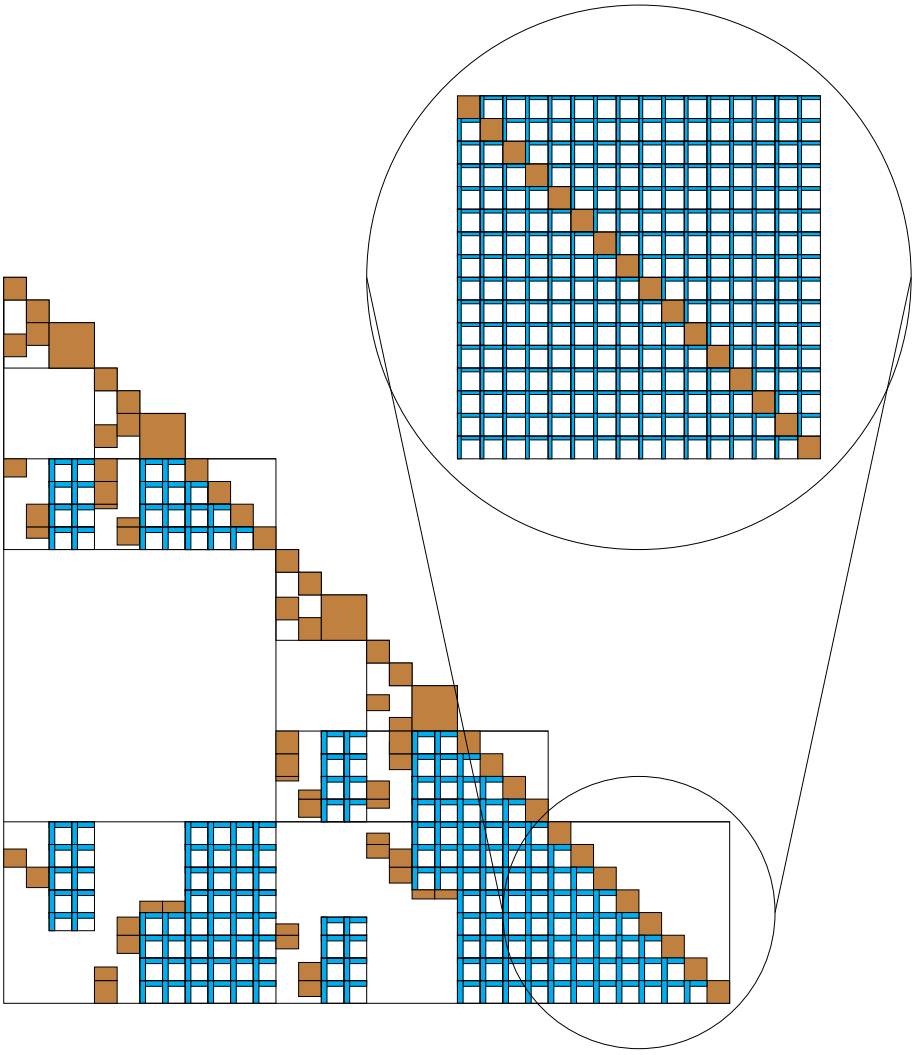


Figure 1.7: Block Low-Rank (BLR) compression on the dense blocks of a sparse matrix. Brown color represents the dense blocks, blue color represents the compressed matrices and white color stands for the zero parts.

blocks and high rank blocks, according to predefined criteria, are kept dense to avoid unnecessary compression. We illustrated this kind of BLR representation of a sparse matrix in Figure 1.7. As we see in the figure, the large zero blocks (in white) do not need storage and operations. Now let us specifically give the necessary background on the PaStiX solver, which is our focus in this thesis and adopts the BLR scheme.

1.3 Low-rank update problem of supernodal solvers

(a) Non-fully structured update (LR2FR) (b) Fully structured update (LR2LR)

Figure 1.8: Representation of two different low-rank update options (C $-=$ AB). Here, the contributing blocks A and B appear respectively in red and in blue. The updated block C is colored in orange and the impact of the contribution (AB) is in purple.

Block compression in the dense solvers and the sparse multifrontal solvers provides lower update flops and improves the memory footprint. However, sparse supernodal solvers use different block sizes in the update operation, where the low-rank cost depends on the largest block. This may result in even more expensive updates than the full-rank one. Nonetheless, supernodal approaches provide more parallelism and have less memory overhead compared to the multifrontal ones.

Figure 1.8 describes two different low-rank update (C $-=$ AB) options in a supernodal solver, which will be our focus in this thesis. Here, the updated block (C) is large and the contribution (AB) is small. The A, B and C blocks are respectively in red, blue and orange. The contribution, AB is shown in purple. M and N represent the dimensions of the updated matrix, while m and n stand for the dimensions of the contribution.

The contributing blocks for both strategies in Figure 1.8 are represented in low-rank. Although these blocks can also appear in full-rank within the solver, this is out of our scope. The updated block, C, is represented in full-rank on the left, whereas it is in low-rank on the right.

The non-fully structured strategy (LR2FR) in Figure 1.8a, computes the contribution, AB, cheaply by taking advantage of the low-rank representation of A and B. Additionally, as C is stored in full-rank, a simple addition of the contribution at a cost of order mnr (r being the rank of the contribution) is performed, further improving the computational complexity of the update operation. However, this approach does not reduce the memory usage as all the blocks are firstly allocated in full-rank to take advantage of the cheap addition of order mnr.

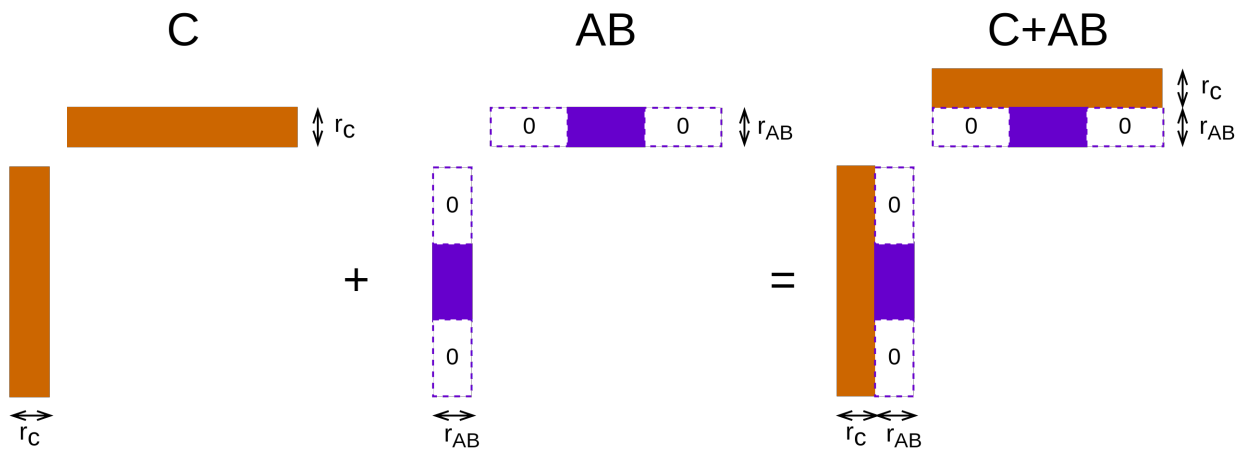


Figure 1.9: Accumulation of the U and V bases during the accumulation of two different size low-rank blocks. Here, the large C (brown) matrix gets the AB (purple) contribution of smaller size. Therefore, there is zero padding.

The fully structured updates (LR2LR) [Xia13b] are illustrated in Figure 1.8b. Here, all the blocks are compressed before starting the operations. As a result, the memory usage is reduced. In addition, the contribution (AB) is computed at a low cost similarly to the non-structured version. However, the addition step into C requires a complex low-rank to low-rank update with padding (zeroes are added to match the dimension of C). Let us illustrate the padding through Figure 1.9.

In Figure 1.9, the accumulation on the U and V bases in the structured approach is shown. Here, we observe that after the accumulation, some zero padding is used to align the contribution block (small) and updated block (large) vectors. Then, after the accumulation of several updates, the rank and size of C can highly increase. Thus, an additional re-compression is necessary to maintain a small storage and rank [Pic18]. This LR2LR operation with re-compression in the context of the supernodal method generates a flops overhead. As a consequence, the factorization time may be highly impacted, and can be even more costly than the full-rank solver, unless the matrix is highly compressible. Now let us introduce the solutions of the sparse supernodal solver PaStiX to the low-rank update problems we discussed.

1.4 PaStiX sparse supernodal direct solver

In our work, we conduct all of our experiments through the BLR supernodal sparse direct solver PaStiX [Pic18, PDF+18]. This solver targets both the real and complex matrices through the LU, LL^T, LDL^T, LL^H and LDL^H factorizations. PaStiX allows both static and dynamic shared memory usages, as well as external Parsec [BBD+13] and StarPU [ATNW11] run-time supports to exploit accelerators. As this is not our main focus in this thesis, we will not detail it here. Interested readers can find further

information in [Fav09, LFB+14]. Now let us explain the necessary background for the following chapters.

In PaStiX, two different compression strategies that differ depending on the update kernels are implemented to lower either the memory footprint or time-to-solution: *Just-In-Time* and *Minimal Memory*.

Algorithm 2 Cholesky BLR factorization through *Just-In-Time* strategy

```
1: for all column block A_*k in A do                    ▷ Numerical factorization
2:     Factorize(A_kk)
3:     for all block A_ik in A_*k do
4:         if A_ik is admissible then                   ▷ Compress all admissible blocks
5:             Compress(A_ik)
6:         Solve( A_kk, A_ik)
7:         for all block A_jk in A_*k (with j <= i) do
8:             Update( A_ik, A_jk, A_ij )
```

The *Just-In-Time* strategy in Figure 1.8a aims to reduce time-to-solution through late compression of C. In this strategy, the blocks are compressed only after receiving all the contributions on them, on the fly, during the factorization. This operation is represented in blue in Algorithm 2. Then, through the resulting non-structured updates (LR2FR), a simple addition of the contribution is performed, reducing the computational complexity of the updates. However, as all the blocks are allocated in full-rank in the beginning of the factorization, this scenario does not improve the memory peak compared to the full-rank solver.

Algorithm 3 Cholesky BLR factorization through *Minimal Memory* strategy

```
1: for all block A_ij in A do                           ▷ Compress all admissible blocks
2:     if A_ij is admissible then
3:         Compress(A_ij)
4: for all column block A_*k in A do                    ▷ Numerical factorization
5:     Factorize(A_kk)
6:     for all block A_ik in A_*k do
7:         Solve( A_kk, A_ik )
8:         for all block A_jk in A_*k (with j <= i) do
9:             Update( A_ik, A_jk, A_ij )
```

The *Minimal Memory* strategy in Figure 1.8b aims to reduce the memory footprint of the solver by early compression of C. This operation is illustrated in red in Algorithm 3. Indeed, as all admissible blocks are compressed before starting any numerical operation, it greatly improves the memory peak since the factorized matrix structure is never fully allocated. However, the fully structured updates (LR2LR) in the *Minimal Memory*

strategy are complex to perform as explained before. A detailed complexity analysis of the *Just-In-Time* and *Minimal Memory* strategies is provided in [PDF+18].

As we can see, neither *Just-In-Time* nor *Minimal Memory* is optimal. In order to take advantage of both the strategies in PaStiX, it is important to come up with a good preselection criterion to distinguish the poorly and highly compressible blocks. In this way, the former ones can be compressed on the fly (like in *Just-In-Time*), while the latter ones are compressed before starting any numerical operation (like in *Minimal Memory*). As a result, we can improve both the flops and memory overhead of the solver in an optimal way. Now let us observe our positioning in more detail compared to the other works.

1.5 Other solvers

Hierarchical matrices have been first introduced by Hackbusch [Hac99] for dense matrices. Since this initial work, there has been a lot of interest on this topic from variety of scientific and engineering domains. In this work, the author expressed the full matrix in the H format to approximate it and then proposed fully structured low-rank operations. In the context of sparse solvers, expressing the full matrix in a low-rank format does not take advantage of the existing sparsity. Thus, we approximate only the dense blocks of the sparse matrix.

In [GKLB09], an extension of the H matrices to the sparse case is proposed. Here, the geometry of the problem is used to obtain the clustering tree. In this work, the authors adopted the nested dissection partitioning to have large zero blocks in the matrix, so that only the remaining dense parts are approximated. In this way, they reduced the factorization cost of the sparse structured $H-LU$ solver. The drawback of their approach is that they do not efficiently exploit the structural zeros: they ignore the zeros of the sub-part to separator interaction. That is, let us consider the first level separator (green) in the graph of Figure 1.2 and the well-separated sub-graphs generated by this separator. Then, in this figure, the zero blocks in the last five rows of the corresponding L matrix is not taken into account in [GKLB09]. Therefore, they skip some memory and flops reduction. In our work, we perform symbolic factorization after the nested dissection process to determine and exploit all the structural zeros efficiently.

A black-box algebraic version of [GKLB09] is proposed in [GKLB08], where the graph information is used for the clustering without the knowledge of the geometry in the $H-LU$ factorization. Here, they presented the numerical results of both the geometrical and the algebraic $H-LU$ (and $H-Cholesky$ for the SPD case) solver in terms of time and memory. Although the geometrical solver shows better results, the ratio of the algebraic and geometric solvers does not exceed small constants. In addition, the authors also provided the implementation in a parallel context. In the parallel experiments, the geometrical and algebraic solvers demonstrate almost identical speedup results.

In [Hac15], the adaptation of some hierarchical matrix strategies from dense to sparse matrices are studied. In this work, an approach similar to the *Minimal Memory* strategy of

PaStiX is used, meaning that low-rank updates are performed. Here, all the contributions from the descendants in the tree are considered dense when generating the low-rank representation: structural zeroes are ignored. The assembly is then applied without zero-padding, which allows to reach good performance. In this work, the overhead of low-rank updates does not exist, opposite to our work, as the operations are always performed between blocks of similar sizes. However, it misses an opportunity for memory savings compared to our work since some zeroes are explicitly stored.

In [CB15], the authors proposed low-rank format similar to HODLR through the sparse supernodal solver Cholmod [CDHR08] for the SPD matrices. They used fixed rank based (depending on the block size) randomized compression kernels both for the off-diagonal blocks of the supernode interactions and the intra-supernode off-diagonal blocks. In practice, determining such a fixed rank to keep data with a desired precision is not trivial. In this work, they demonstrated good memory savings compared to the standard direct methods, which makes the low-rank solver suitable for large problems with high memory requirements. Nonetheless, their factorization is slower than the full-rank version and they do not fully take the sparsity of off-diagonal blocks into account. In our work, we allow to determine the rank numerically (with a precision based criterion) and exploit the structural zeroes efficiently.

In [Xia13a], HSS approximation is performed on the large fronts of a sparse multifrontal solver through a randomized technique for SPD matrices. The hierarchical low-rank algorithm in this work is extended efficiently for general cases in [GLR+16] through the sparse multifrontal solver StrumPACK. Here, the authors proposed a shared memory implementation. In this work they focused on finding an efficient and robust sparse solver, especially for the discretized PDE systems. They avoided saving the dense matrices explicitly thanks to the adaptive randomized sampling, which in return reduced the memory usage. They showed time improvements compared to the classical multifrontal solver. In [GLGR17], the authors improved the same work for distributed memory usage. Here, they focused on using this low-rank solver as a preconditioner. The algebraic StrumPACK multifrontal solver provides a fully structured approach, where all the blocks are compressed before the numerical operations. The HSS format in this solver provides better complexity than the BLR format that we adopt. However, opposite to us, the HSS based solver requires the existence of the nested bases.

The block separable (BS) representation in [Gil11] adopts a flat block clustering with a weak admissibility. Similarly, the block low-rank (BLR) representation, introduced in [AAB+15] through the multifrontal sparse direct solver Mumps [Mar17], also uses a non-hierarchical clustering. Here, BLR can be seen as an extended version of BS since it is not restricted to use weak admissibility condition.

In the Mumps solver, the CUFS (Compress, Update, Factor, Solve) technique is a similar approach to the *Minimal Memory* scenario from PaStiX. Using this technique, low-rank updates are performed within each front, but not between fronts. Thus, as fronts are allocated in full-rank before being compressed, memory savings are less important than the ones of the *Minimal Memory* strategy. Alternatively, the FCSU (Factor, Compress, Solve, Update) method applied to the fronts is similar to our *Just-In-Time* scenario.

Here, the updates are applied on the full-rank blocks and the panel compression is done during the factorization. Our *Minimal Memory* strategy within the supernodal approach enables a larger memory saving by preventing the allocation of the fronts inherent to the multifrontal method.

In [Mar17], the complexity and performance comparison of the multifrontal BLR solver MUMPS and the HSS solver STRUMPACK is provided. Here, by using the experimental results, the author offers to use BLR when high accuracy solution is required, while HSS is more convenient for low precisions.

In [MMPV22], the authors conducted a very similar work to us. Here, a solution to the problem of preselecting the poorly compressible blocks is proposed through the sparse supernodal solver PASTIX. The authors exploit performance models of the update kernel to decide whether or not to delay the compression of some of the blocks. In this work, this decision is taken at run-time during the numerical factorization and requires to generate correct models of the problem. In this initial work, they conducted the work in sequential context, while their final aim is the parallel environment in the future. In Chapter 3 and in Chapter 4, we focus on finding such a preselection criterion with different angles, where we identify the poorly compressible blocks before starting any numerical operations.

In this thesis, we focus on finding algebraic solutions to provide a low accuracy sparse direct solver or high accuracy preconditioner for the widest spectrum of applications. The precision of the solver can be determined by the user. Our aim is to study solutions to reduce the time-to-solution and/or the memory footprint of the BLR solver PASTIX by correctly preselecting the blocks and efficiently compressing them.

Chapter 2

Compression Kernels

Many fields like data mining, signal processing and computer vision need to handle large matrices. As the storage and matrix operations are too expensive for these matrices, some approximation (compression) techniques are proposed to reduce the cost. These techniques aim to represent the original matrix, A, in the form

$$A_{m\times n} \approx U_{m\times r} V^T_{r\times n} \tag{2.1}$$

such that

$$||A_{m\times n} - U_{m\times r} V^T_{r\times n}|| < \epsilon ||A||. \tag{2.2}$$

Here, the objective is to exhibit an approximation at a required precision (ϵ), where r is much smaller than $min(m, n)$. Unfortunately, determining the most suitable compression technique for arbitrary problems is difficult. Therefore, each application needs to come up with a convenient method according to the requirements like memory, time, stability and accuracy.

As mentioned in Chapter 1, we exploit the existing sparse structure in our sparse solver and split the dense parts of the matrix into blocks (in BLR format). Then, these blocks are approximated by the product of two lower dimensional blocks if they are admissible. In this chapter, our purpose is to compare some interesting compression kernels that can be used on these blocks to improve the solver. More precisely, we want to determine a fast and stable kernel that compresses admissible dense matrix blocks in the form (2.1), with r as small as possible. Here, compressing the data with smaller r values targets to improve both the memory footprint and the flop count during the factorization and solve steps.

In Section 2.1, we provide all the necessary background on the compression kernels. In this section, the motivation of different kernels, especially the ones on which we will focus, is also explained. The literature review specific to the four compression kernels that we concentrate on is provided in Section 2.2. In Section 2.3, we define some notations to be used in the algorithms of this chapter. Then, in Section 2.4, four different compression methods are studied in detail. In Section 2.5, their complexities are shortly summarized

to show the big picture. In Section 2.6, the numerical results of some generated matrices are observed in terms of stability, performance, compression ranks and accuracy, whereas in Section 2.7, the real-life case matrix results are discussed. Finally, in Section 2.8, the conclusion of this chapter is provided.

2.1 Background

In order to understand the compression kernels, we first need to provide the meaning of r in the approximation (2.1). Mathematically, the rank of a matrix (r) stands for the number of linearly independent columns/rows of the matrix. Then, if we represent a matrix as multiplication of two matrices as

$$A_{m\times n} = U_{m\times r} V^T_{r\times n}, \tag{2.3}$$

r can be determined through the singular value decomposition (SVD), which has the form

$$A_{m\times n} = \hat{U}_{m\times m} \Sigma_{m\times n} \hat{V}_{n\times n}. \tag{2.4}$$

Here, $\hat{U}$ and $\hat{V}$ are orthogonal matrices, while Σ is a diagonal matrix with the singular values, σ_j, on the diagonal with $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_r > 0$ and $\sigma_{r+1} = \sigma_{r+2} = ... = \sigma_n = 0$. Here, the rank is equal to the number of non-zero singular values in Σ.

When we work in finite precision, we are usually interested in only the large values and we omit the small values depending on our precision requirement. In this case, as mentioned in Section 1.2, an approximated rank is determined by using either a fixed-rank or a fixed-precision based criterion to satisfy the condition (2.2), where r is the rank of the approximation and ϵ stands for the precision. Observe that this rank is numerically computed and it can have different values for different precisions. From now on, r will only represent approximation ranks in this chapter. Determining a convenient fixed-rank to ensure a targeted accuracy is not trivial in practice. Thus, the fixed-precision based criterion provides better solutions for abstract problems. We will adopt a fixed-precision based criterion in our work.

If the equation (2.4) is written in the block form

$$A = \begin{bmatrix} \hat{U}_1 & \hat{U}_2 \end{bmatrix} \begin{bmatrix} \Sigma_{11} & 0 \\ 0 & \Sigma_{22} \end{bmatrix} \begin{bmatrix} \hat{V}_1 \\ \hat{V}_2 \end{bmatrix},$$

the approximation (2.1) can be written as

$$A \approx UV^T = \hat{U}_1 \Sigma_{11} \hat{V}_1, \tag{2.5}$$

where $U = \hat{U}_1$, $V^T = \Sigma_{11}\hat{V}_1$ and Σ_{11} is of size $r \times r$.

In this chapter, we will use either

$$\frac{||\Sigma_{22}||_2}{||A||_F} = \frac{\sigma_{r+1}}{||A||_F} < \epsilon \tag{2.6}$$

or

$$\frac{||\hat{U}_2\Sigma_{22}\hat{V}_{22}^T||_F}{||A||_F} = \frac{||\Sigma_{22}||_F}{||A||_F} = \frac{\sqrt{\sum_{i=r+1}^{n}\sigma_i^2}}{||A||_F} < \epsilon \tag{2.7}$$

to find the numerical rank r through SVD.

The SVD method is not the only way to compress the matrices in the form (2.1). Depending on the application requirements, different compression methods can be adopted. For example, for large sparse matrices, some methods that maintain the sparse structure can be used [BPS05]. These methods are out of our scope. In our work we specifically focus on compressing the dense blocks resulting from the BLR representation of sparse matrices as explained in Section 1.2.3. Additionally, we are interested in compressing in a sequential environment since these blocks are not large enough to take advantage of parallel environments. As the precision based numerical stopping criterion (presented in Section 2.1.1) in our work ensures the precision of the compression, our main aim is to choose a stable and fast kernel for our solver, which can keep the representative data with a rank as small as possible.

At a given fixed numerical rank, the Singular Value Decomposition (SVD) reaches to the optimal error, through both the spectral and Frobenius norms [EY36]. However, the cost of SVD is too high for large matrices, as it requires all the decomposition to complete to determine the rank. Therefore, some alternative methods are proposed to obtain a rank as close as possible to the one obtained using SVD, while being much faster at a given precision. These methods include but are not limited to the adaptive cross approximation (ACA) [BR03], column pivoted QR (QRCP) [DG15], rank revealing QR (RRQR) [Cha87], interpolative decomposition [LWM+07, MT11], UTV [MQOH17], and randomized SVD [HMT11].

The ACA method is proposed as a very efficient method for smooth kernel matrix approximations. However, since the correct construction of the low-rank basis is not ensured, it might fail to converge [BG05]. Therefore, despite its low cost, we prefer more stable methods.

Some methods like interpolative decomposition and UTV are powerful in terms of the approximation quality. Nonetheless, the QR factorization with column pivoting (QRCP) methods can produce ranks close to the one of SVD, while being much faster. For example, in [MQOH17], the authors show that the randomized UTV method they propose is competitive to QRCP, as they have the same asymptotical complexity. However, it can run faster than QRCP only in a parallel environment with matrices larger than the ones we are interested in. In a sequential environment with matrix sizes of our interest (dimensions smaller than 2000), QRCP can run more than 2× faster than UTV. Therefore, QRCP methods are commonly adopted in our context.

The disadvantage of the QRCP methods is that some ill-conditioned matrices, like Kahan matrix, can reach larger ranks than expected [GvL13]. However, it is worth noting

that this problem occurs very rarely in real life. Nevertheless, some rank revealing QR (RRQR) methods are proposed, with an additional cost, to avoid this issue. From now on, we focus on some promising QR methods in our context.

2.1.1 QR based compression kernels as an alternative to SVD

In this section, we briefly present the QR based compression methods before providing more specific details on the ones that we will study in this chapter. A QR decomposition has the form

$$A = QR,$$

where Q is orthonormal and R is an upper trapezoidal matrix (non-square matrix, where all elements under the diagonals are zero). Opposite to SVD, this decomposition, by its nature, does not provide a rank revealing feature. For this purpose, an orthogonal matrix Q_G (i.e. $Q_G Q_G^T = Q_G^T Q_G = I$, where I is the identity matrix) is applied to the original matrix A. In this way, we gather the representative data on the left side to omit the remaining unnecessary ones on the right. Then, considering the block form of the QR compression method as

$$AQ_G = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix} \approx Q_1 \begin{bmatrix} R_{11} & R_{12} \end{bmatrix}, \tag{2.8}$$

where R_{11} is of dimension $r \times r$, we obtain the low-rank form (2.1) with $U = Q_1$ and

$$V^T = [R_{11} R_{12}] Q_G^T.$$

Remember that the SVD method is the optimal compression kernel. Therefore, as the QR based compression kernels aim to produce an approximation as close as possible to SVD, the desired approximation criterion of the QR methods can be based on the Frobenius norm of the trailing matrix as

$$||Q_2 R_{22}|| = ||R_{22}|| \approx ||\Sigma_{22}||, \tag{2.9}$$

where $||Q_2 R_{22}||$ is the trailing matrix norm of the block QR representation in the equation (2.8). Then, similar to the criterion (2.7), the compression rank of the QR methods can be determined numerically through the criterion

$$\frac{||R_{22}||}{||A||} < \epsilon.$$

Note that, opposite to SVD, whenever the stopping criterion is satisfied (if we can evaluate it), the QR process terminates and Q_2 and R_{22} are not computed as they are not needed. That is why, QR compression kernels are cheap alternatives to SVD when they are used to satisfy 2.9.

There are different ways to compute the QR decomposition: Gram-Schmidt process, Householder transformation or Givens rotations method [GvL13, TB97]. As the

Householder transformations are more stable than the Gram-Schmidt and cheaper than the Givens rotations, we adopt the Householder based QR approaches in our work.

Note that in this chapter we focus on panel-wise QR factorizations (instead of factorizing the matrix index by index) for exploiting the level-3 BLAS [DDCHD90] operations to be more efficient on top of modern architectures. The panel-wise QR is illustrated in Figure 2.1 on the left, where b stands for block size. Here, the current panels of b columns and rows, respectively, are in yellow and the trailing matrix is in green. Then, during the factorization, the trailing matrix is updated at the end of each panel iteration through the panel contributions at once, instead of at each column contribution separately. Although this panel-wise update operation improves the performance, the column pivoted version of the QR factorization requires an additional storage to be able to update the column norms, which is necessary to determine the next column pivot [QOSB98]. Now let us focus on more specific background on the compression kernels that we will compare in this chapter.

2.1.2 Different features of QR based compression kernels

In this section, we briefly discuss the differences of the QR based compression kernels that we will study in the remainder of the chapter. The first difference is the way of gathering the representative data to the left part of the matrix. The second one is whether the randomized sampling is used or not, while the third one is based on when the trailing matrix updates are performed. Let us respectively explain these three differences in three sub-sections.

Gathering the data to the left of the matrix

The first difference of the compression kernels in the following sections is the choice of the Q_G matrix in the approximation (2.8). This choice affects how good we can approximate the SVD method. It can simply be chosen as a column permutation matrix to gather the representative data only to the left part of the matrix. For example, the column with maximum 2-norm can be chosen as the pivot at each iteration. Alternatively, the Q_G matrix can be selected as a convenient rotation matrix to have more control over the quality of the approximation.

No matter whether a permutation or a general rotation matrix is chosen, the computation of the Q_G matrix is inefficient and it is a bottleneck for large matrices. Therefore, in the next section we explain a technique to reduce this cost.

Randomization technique

The second difference of our kernels, randomized sampling, aims to reduce the cost of generating the Q_G matrix. This technique projects the original large matrix $A_{m\times n}$ on a lower dimensional representative matrix $B_{d\times n}$, where $d \ll m$. Through this projection, the cost of memory-bound operations can be reduced.

As an example, we can observe an original matrix and its representative (sample) matrix in a column pivoted QR factorization in Figure 2.1. Here, j stands for an arbitrary index during the factorization and b is the block size. Green color represents the sub-matrices which will be used in the next panel iterations, while the yellow color stands for the panels which are completed in the current iteration. Observe that the row dimension of the sample matrix is chosen as $b+p$. Here, the oversampling size, p, stands for a small fixed number, which is used to improve the quality of the projection. Then, thanks to this projection, we can cheaply compute the panel pivots through the matrix B instead of computing the column pivots through the original matrix. As a result, we can improve the performance for large matrices since column pivoting is a bottleneck in the QR method.

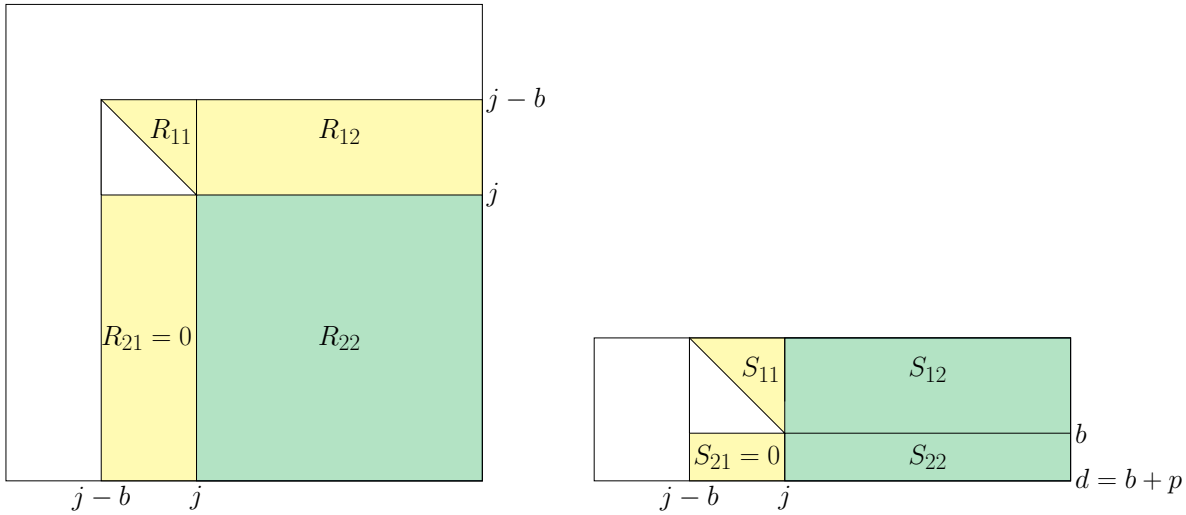


Figure 2.1: Panel-wise QR with column pivoting on the original matrix $A_{m\times n}$ (on the left) and on the sample matrix $B_{(b+p)\times n}$ (on the right). Here, $b+p \ll m$, where b stands for the block size and p is the oversampling size, which is used to obtain more representative data on the sample matrix. The yellow panels represent the parts which are completed after the current panel iteration. The green parts illustrate the trailing matrices which will be used in the next panel iterations. The upper triangular yellow matrices stand for the factorized partial R matrices.

Left-looking and right-looking approaches

The last difference of our methods is whether the decomposition is right-looking or left-looking like the LU factorization as mentioned in Section 1.1.3. This feature is based on when the trailing matrix is updated. Let us explain it through Figure 2.2.

On the left of Figure 2.2, the left-looking approach is presented. Here, j stands for the current index. In this method, the already computed yellow panel contributions are used to update the current green panel. Here, the current panel is updated just before being factorized. In this way, the unnecessary full trailing matrix updates at

each panel iteration are avoided. However, in the left-looking approach an additional storage is required compared to the right-looking version. That is, in the classical QRCP algorithm, the column norms should be downdated at each column iteration to be able to determine the next pivot column. When the trailing matrix update is postponed during the QRCP factorization, this downdate operation causes a problem. As explained in [QOSB98], although the block-wise update operation requires the earlier row values of A before the block-wise column reductions, these rows should have been already updated for downdating the column norms. Therefore, this problem can be solved at the expense of an extra storage for the previous reflector accumulations.

On the right of Figure 2.2, the right-looking approach is illustrated. Here, at the end of each panel iteration, all the trailing matrix (green part) is updated through the contributions from the current panel (yellow parts) in the figure. This feature is costly, especially for large matrices with small ranks, as the trailing matrix after the approximation rank is unused. However, since the panels are small and the updated parts are large for these matrices, this feature is promising for parallel environments.

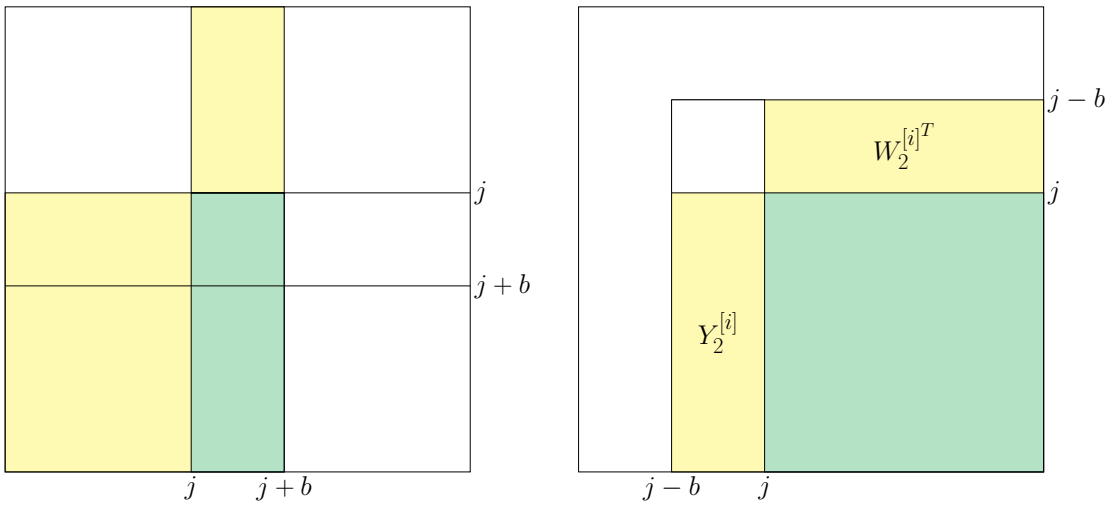


Figure 2.2: Inter-panel left-looking and inter-panel right-looking approaches. The former is illustrated on the left, while the latter is on the right. The contributions of the yellow color parts are used for updating the green parts. Right-looking approach updates all the trailing matrix at the end of each panel iteration, whereas the left-looking one only updates the current panel at the beginning of each panel iteration.

2.1.3 Specifications

In this section, we provide the necessary specifications to understand the following sections better. For the sake of simplicity, we only study the matrices with real values, where the superscript T represents their transpose. However, all formulas can be easily adapted to complex matrices. In the following sections, the computational complexities of the methods stand for the square matrices of dimension $n \times n$.

In the algorithms, we use indices starting from 0. At the end of the algorithms, R is stored in-place at the upper triangular part of A. Therefore, by applying the inverse permutation or rotation, depending on the method used, we obtain the V^T matrix. The $U = Q$ matrix can be obtained by using both the reflectors, stored in-place in the lower part of the matrix, and their scalar factors vector. This can be performed easily through the $xUNGQR$ routine of LAPACK [ABB+07] for example.

The stopping criterion (2.6) allows to reach smaller ranks with SVD compared to the criterion (2.7). However, we use the Frobenius norm of the trailing matrix when applying our stopping criterion within the QR methods. Therefore, for being consistent with the QR methods of this chapter, we will mostly focus on the latter criterion (2.7) with Frobenius norm.

2.2 Related work

In this section, we explain the related work on the four QR methods that we study in this chapter, which differ from each other through the features discussed in Section 2.1.2.

In [DG15], the authors propose a panel-wise randomized QR with column pivoting (RQRCP) method with a new sample matrix update formula, where they adopt a fixed-rank based stopping criterion. The update formula in this work improves the randomization method cost by eliminating the resampling at each panel iteration. Moreover, they also offer a truncated randomized QR with the column pivoting (TQRCP) method to avoid unnecessary trailing matrix updates. Through these methods, they also come up with an efficient truncated SVD algorithm. In this chapter, we will study the RQRCP and TQRCP methods in [DG15] respectively in Sections 2.4.2 and 2.4.3.

In [MQOHvdG17], the authors independently conduct a similar work to [DG15] on the RQRCP method that is based on Householder reflections, with a fixed rank criterion. Here, they also contribute to the RQRCP method with a sample matrix update, instead of resampling, which is mathematically equivalent to [DG15].

In [XGL17], both [DG15] and [MQOHvdG17] RQRCP methods are investigated with a fixed rank based criterion. The authors analyze the efficiency of the sample matrix updates from these two papers, as well as analyzing the reliability of the randomized QR with column pivoting. In addition, they propose a spectrum revealing QR method for the rank revealing problem of the ill-conditioned matrices, thanks to some extra interchanges.

In [Mar15], the author proposes both column pivoted and rotational rank revealing QR algorithms, as well as a power method for a better range convergence, with a fixed rank criterion. Depending on the convergence to the singular values, the randomized rotational QR (RQRRT) method can give closer results to SVD by collecting most of the data mass on the diagonals. Although the RQRRT method is more expensive than the QR with column pivoting methods, we will include this method in our study, in Section 2.4.4, to see if a good trade-off between performance and approximation quality can be reached.

In [MV16], the authors propose numerically determined rank procedures. Here, the algorithms are more general randomized methods and can be adopted as partial SVD,

CUR, interpolative decomposition or RRQR. The offered rank revealing QR can be seen as a modified version of column pivoted Gram-Schmidt. This method is using the numerical stopping criterion similar to us. In this work, they also propose a power method to improve the accuracy when handling slowly decreasing singular values.

In [HMT11], the authors conduct a detailed study of different randomized methods. Here, also a randomized numerical rank decision criterion is proposed.

Note that all the methods we explain in this chapter are existing methods. However, we aim to implement them in a similar fashion for a better experimental comparison. As mentioned before, we determine the compression ranks numerically through our stopping criterion to exploit low-rank features, without any underlying problem knowledge requirement. This criterion is applied to the TQRCP and RQRRT methods for the first time in the literature. Now let us provide some necessary notations that will be used in the following sections.

2.3 Notations

For the sake of simplicity, we use some notations to avoid detailed index specifications inside the algorithms of this chapter:

- $A^{[i]}$ represents the sub-matrix part that we use in the i^{th} panel iteration. For example, considering we already factorized $2b$ columns of A, $A^{[2]}$ represents the matrix $A_{2b:m;2b:n}$. The notation can be trivially converted to the vectors.

- At the i^{th} panel iteration, $A^{[i]}$ is represented in the block form

$$A^{[i]} = \begin{bmatrix} A_{11}^{[i]} & A_{12}^{[i]} \\ A_{21}^{[i]} & A_{22}^{[i]} \end{bmatrix},$$

where $A_{22}^{[i]}$ stands for the trailing matrix and it is used as $A^{[i+1]}$. We illustrate the matrix and vector (N) representations at the i^{th} panel iteration in Figure 2.3.

- $A^{(j)}$ represents the sub-matrix part that we use in the j^{th} column iteration. For example, considering we already factorized $2b+3$ columns of A, $A^{(2b+3)}$ represents the matrix $A_{(2b+3):m;(2b+3):n}$. The notation can be trivially converted to the vectors.

- At the j^{th} column iteration, $A^{(j)}$ is represented in the block form

$$A^{(j)} = \begin{bmatrix} a_{11}^{(j)} & A_{12}^{(j)} \\ A_{21}^{(j)} & A_{22}^{(j)} \end{bmatrix},$$

where $A_{22}^{(j)}$ stands for the trailing matrix and it is used as $A^{(j+1)}$. We illustrate the matrix and vector (N) representations at the j^{th} column iteration in Figure 2.4.

- $A_{(l)}$ represents the l^{th} column of the matrix A. In case of a vector, it stands for the l^{th} index of the vector.

Observe that both $A^{[0]}$ and $A^{(0)}$ represent the original matrix A in our notations. Now let us focus on the compression kernels that we will compare in this chapter.

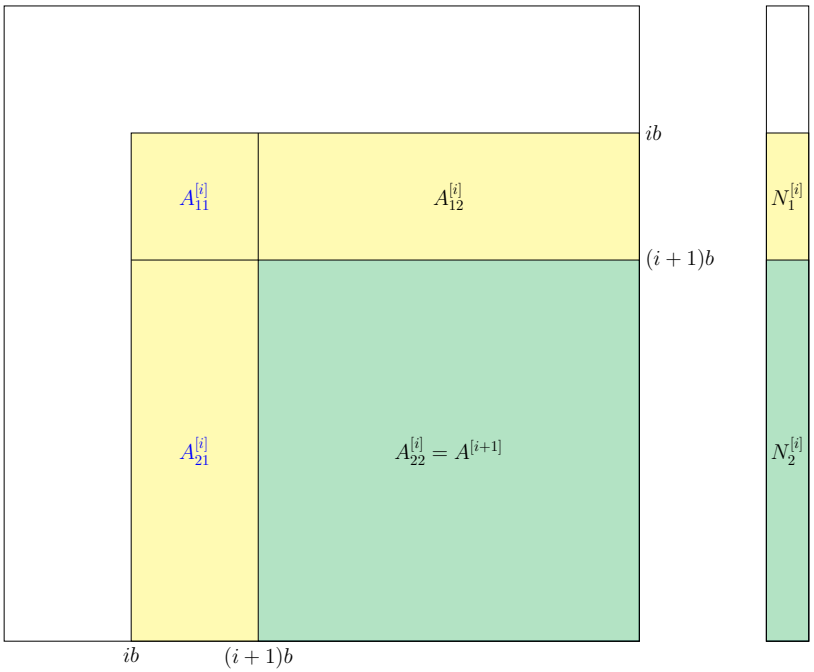


Figure 2.3: Our notations and the sub-parts they stand for during the factorization. On the left we present the panel-wise matrix (A) notations, while on the right we show the vector notations on an arbitrary vector N.

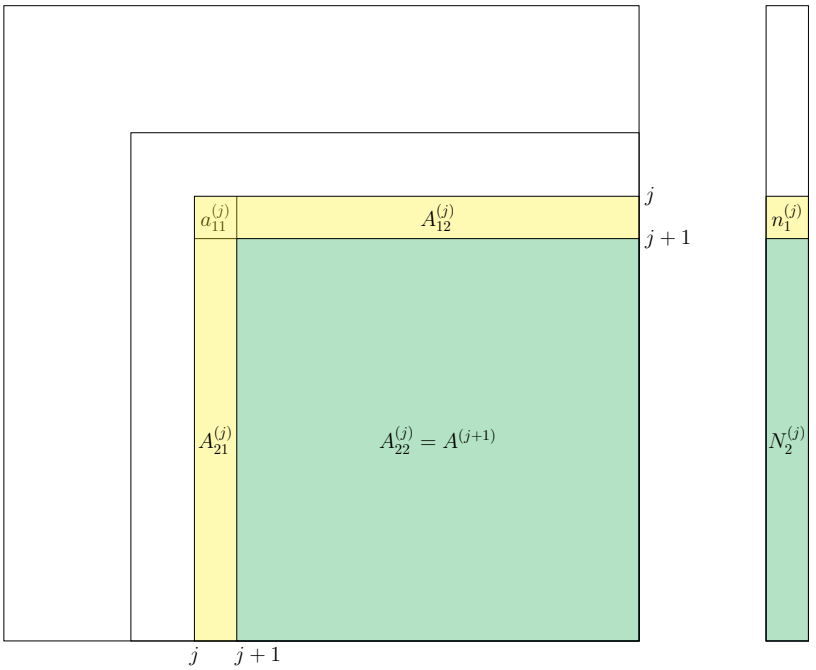


Figure 2.4: Our notations and the sub-parts they stand for during the factorization. On the left we present the column-wise matrix (A) notations, while on the right we show the vector notations on an arbitrary vector N.

2.4 QR based compression kernels

In this section, we study four different QR based compression methods in detail. We explain our most basic method, QR with column pivoting (QRCP), in Section 2.4.1. In Section 2.4.2, the randomized QR with column pivoting (RQRCP) is studied. Then, the left-looking version of the RQRCP method is provided in Section 2.4.3. Finally, a rank revealing QR based compression kernel which adopts a rotational matrix is explained in Section 2.4.4.

2.4.1 QR with column pivoting (QRCP)

QR with column pivoting (QRCP) can be seen as the most basic method in this chapter. The QRCP we adopt is based on the LAPACK $xGEQP3/xLAQPS$ methods, similar to [DB08]. We start with a version of $xGEQP3$ which was modified by Alfredo Buttari for the MUMPS BLR solver [Mar17]. This method uses a panel-wise approach to take advantage of the level-3 BLAS operations and thus reduces the slow memory-cache passes of large matrices.

QRCP has an intra-panel left-looking feature. That is, each column of the panel is updated just before being factorized. On the other hand, it uses a right-looking approach for the inter-panel operations. Therefore, at the end of each panel iteration, the whole trailing matrix is updated with the current panel contributions. In Figure 2.5, we illustrate the intra-panel left-looking feature, while in Figure 2.6 we show the inter-panel right-looking feature of QRCP. In the figures, j stands for the current index and r is the last index of the previous panel. The yellow parts are used for updating the green ones similarly to Figure 2.2.

Algorithm 4 illustrates the QRCP method. Here, C stands for the column norms array (of size n), while P represents the permutation matrix. In practice, P is an array, which keeps all the corresponding pivot indices, $p^{(j)}$, to swap the columns cheaply. In our code, the column norms are either updated or explicitly computed (only when required) as in [DB08]. Then, as we have the 2-norm values of the columns thanks to the column norms array C, whenever our pivot column norm is smaller than the required precision, we compute the trailing matrix norm. More precisely, we compute the Frobenius norm of the trailing matrix by computing the 2-norm of array C. Here, if the trailing matrix norm is also smaller than the required precision (our stopping criterion), we set the numerical rank to the current index value and terminate the function.

As seen in the algorithm, we start each column iteration by checking our stopping criterion. If it is not satisfied, the column which has the maximum norm-2 value is chosen as the pivot. As the algorithm is left-looking intra-panels, at each column iteration j, we update the current column by using the computed panel reflector matrix, $Y^{(j)}$, and the corresponding part of the panel accumulation matrix, i.e. $W_1^{(j)^T}$. In Figure 2.5, the yellow parts illustrate these contributing parts to the current green column. After generating the elementary Householder reflection for the current column, we compute the accumulations caused by the current row, $w^{(j)^T}$. As we mentioned before, this storage for

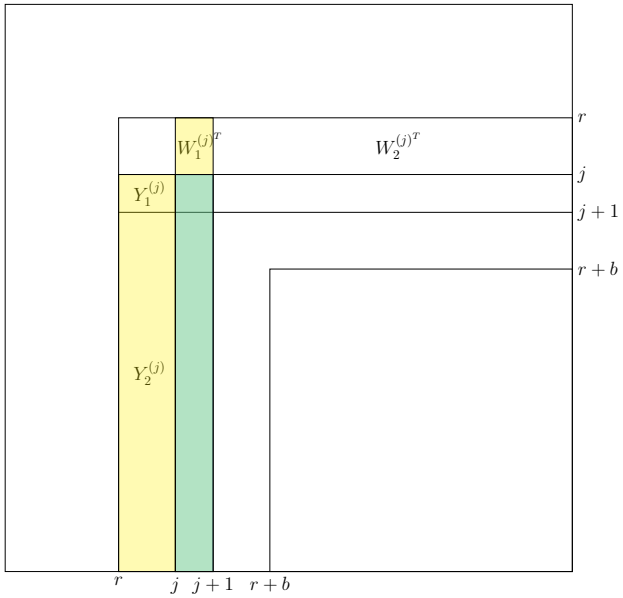


Figure 2.5: Left-looking intra-panel update of QRCP. The updated column is in green, whereas the contributing parts are in yellow.

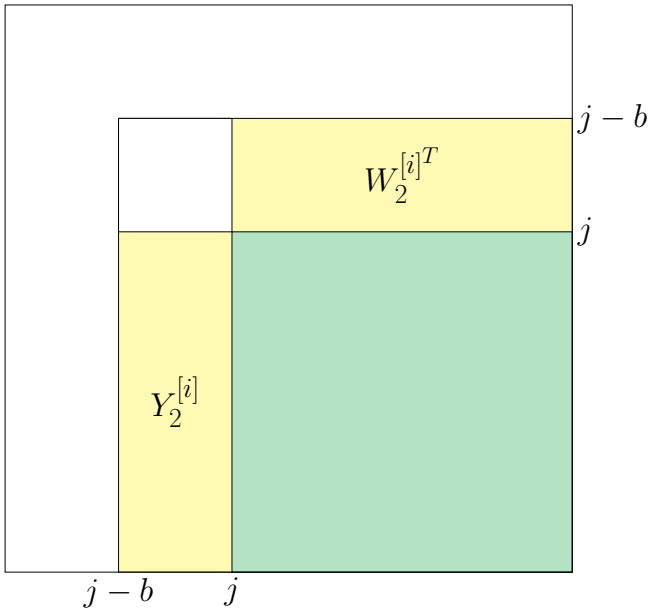


Figure 2.6: Right-looking inter-panel update of QRCP. The updated parts are in green, whereas the contributing ones are in yellow.

Algorithm 4 Block QR with Column Pivoting

1: **function** QRCP(A, ϵ, b)
2: **for** l = 1, 2, ..., n **do**
3: $C_{(l)} = ||A_{(l)}||_2$ ▷ Initialize column norms array
4: **for** i = 0, b, 2b, ..., min(m,n) **do** ▷ Panel iteration
5: **for** j = i, i+1, ..., min(i+b-1, min(m,n)) **do** ▷ Column iteration
6: **if** $||C^{(j)}|| \leq \epsilon||A||$ **then return** $r = j$ ▷ Check the stopping criterion
7: $UpdateNorms(C^{(j)})$ ▷ Update the column norms
8: $p^{(j)} = \underset{l \in [j,...,n]}{\operatorname{argmax}} C_{(l)}$
9: $swap(A_{(j)}, A_{(p^{(j)})})$, $swap(W_{(0)}^{(j)^T}, W_{(p^{(j)}-j)}^{(j)^T})$ ▷ Swap pivot column to current index
10: $\begin{bmatrix} a_{11}^{(j)} \\ A_{21}^{(j)} \end{bmatrix} = \begin{bmatrix} a_{11}^{(j)} \\ A_{21}^{(j)} \end{bmatrix} - Y^{(j)} W_{(0)}^{(j)^T}$ ▷ Update the current column
11: $H^{(j)} = I - y^{(j)} \tau^{(j)} y^{(j)^T}$ ▷ Compute the Householder reflection
12: $w^{(j)T} = \tau^{(j)}(y^{(j)^T} \begin{bmatrix} A_{12}^{(j)} \\ A_{22}^{(j)} \end{bmatrix} - (y^{(j)^T} Y^{(j)}) W^{(j)^T})$ ▷ Compute the accumulation
13: $R_{12}^{(j)} = A_{12}^{(j)} - Y_1^{(j)} W_2^{(j)^T}$ ▷ Update current row
14: $A^{[i+1]} = A^{[i+1]} - Y_2^{[i]} W_2^{[i]^T}$ ▷ Update the trailing matrix

Output : [Q, R, P, r]

the accumulations is needed in the panel-wise column pivoted QR since the trailing matrix is not updated at each iteration, while the column norms should be updated. Note that, the reflector and accumulation matrices are the concatenation of each column reflection vector, $Y = [y^{(0)}, y^{(1)}, ...]$, and accumulation vector, $W^T = [w^{(0)}, w^{(1)}, ...]^T$, respectively. Finally, each column iteration terminates by updating the current row of the matrix.

Let us represent the panel reflectors at the end of the i^{th} panel iteration as $Y^{[i]} = \begin{bmatrix} Y_1^{[i]} \\ Y_2^{[i]} \end{bmatrix}$ and the accumulation matrix as $W^{[i]^T} = \begin{bmatrix} W_1^{[i]^T} & W_2^{[i]^T} \end{bmatrix}$. Then, $Y_2^{[i]}$ stands for the $A_{21}^{[i]}$ part of Figure 2.3, while $W_2^{[i]^T}$ represents the contributions from $A_{12}^{[i]}$. In other words, the left-most and the upper yellow parts of Figure 2.6 illustrate $Y_2^{[i]}$ and $W_2^{[i]^T}$, respectively. In Algorithm 4, every time the inner loop ends, the computation of $R_{11}^{[i]}$, $R_{12}^{[i]}$, $Y^{[i]}$ and $W^{[i]}$ is completed. Then, at each panel iteration, the whole trailing matrix is updated by using the contributions from these yellow parts as it has an inter-panels right-looking approach.

Note that all the methods in this chapter use intra-panel left-looking approach.

Therefore, from now on, the algorithms will be called left-looking or right-looking according to their inter-panel update features.

Observe that Line 12 in the panel-wise column pivoted QR algorithm (Algorithm 4) requires large level-2 BLAS operations, i.e. $y^{(j)^T} \begin{bmatrix} A_{12}^{(j)} \\ A_{22}^{(j)} \end{bmatrix}$ and $y^{(j)^T} Y^{(j)}$. These level-2 BLAS operations are inefficient because of the memory passes for large matrices at each column iteration. Therefore, we will explain a technique to avoid these inefficient memory passes in the next section.

The QRCP method reduces the complexity compared to SVD (with the complexity $\mathcal{O}(n^3)$) to $\mathcal{O}(n^2 r)$. Here, the pivot selection operation costs $\mathcal{O}(n^2 r)$, while applying the pivot is as cheap as simple column swaps. As QRCP is right-looking inter-panels, the whole trailing matrix updates are performed at a cost of $\mathcal{O}(n^2 r)$.

2.4.2 Randomized QR with column pivoting (RQRCP)

As mentioned in the previous section, the need of pivoting leads to many inefficient level-2 BLAS operations in the panel-wise QRCP factorization, which reduces the performance for large matrices. In this respect, some randomization methods are proposed to use a lower dimensional representative matrix, instead of the original one, to determine the pivot columns of the panels. For this purpose, a Gaussian matrix with independent and identically distributed random variables (Gaussian i.i.d.), Ω, can be used to project the original matrix onto the sample matrix, B, as

$$B_{d \times n} = \Omega_{d \times m} A_{m \times n},$$

where $d \ll m$. Here, we need to get the representative data with small uncertainity to be able to determine the correct pivoting for the current panel. That is, let us consider the j^{th} column of A, $a_{(j)}$, and the corresponding sample matrix column $b_{(j)}$. Then, $\frac{||b_{(j)} - b_{(i)}||_2^2}{d||a_{(j)} - a_{(i)}||_2^2} - 1 \leq \tau$, with $0 < \tau < 0.5$ being the threshold and $0 \leq i, j < n$, should be satisfied with high probability [DG15]. This is why we do not choose the row dimension of B as block size b. Instead, we use a pre-defined oversampling size, p. Then, the row dimension is $d = b + p$. In [XGL17], it is shown that the failure probability of RQRCP is exponentially decreasing for larger oversampling sizes. Although we need this parameter for the quality of the sampling, we should be careful to choose it as small as possible, while respecting the error bounds, to efficiently improve the performance of the randomized kernel.

Note that we have to resample the matrix at each panel iteration in the classical approach to be able to determine the panel pivots. However, it is too expensive in practice. Instead, we can update the sample matrix in a cheaper way and use it at each iteration as proposed in [DG15, MQOHvdG17]. In these works, the authors prove that the performance results of these randomized techniques become close to the unpivoted QR methods, while the accuracy is very close to the one of unrandomized methods.

Therefore, in this section and in Section 2.4.3, we adopt the randomized methods through the updated sample versions as in these works.

As explained in [DG15], considering the block factorization of the matrix B

$$BP^{[i]} = \Omega A P^{[i]} = Q_B^{[i]} \begin{bmatrix} S_{11}^{[i]} & S_{12}^{[i]} \\ 0 & S_{22}^{[i]} \end{bmatrix},$$

the update of the sample matrix can be computed as

$$\begin{bmatrix} B_1^{[i]} \\ B_2^{[i]} \end{bmatrix} = \Omega^{[i]} A^{[i]} = \begin{bmatrix} S_{12}^{[i]} - W_{11}^{[i]} R_{12}^{[i]} \\ S_{22}^{[i]} \end{bmatrix} = \begin{bmatrix} S_{12}^{[i]} - S_{11}^{[i]} {R_{11}^{[i]}}^{-1} R_{12}^{[i]} \\ S_{22}^{[i]} \end{bmatrix}.$$

Algorithm 5 Randomized QR with Column Pivoting

function RQRCP(A, ϵ, b)
  $\Omega^{[0]}$ ▷ Generate the Gaussian i.i.d. matrix
  $B^{[0]} = \Omega^{[0]} A^{[0]}$ ▷ Compute the sample matrix
  for $i = 0, 1, ..., \ulcorner \frac{min(m,n)}{b} \urcorner - 1$ **do**
    $[Q_B^{[i]}, S^{[i]}, P_B^{[i]}, r] = QRCP(B^{[i]}, \sqrt{(b+p)}\epsilon, b)$ ▷ Compute permutation
    $A = AP_B^{[i]}$ ▷ Apply the permutation
    $[Q^{[i]}, R_{11}^{[i]}] = QR(A^{[i]})$ ▷ Apply QR on the permuted matrix A
    $\begin{bmatrix} R_{12}^{[i]} \\ A^{[i+1]} \end{bmatrix} = {Q^{[i]}}^T \begin{bmatrix} A_{12}^{[i]} \\ A_{22}^{[i]} \end{bmatrix}$ ▷ Update rightmost matrix
    $B_1^{[i+1]} = S_{12}^{[i]} - S_{11}^{[i]} {R_{11}^{[i]}}^{-1} R_{12}^{[i]}$ and $B_2^{[i+1]} = S_{22}^{[i]}$ ▷ Update the sample matrix
  Output : $[Q, R, P, r]$

Now, let us observe the randomized QRCP method, with the sample matrix update formula, in Algorithm 5. Here, the stopping criterion is applied implicitly through the QRCP function, which is used for computing the permutation matrix P_B through the sample matrix. As the permutation is computed through the sample matrix, the threshold should also be arranged according to B. As explained in [DG15], the column norms of B and A have a constant relative ratio through the Chi-squared distribution [Dev12]. As a consequence, their residual error ratio is approximately $\sqrt{\frac{1}{b+p}}$, with $b+p$ being the row dimension of the sample matrix. Therefore, assuming ϵ is the threshold for the matrix A, the threshold for the sample matrix B is computed as

$$\epsilon_B = \sqrt{(b+p)}\epsilon.$$

As seen in the algorithm, RQRCP is similar to QRCP (Algorithm 4), except for the computation of the permutation matrix and the sample matrix generation/update

operations. Then, in this algorithm, the additional cost to QRCP is the cost of generating the sample matrix $B_{(b+p)\times n}$ in the beginning, $\mathcal{O}(n^2 b)$, as well as its update at all iterations, $\mathcal{O}(nrb)$. However, the pivot finding operation using the input matrix A, $\mathcal{O}(n^2 r)$, is avoided. The pivoting operation through the sample matrix reduces this cost to $\mathcal{O}(nrb)$. Through these computational complexities, we can observe that the randomization method does not aim to reduce the theoretical complexity compared to QRCP. Instead, it targets to reduce the time consuming data movements that occur when computing the column norms through large matrices.

2.4.3 Truncated randomized QR with column pivoting (TQRCP)

As explained in Section 2.1.2, the left-looking feature aims to reduce the complexity by avoiding the whole trailing matrix updates. In [DG15], the authors propose the left-looking version of the RQRCP method for this purpose. In this section, we study this left-looking randomized method, which is called truncated randomized QRCP (TQRCP).

Algorithm 6 Truncated Randomized Block QR with Column Pivoting

1: **function** TQRCP(A, ϵ, b)
2: $\Omega^{[0]}$ ▷ Generate the Gaussian i.i.d. matrix
3: $B^{[0]} = \Omega^{[0]} A^{[0]}$ ▷ Compute the sample matrix
4: **for** $i = 0, 1, ..., \ulcorner \frac{min(m,n)}{b} \urcorner - 1$ **do**
5: $[Q_B^{[i]}, S^{[i]}, P_B^{[i]}, r] = QRCP(B^{[i]}, \sqrt{(b+p)}\epsilon, b)$ ▷ Compute permutation
6: $A = AP_B^{[i]}$ and $W_G^{[i]^T} = W_G^{[i]^T} P_B^{[i]}$ ▷ Apply the permutation
7: $\begin{bmatrix} A_{11}^{[i]} \\ A_{21}^{[i]} \end{bmatrix} = \begin{bmatrix} A_{11}^{[i]} \\ A_{21}^{[i]} \end{bmatrix} - \begin{bmatrix} Y_{G1}^{[i]} \\ Y_{G2}^{[i]} \end{bmatrix} W_{G1}^{[i]^T}$ ▷ Update current panel
8: $[Q^{[i]}, R_{11}^{[i]}, Y^{[i]}] = QR(\begin{bmatrix} A_{11}^{[i]} \\ A_{21}^{[i]} \end{bmatrix})$ ▷ Apply QR on the current panel
9: $W^{[i]^T} = T^{[i]^T}(Y^{[i]^T} A - (Y^{[i]^T} Y_G^{[i]}) W_G^{[i]^T})$ ▷ Compute the accumulations
10: $R_{12}^{[i]} = A_{12}^{[i]} - Y_{G1}^{[i]} W_{G2}^{[i]^T}$ ▷ Update the row panel
11: $B_1^{[i+1]} = S_{12}^{[i]} - S_{11}^{[i]} R_{11}^{[i]^{-1}} R_{12}^{[i]}$ and $B_2^{[i+1]} = S_{22}^{[i]}$ ▷ Update the sample matrix
Output : $[Q, R, P, r]$

As explained before, in the right-looking approach, we use panel-wise reflectors and panel-wise accumulations to update the whole trailing matrix. In the left-looking approach, we use global reflectors, Y_G, and global accumulations, W_G^T, to update only the current panel. In Figure 2.7, we illustrate the Y_G and W_G^T matrices of the left-looking

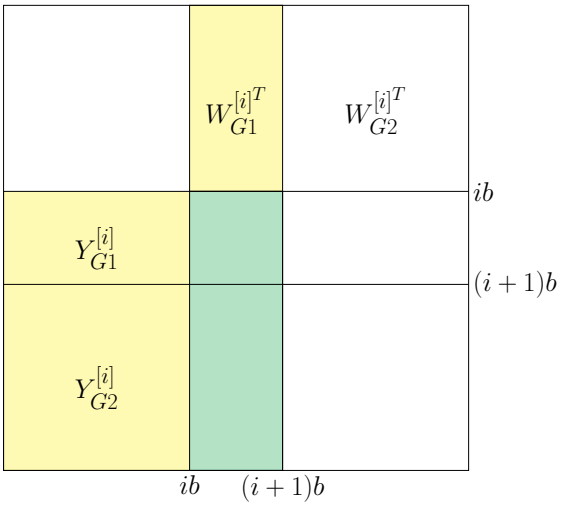


Figure 2.7: Global reflectors, Y_G, and global accumulations, W_G^T, at the beginning of the i^{th} panel iteration. Here, the yellow parts are used to update the current green panel.

approach at the beginning of the i^{th} panel iteration. Here, the yellow global reflectors and yellow global accumulations are used to update the current green panel.

In Algorithm 6, we provide the code for the left-looking TQRCP method. As explained in [DG15], TQRCP follows the same logic as converting the column-wise QRCP to the panel-wise QRCP, but at a panel-wise level. That is, let us consider the composition of all the global reflectors, $Y_G^{[i+1]}$, as

$$Y_G^{[i+1]} = \begin{bmatrix} Y_G^{[i]} & Y^{[i]} \end{bmatrix},$$

the block reflectors are updated as

$$(I - Y_G^{[i]} T_G^{[i]} Y_G^{[i]^T})(I - Y^{[i]} T^{[i]} Y^{[i]^T}) = I - Y_G^{[i+1]} T_G^{[i+1]} Y_G^{[i+1]^T},$$

where $T_G^{[i+1]}$ is

$$T_G^{[i+1]} = \begin{bmatrix} T_G^{[i]} & -T_G^{[i]} Y_G^{[i]^T} Y^{[i]} T^{[i]} \\ 0 & T^{[i]} \end{bmatrix}.$$

Then, the global accumulation matrix, which results in extra storage, for this panel-wise left-looking code is

$$W_G^{[i+1]^T} = T_G^{[i+1]^T} Y_G^{[i+1]^T} A = \begin{bmatrix} W_G^{[i]^T} \\ W^{[i]^T} \end{bmatrix} = \begin{bmatrix} T_G^{[i]^T} Y_G^{[i]^T} A \\ T^{[i]^T} (Y^{[i]^T} A - (Y^{[i]^T} Y_G^{[i]}) W_G^{[i]^T}) \end{bmatrix}.$$

As seen in the algorithm, TQRCP is indeed the panel-wise left-looking version of RQRCP. That is, after each panel iteration, RQRCP updates $\begin{bmatrix} A_{12}^{[i]} \\ A_{22}^{[i]} \end{bmatrix}$, while TQRCP stores

the contribution accumulations and only updates $A_{12}^{[i]}$. Observe that TQRCP also applies the stopping criterion implicitly, similarly to RQRCP.

Compared to RQRCP, there is an extra storage cost for the matrix W_G^T, of size $r \times n$, in the TQRCP method. However, at each panel iteration of the TQRCP method, the update of $A_{22}^{[i]}$, with size $(n-r-b) \times (n-r-b)$, is avoided. This reduces the total trailing matrix update cost to $\mathcal{O}(nr^2)$ in TQRCP compared to $\mathcal{O}(n^2 r)$ in RQRCP. The complexity of computing W_G^T in TQRCP is $\mathcal{O}(n^2 b)$. As the sample matrix generation, of complexity $\mathcal{O}(n^2 b)$, is performed only once at the beginning of the algorithm, it is not necessarily dominant since actually it has a constant 1. Then, thanks to the low cost trailing matrix updates, TQRCP has the lowest computational complexity among the four methods that we study. For large matrices with low ranks, this method highly reduces the update cost of the trailing matrix.

2.4.4 Randomized QR with rotation (RQRRT)

In this section, the rotational QR method in [Mar15] is studied. This method is proposed to fix the rank revealing issue of the column pivoted QR method on some matrices.

Remember that the rank revealing QR factorization aims to produce an approximation as close as possible to the optimal approximation (SVD) (see the approximation criterion 2.9). Then as explained in [Mar15], we can apply a convenient orthonormal matrix to A to produce a rank closer to SVD to satisfy this criterion, instead of only permuting the columns.

In the randomized QR factorization with rotation (RQRRT) method, the sample matrix is used similarly to the randomized QRCP. However, this time the original matrix, A, is not permuted through the permutation that is obtained by using QRCP of the sample matrix, B. Instead, an orthonormal matrix is generated through the unpivoted QR factorization of B^T. This is indeed an effective method to get the mass to the left of the matrix, since the sample matrix is generated in a way that it really represents the linear dependencies of the original matrix. Therefore, this orthonormal matrix represents the rotation of the input matrix when the QR factorization is applied. When this rotation matrix is applied to matrix A, its representative data moves to the diagonals to have closer results to SVD.

Algorithm 7 stands for the RQRRT compression kernel. Here, the output rotational matrix is equivalent to $Q_{rot} = Q_B^{[0]} Q_B^{[1]} Q_B^{[2]} ... Q_B^{[\lceil \frac{r}{b} \rceil - 1]}$. Let us compare this RQRRT algorithm to the randomized QRCP (Algorithm 5). Their first difference is the way of gathering the representative data. In the randomized QRCP algorithm, we find

$$AP_B = Q_A R_A,$$

where P_B is obtained through the QRCP algorithm as

$$BP_B = Q_B R_B.$$

Algorithm 7 Randomized Block QR with Rotation

1: **function** RQRRT(A, ϵ, b)
2: $\Omega^{[i]}$ ▷ Generate the Gaussian i.i.d. matrix
3: **for** $i = 0, 1, 2, ..., \ulcorner \frac{min(m,n)}{b} \urcorner - 1$ **do**
4: $B^{[i]} = \Omega^{[i]} A^{[i]}$ ▷ Resample B
5: $[Q_B^{[i]}, S^{[i]}] = QR(B^{[i]^T}, b)$ ▷ Compute the rotation matrix
6: $A^{[i]} = A^{[i]} Q_B^{[i]}$ ▷ Apply the rotation
7: $[Q^{[i]}, R_{11}^{[i]}] = QR(A^{[i]})$ ▷ Apply QR on A
8: $\begin{bmatrix} R_{12}^{[i]} \\ A^{[i+1]} \end{bmatrix} = Q^{[i]^T} \begin{bmatrix} A_{12}^{[i]} \\ A_{22}^{[i]} \end{bmatrix}$ ▷ Update the trailing matrix
9: $r = StoppingCriterion(R_{11}^{[i]}, R_{22}^{[i]}, ||A^{[0]}||_F, \epsilon, b)$ ▷ Check stopping criterion

Output : [Q, R, Q_{rot}, r]

However, in the RQRRT method, the $AQ_B^{'}$ matrix is used as

$$AQ_B^{'} = Q_A^{'} R_A^{'},$$

where the orthogonal matrix is obtained from the partial QR of the sample matrix as

$$B^T = Q_B^{'} R_B^{'}.$$

This shows the column orthonormalization of B^T. Therefore, the leftmost b column spanning the matrix A will have almost the same spanning as the leading b singular vectors of A. Therefore, it gets a closer result to SVD in terms of rank revealing quality [Mar15].

The second difference of the rotational method is the resampling operation. In the column pivoted method, the sample matrix is cheaply updated at each panel iteration. However, in the rotational method, as its transpose is used, there is no update formula (up to our knowledge).

The third difference of the rotational method is the explicitly implemented stopping criterion (at Line 9), since we do not call QRCP function anywhere in this code. Here, the input matrix $R_{22}^{[i]}$ stands for $A^{[i+1]}$.

The detailed stopping criterion in RQRRT code is given in Algorithm 8. Everytime this function is called, the trailing matrix norm on Line 2 is explicitly computed. Here, it is important to observe that at the end of the panel iteration, if the criterion (on Line 2) is satisfied, we check the previous trailing matrix norms by adding each column contribution starting from the last column of the panel (on Line 4). Bear in mind that the value $||(R_{11}^{[i]})_{(l)}||_2$ does not have to be computed explicitly since it is equal to the magnitude of the first element of this column. If we check the trailing matrix norms with the accumulations of the left-most columns first, we can lose stability because of the

round-off errors. Note that in the QRCP method, the stability issue is already taken care of and it is out of scope in this thesis.

In Figure 2.8, the stability issue is observed. Here, the result is obtained by using the *S-shape* matrix (see Section 2.6.1) of size 500×500, where the generation rank is 25. In the figure, RQRRT (right) represents our algorithm and RQRRT (left) stands for the unstable accumulation version. As a reference, the trailing matrix norms at each panel index at generation, $\sqrt{\sum_{i=K}^{n} \sigma_i^2}$, is also shown (called Reference). The y-axis stands for $||A - U_K V_K^T||_F$, while x-axis shows the indices, K, of the first panel iteration. Here, U_K is of dimension $m \times K$ and represents the K columns of the matrix U. Similarly, V_K^T stands for the first K rows of V^T.

Figure 2.8 shows that indeed the right-most accumulation gets closer results to Reference. As opposed to our approach, the left-most accumulation diverges after the index 23 supporting our statement.

Algorithm 8 Stopping criterion in RQRRT

function $StoppingCriterion(R_{11}^{[i]}, R_{22}^{[i]}, ||A^{[0]}||_F, \epsilon, b)$
  if $||R_{22}^{[i]}||_F \leq \epsilon ||A^{[0]}||_F$ **then**
    for $l = b-1, b-2, ..., 1$ **do** ▷ Accumulations starting from right-most column
      $||R_{22}^{(ib+l)}||_F = \sqrt{||R_{22}^{(ib+l+1)}||_F^2 + ||(R_{11}^{[i]})_{(l)}||_2^2}$
      if $||R_{22}^{(ib+l)}||_F > \epsilon ||A^{[0]}||_F$ **then return** $r = ib + l + 1$
Output : $[r]$

For RQRRT, finding the rotation matrix has a complexity of $\mathcal{O}(nrb)$, while applying it to the matrix A is of cost $\mathcal{O}(n^2 r)$. However, for RQRCP, the pivot finding cost is $\mathcal{O}(nrb)$ and the AP_B operation is only as cheap as swapping b columns of A. Another cost difference of RQRRT is the resampling at all iterations, $\mathcal{O}(n^2 r)$, compared to the sample matrix update cost in RQRCP, $\mathcal{O}(nrb)$. As seen here, the complexity differences between these two methods can be significant for large matrix sizes, which makes RQRRT the most costly QR method that we study.

2.5 Summary of the complexities

In this section, we aim to summary and comment on the complexities of the compression kernels studied in the previous sections. Table 2.1 illustrates the computation complexities of the compression methods for a generic matrix of size $n \times n$, with a block size b and a compression rank r. Here, only the operations leading to a difference between the methods are shown for the sake of simplicity. The third column of the table stands for the total cost of each operation, while the last column shows the overall cost of each method. When determining the values in the last column, we assume $r > b$.

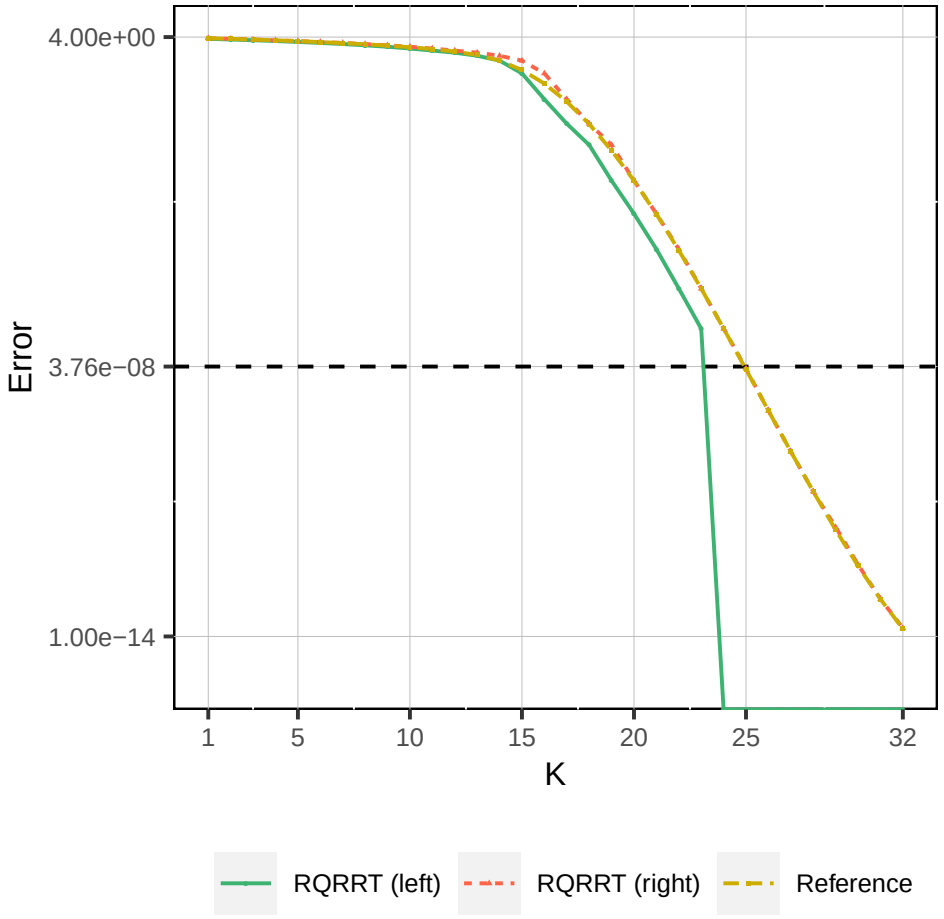


Figure 2.8: The round-off error impact when the trailing matrix norm columns are accumulated starting from right or left. The y-axis stands for $||A-U_K V_K^T||_F$ in logarithmic scale, while x-axis stands for each index in the first panel iteration. The result represents the *S-shape* matrix (see Section 2.6.1) of size 500×500, where the generation rank is 25.

Method	Operation	Total cost	Overall cost
SVD			$\mathcal{O}(n^3)$
QRCP	Pivot finding	$\mathcal{O}(n^2r)$	$\mathcal{O}(n^2r)$
	Trailing matrix update	$\mathcal{O}(n^2r)$	
RQRCP	Sample matrix generation (beginning)	$\mathcal{O}(n^2b)$	$\mathcal{O}(n^2r)$
	Pivot finding	$\mathcal{O}(nrb)$	
	Update of sample matrix B	$\mathcal{O}(nrb)$	
	Trailing matrix update	$\mathcal{O}(n^2r)$	
TQRCP	Sample matrix generation (beginning)	$\mathcal{O}(n^2b)$	$\mathcal{O}(n^2b)$
	Pivot finding	$\mathcal{O}(nrb)$	
	Update of current panel	$\mathcal{O}(nr^2)$	
	Generating accumulation matrix (W^T)	$\mathcal{O}(n^2b)$	
	Update of sample matrix B	$\mathcal{O}(nrb)$	
RQRRT	Resampling (all iterations)	$\mathcal{O}(n^2r)$	$\mathcal{O}(n^2r)$
	Rotation computing	$\mathcal{O}(nrb)$	
	Rotation of A	$\mathcal{O}(n^2r)$	
	Trailing matrix update	$\mathcal{O}(n^2r)$	

Table 2.1: Important complexity changes between the compression methods. Here, n, r and b stand for the matrix size, the compression rank and the block size, respectively.

As seen in the table, the truncated method has a total of $\mathcal{O}(n^2b)$ complexity, while all other QR variants have a $\mathcal{O}(n^2r)$ complexity. As in general $b < r$ and TQRCP reduces the trailing matrix update cost, this truncated method is the cheapest procedure for large matrices with low ranks, in a sequential environment. Since SVD has a $\mathcal{O}(n^3)$ complexity, it is the most costly method, as expected.

For the QRCP, RQRCP and RQRRT methods, the overall complexity is asymptotically the same. However, when each operation complexity is observed, one can see that the usage of the rotational feature in RQRRT introduces large additional costs compared to the other two methods. That is why, it is the most costly QR variant. In RQRCP, the pivoting cost is reduced compared to QRCP. However, there are additional costs for sample matrix generation (in the beginning) and for updating it at each iteration. Thus, this method is advantageous for large matrices, where mostly the communication of the column norms computations are costly. To conclude this table, we expect in our experiments to observe the performance order as (from worst to best, in a sequential environment): $SVD \ll RQRRT < RQRCP < QRCP \ll TQRCP$.

2.6 Experiments on generated matrices

In this section, we compare the compression methods on different generated matrices, including challenging cases. For the sake of simplicity, the row and column dimensions are chosen to be equal. From now on, the letter r_G will represent the generation rank of

the matrices, whereas r represents the approximation rank of them.

As a matter of interest, the compression kernels are in sequential within the PaStiX framework. Therefore, the results in this section demonstrates the sequential results, since parallelism is not worth for the matrix sizes we are interested in. For the RQRCP and TQRCP methods, the block size is set as $d = b + p$, where $b = 27$ and the oversampling parameter is set to $p = 5$. For the other methods, the block size is $d = b = 32$. Let us emphasize that we implement all the compression methods in a similar fashion and we determine the compression ranks numerically. Here, let us also note that setting $b = 32$ and $p = 5$ would be a better choice when comparing RQRCP and TQRCP to the other methods, where $d = b = 32$. However, it will not affect our results observably.

In Section 2.6.1, we first provide the information about how we generate the experimental matrices. Then, in Sections 2.6.2, 2.6.3, 2.6.4 and 2.6.5, we discuss the stability, performance, numerical ranks and accuracy results on these matrices, respectively.

2.6.1 Generation of the experimental matrices

In this section, five different type of generated matrices are used. These matrices are similar to the ones in [Mar15], and are shown in Figure 2.9, Figure 2.10 and Figure 2.11. In these figures, the x-axis stands for the matrix index, K. The figures on the left illustrate the singular value distribution of a matrix of size 500×500, where the generation rank is 100 at the generation precision $\epsilon = 10^{-8}$. The figures on the right stand for the relative Frobenius norm of the trailing matrix, $\frac{\sqrt{\sum_{i=K}^{n} \sigma_i^2}}{||A||_F}$, of the same matrices. The solid black lines in the figures represent the generation precision.

The first matrix type is called *k-rank* and it is observed in Figure 2.9. In order to generate this matrix, a normalized Gaussian i.i.d. matrix is used, where the singular values decrease slowly and suddenly drop to ignorable values at the generation rank. Because of the sudden drop to ignorable values at r_G, this matrix type is trivial for the compression methods.

The second and third matrix types are called *Z-shape* and *Z-shape-short*, which are shown in Figure 2.10. Both of these matrices are computed as $A = UDV^T$, where U and V are random orthonormal matrices. The singular values (diagonal values of the matrix D) are the same for both cases and decrease rapidly up to the generation rank. However, the *Z-shape* drops faster to ignorable values at the index $\frac{3}{2}r_G = 150$, while *Z-shape-short* stagnates at 10^{-9}. Here, we expect the methods to find close approximation ranks to $r_G = 100$ for the *Z-shape* because of the fast drop after the generation rank. Opposite to *Z-shape*, as seen in the right-most figure, the trailing matrix of the *Z-shape-short* matrix contains important data at r_G with respect to the compression precision 10^{-8}. Thus, this case is challenging for our stopping criterion to determine the compression rank close to the generation one.

The last matrix types are called *S-shape* and *S-shape-short*, which are shown in Figure 2.11. These matrices are computed as $A = UDV^T$, with random orthonormal matrices U and V. For these matrices, the singular values stagnate at 1 up to the index

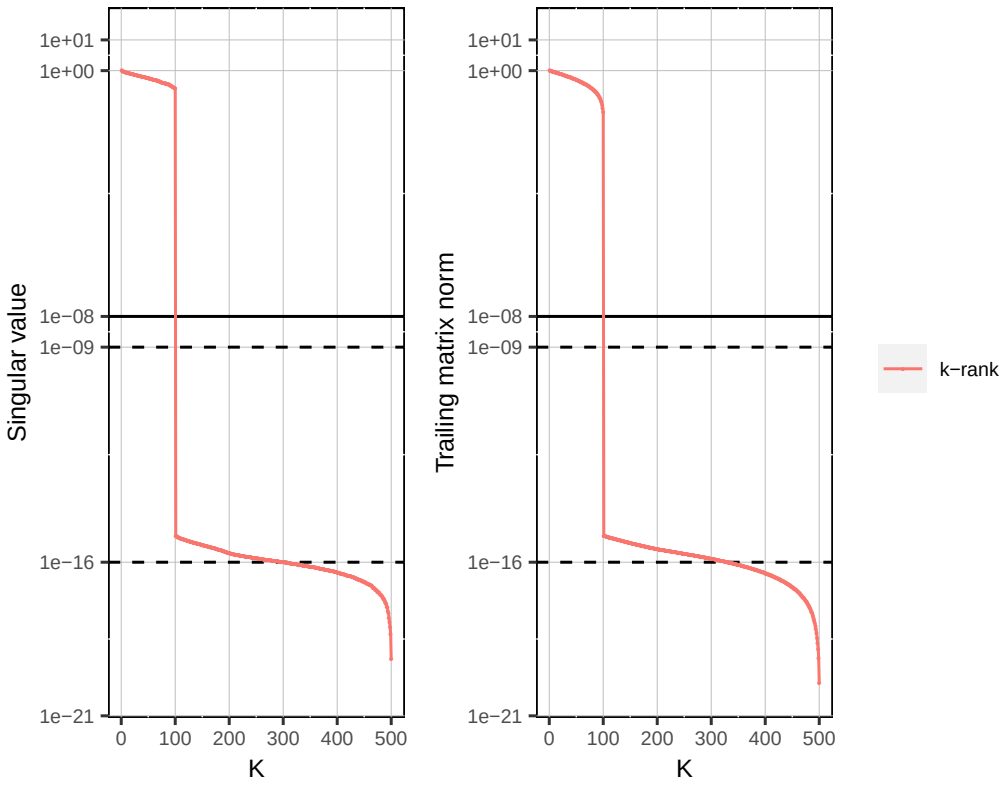


Figure 2.9: The *k-rank* matrix of size 500×500, where the generation rank is $r_G = \frac{20n}{100} = 100$ at the generation precision $\epsilon = 10^{-8}$. The Frobenius norms of the trailing matrices at each index ($\frac{\sqrt{\sum_{i=K}^{n} \sigma_i^2}}{||A||_F}$) are on the right, while the singular values at each index (σ_K) are on the left. The x-axis stands for the matrix indices, K. The singular values firstly decay slowly, then suddenly drop to very low values.

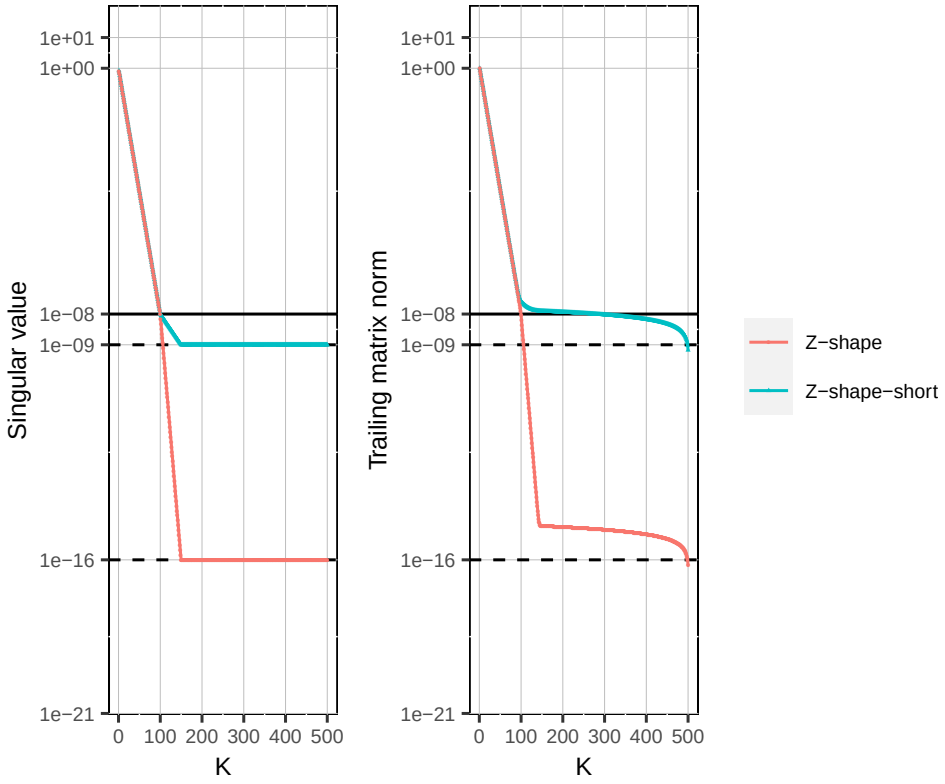


Figure 2.10: The *Z-shape* and *Z-shape-short* matrices of size 500×500, where the generation rank is $r_G = \frac{20n}{100} = 100$ at the generation precision $\epsilon = 10^{-8}$. The Frobenius norms of the trailing matrices at each index $(\frac{\sqrt{\sum_{i=K}^{n} \sigma_i^2}}{||A||_F})$ are on the right, while the singular values at each index (σ_K) are on the left. The x-axis stands for the matrix indices. Both matrices have the same singular values up to the index $r_G = 100$. However, after this point, *Z-shape* case singular values decrease faster and reach to almost machine precision at the index $\frac{3}{2}r_G$, whereas the *Z-shape-short* case singular values reach to $\frac{\epsilon}{10} = 10^{-9}$ at this index.

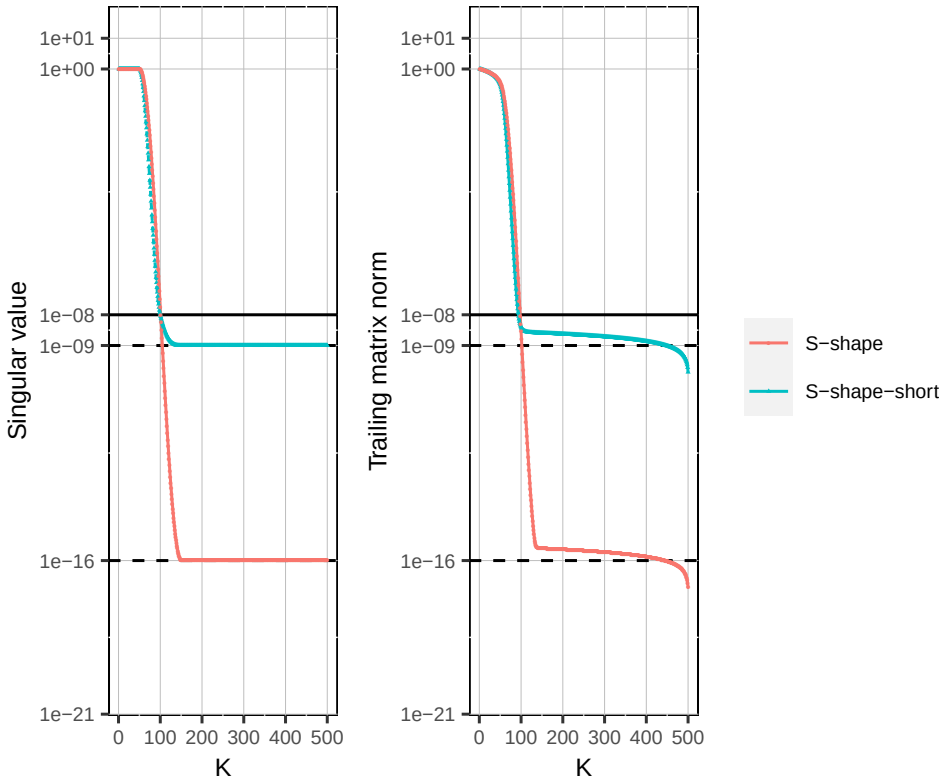


Figure 2.11: The *S-shape* and *S-shape-short* matrices of size 500×500, where the generation rank is $r_G = \frac{20n}{100} = 100$ at the generation precision $\epsilon = 10^{-8}$. The Frobenius norms of the trailing matrices at each index ($\frac{\sqrt{\sum_{i=K}^{n} \sigma_i^2}}{||A||_F}$) are on the right, while the singular values at each index (σ_K) are on the left. The x-axis stands for the matrix indices. The singular values of the *S-shape* and *S-shape-short* matrices firstly stagnate at the value 1, then they decrease fast and in the end they stagnate again after a heavy tail. Both of the singular values reach the value $\epsilon = 10^{-8}$ at the index $r_G = 100$. However, the last stagnation point is $\epsilon^2 = 10^{-16}$ for the *S-shape* matrix, whereas it is $\frac{\epsilon}{10} = 10^{-9}$ for the *S-shape-short* matrix.

$\frac{r_G}{2}$, and then they drop rapidly to reach $\epsilon = 10^{-8}$ at index r_G. After that point, the *S-shape* case stagnates at the value $\epsilon^2 = 10^{-16}$, while *S-shape-short* stagnates at $\frac{\epsilon}{10} = 10^{-9}$, which is close to the generation precision. These matrices are challenging experiments for the randomization methods as they have a heavy tail.

In the following sections, we use the compression precision $\epsilon = 10^{-8}$, which is equal to the generation precision. We observe the SVD method with two different stopping criteria. One of them uses the Frobenius norm of the trailing matrix to determine the compression rank as in 2.7, while the other uses the spectral norm for this purpose as in 2.6. The former, SVDF, provides better comparison to the QR methods, whereas the latter, SVD2, leads to closer ranks to the generation ones.

2.6.2 Stability

In this section, we observe the stability of the compression kernels. In Figure 2.12, the relative residual with respect to the initial matrix, $\frac{||A-U_K V_K^T||_F}{||A||_F}$, is observed at each index K. As these values show the approximated values of the relative Frobenius norms that were presented on the right side of Figures 2.9, 2.10 and 2.11, we want to obtain close results to them.

As seen in Figure 2.12, all the methods have good stability. As RQRCP and TQRCP use a relative stopping criterion, which uses an approximation ratio between the original and sample matrix columns, there are more magnitude changes between the panel iterations. However, it will be shown in Section 2.6.5 that these methods still ensure a good accuracy, so this does not indicate a problem. As we can see in the figures, the RQRRT method indeed obtains very close results to SVD.

Another interesting point in these figures is the black horizontal line, which stands for the compression precision $\epsilon = 10^{-8}$. That is, observe that our stopping criterion (see Section 2.1) compares the y-axis values in Figure 2.12 to the ϵ value to determine the approximation rank. Therefore, whenever a residual value reaches this black line in the figures, we know that the compression rank is the corresponding index. Therefore, through these stability figures, we observe that for the *Z-shape-short* case, all the methods get much larger compression ranks than $r_G = 100$. Thus, this case is challenging for our numerical stopping criterion. However, for all the other cases, all the methods reach close approximation ranks to the generation one.

In Figure 2.13, the same information as in Figure 2.12 is shown. However, here the relative residual with respect to the SVDF is presented to better observe the stability compared to the SVD method.

The main observation in Figure 2.13 is that QRCP, RQRCP and TQRCP can have a relative error around 2.5× larger than the one of the SVD around the index r_G. However, this relative error is very close to SVDF for the rotational QR variant, which confirms that indeed RQRRT provides closer quality approximations to SVD.

Note that for the *k-rank*, *Z-shape* and *S-shape* test cases, the last values are around the machine precision. Therefore, the extreme relative ratios at large indices are not representative, so we dropped them in the figure.

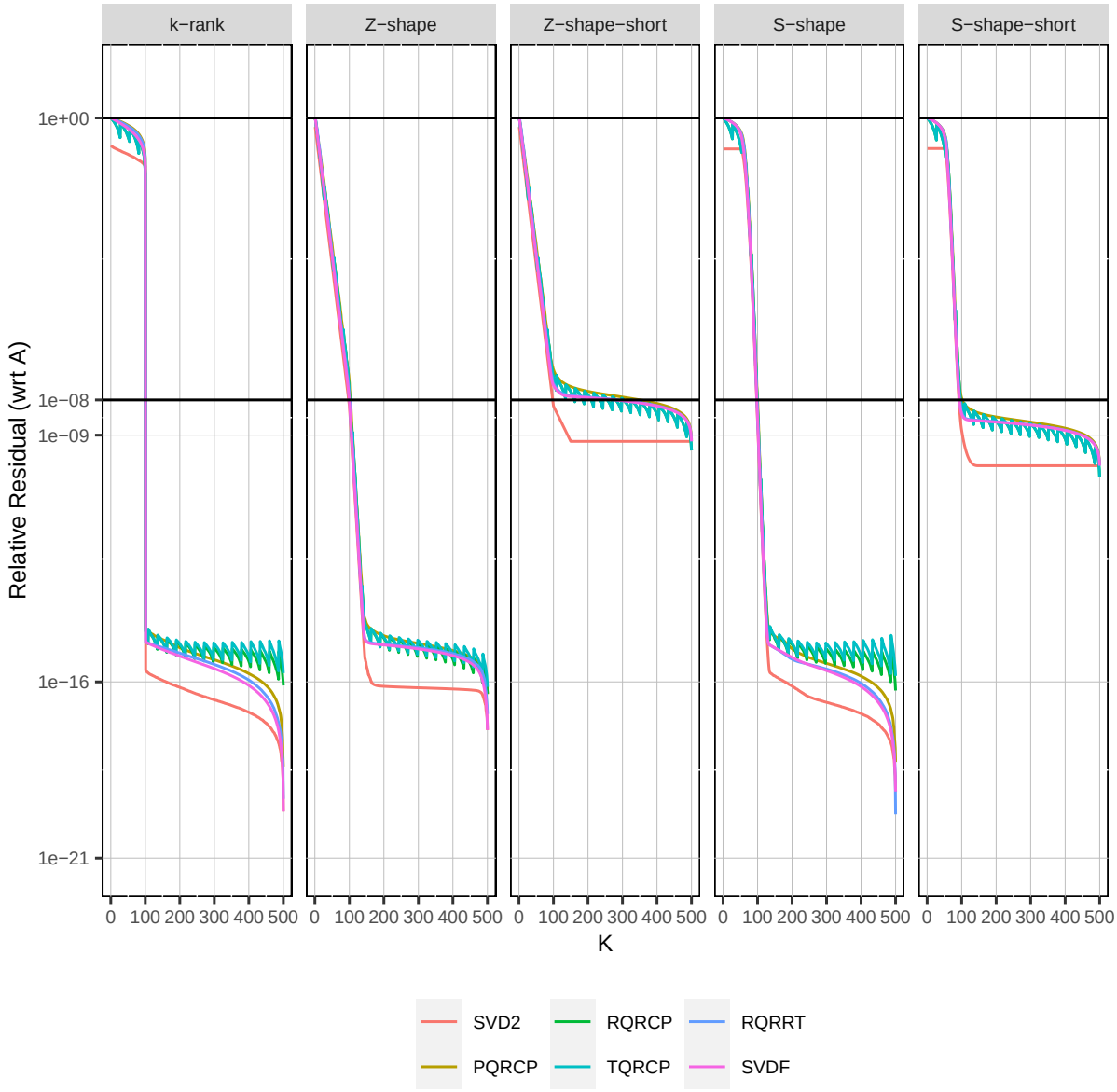


Figure 2.12: In the stability figures, the relative residual value with respect to the initial matrix, $\frac{||A-U_K V_K^T||_F}{||A||_F}$, at each index K is illustrated.

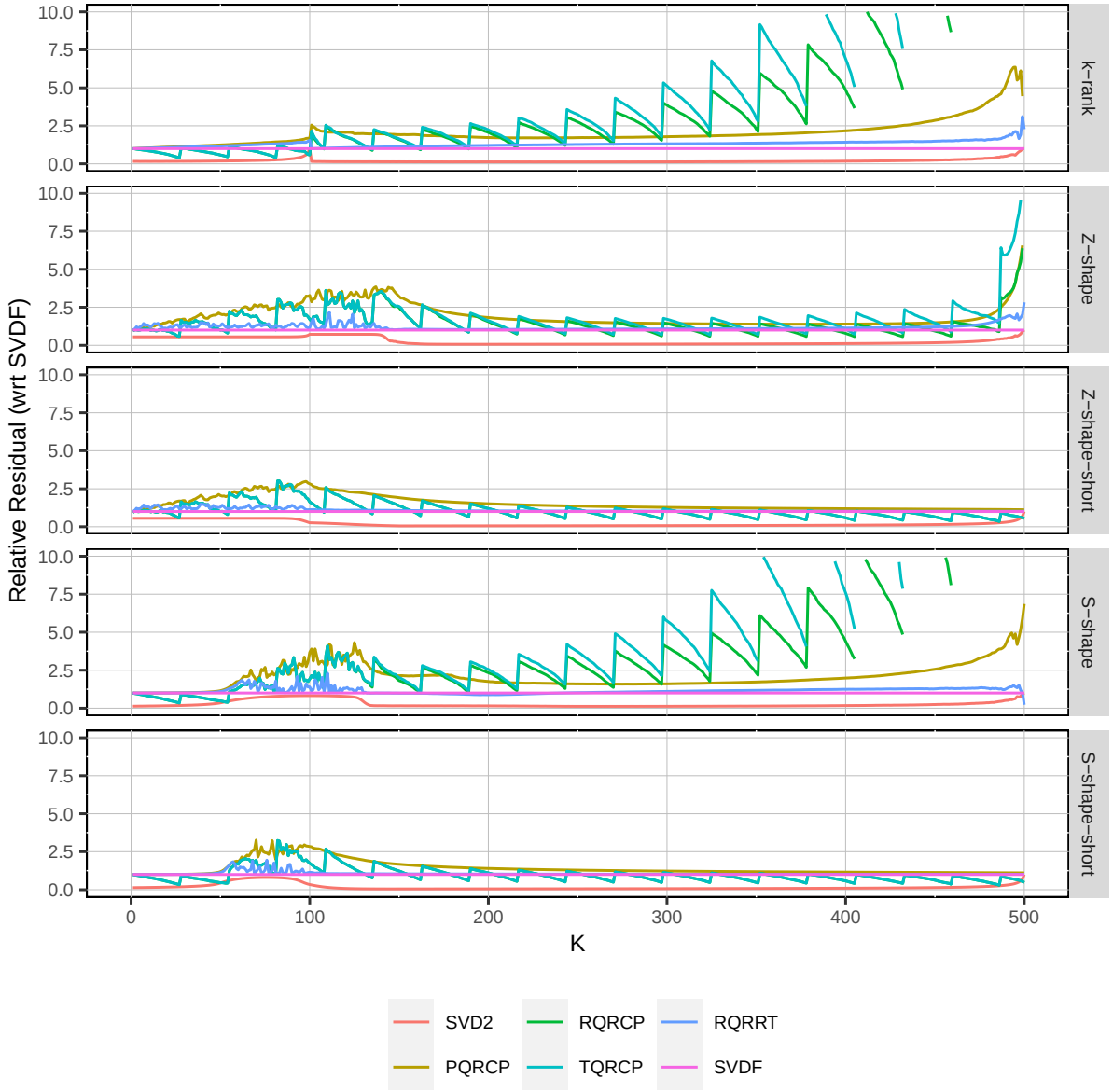


Figure 2.13: In the relative stability graphs, the relative residual value with respect to the SVDF, $\frac{||A-U_K V_K^T||_F}{||A-U_K V_K^T||_F^{SVDF}}$, at each index K is illustrated.

2.6.3 Performance

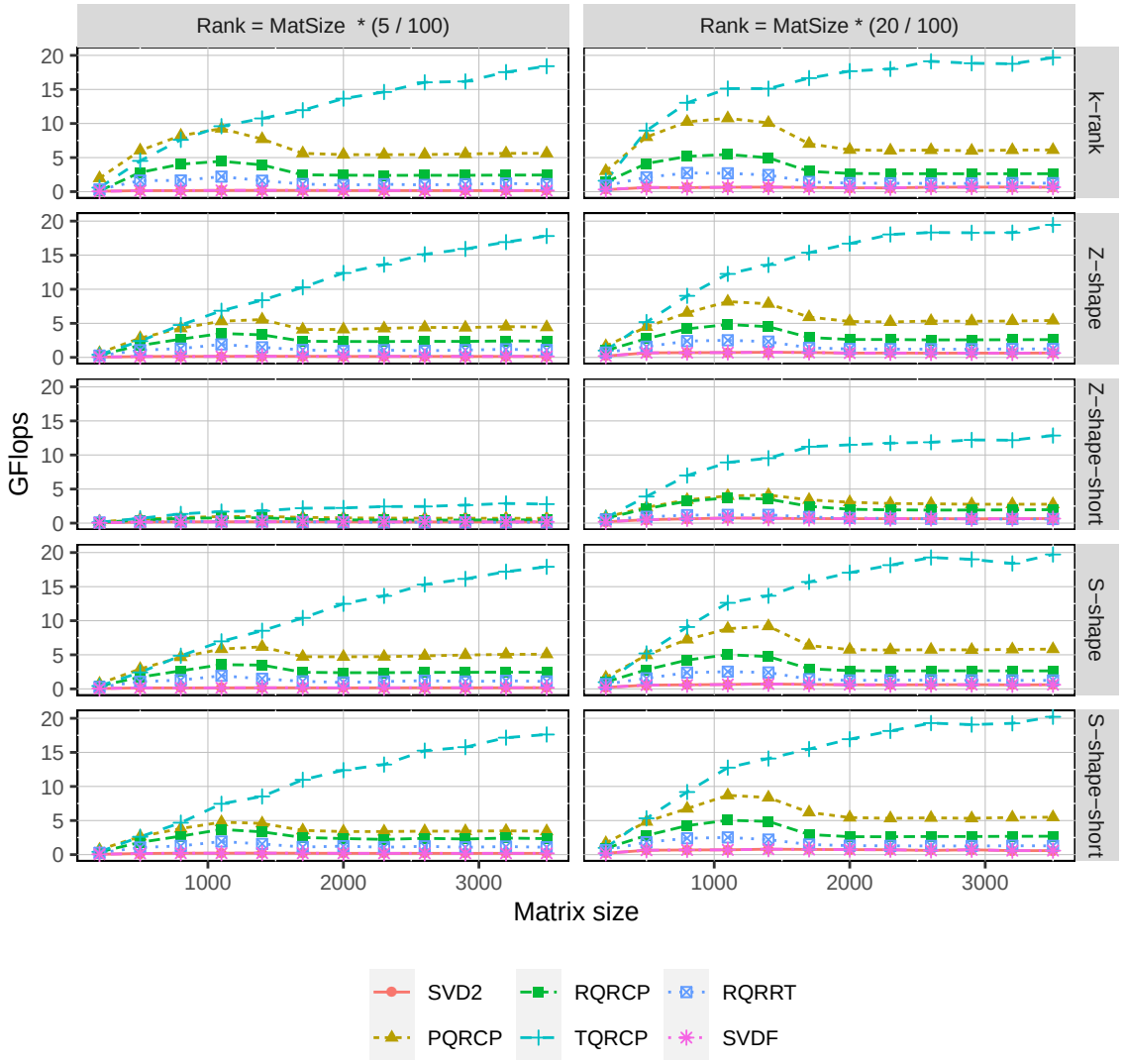


Figure 2.14: In the performance figures, $GFlops = \frac{MinGFlop}{t}$ is observed for different matrix sizes and r_G values. In this figure, each row illustrates a different test case, whereas each column represents a different generation rank. Here, we applied the stopping criterion within the compression methods.

In this section, we observe the performance results of the generated matrices that are explained in Section 2.6.1. In Figure 2.14, each test case is observed with different matrix sizes and different generation ranks.

In the figure, every grid row shows a different test case, while the columns stand for different generation ranks. The y-axis of the figures stands for $GFlops = \frac{MinGFlop}{t}$. Here,

t (in seconds) represents the time to compress the matrix with the compression rank r for each method, whereas $MinGFlop$ stands for the minimum number of GFlop to compress the matrix of size $n \times n$ and rank r. We compute it as

$$MinGFlop = GFlop_{GEQRF}(n, r) + GFlop_{UNMQR}(n, n - r, r) + GFlop_{UNGQR}(n, r, r).$$

Here, the $GEQRF$, $UNMQR$ and $UNGQR$ names are adopted from LAPACK. The $GFlop_{GEQRF}(n, r)$ represents the number of flops to perform a QR factorization on a $n \times r$ matrix. $GFlop_{UNMQR}(n, n - r, r)$ stands for the flops to apply the Q to the V part of size $r \times n$ of the approximation. Finally, $GFlop_{UNGQR}(n, r, r)$ represents the number of flops required to generate the final $n \times r$ matrix U from the Q matrix in the Householder form.

As seen in Figure 2.14, TQRCP has closer flops to the minimal flops and has always the best performance (up to 20 GFlops) for large matrix sizes, which proves the effectiveness of the left-looking approach in sequential environments.

On average, QRCP reaches the second best performance in the figures, as expected, while it has even the best performance for small matrix sizes. This is an interesting result since the randomization technique in RQRCP was introduced to increase the performance of QRCP by reducing the slow memory passes. However, since the tests run in sequential with matrix dimensions lower than 3500 and RQRCP introduces more computational complexity compared to QRCP, it is not an unexpected result. If the experiments were performed in a parallel environment for larger matrices, RQRCP would have better performance than QRCP.

As expected, RQRRT has the worst performance compared to other QR variants because of its high complexity. However, SVD has the worst performance overall since no matter which stopping criterion is used, this method should apply the full decomposition to be able to determine the compression rank.

The important point in the figures is that the QRCP method has a better performance than TQRCP for small matrix sizes, as mentioned before. However, for the larger sizes TQRCP is the best. Therefore, tuning the methods according to the matrix size is also important since there is not an absolute best method.

Another important point is that in the stability section, we have seen that for the test cases, all the compression ranks are close to the generation rank, except for the *Z-shape-short* case. This causes a significant performance reduction for all the methods through this test case. Especially, for the small ranks, the performance loss is very critical for the *Z-shape-short* case. However, we can also observe that even if the performance is reduced significantly, the performance order of the methods is still the same as the other test cases, which shows a consistency among all the figures.

2.6.4 Compression ranks

In this section, we observe the compression rank comparison of the methods for the five generated matrices we introduced. In Figure 2.15 and Figure 2.16, the relative rank

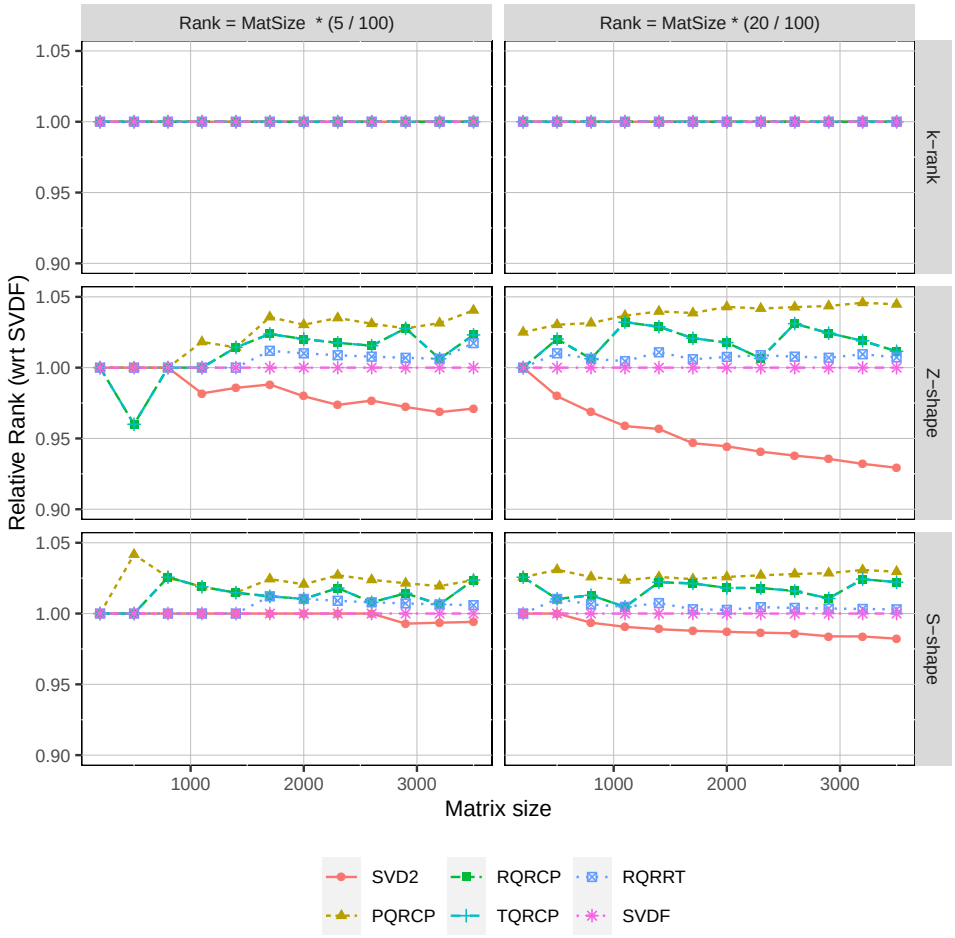


Figure 2.15: In the relative rank graphs, the relative rank with respect to SVDF, $\frac{r}{r_{SVDF}}$, for different matrix sizes and r_G values is observed. In this figure, each row illustrates a different test case, whereas each column represents a different generation rank. Here, the stopping criterion is applied in the compression methods.

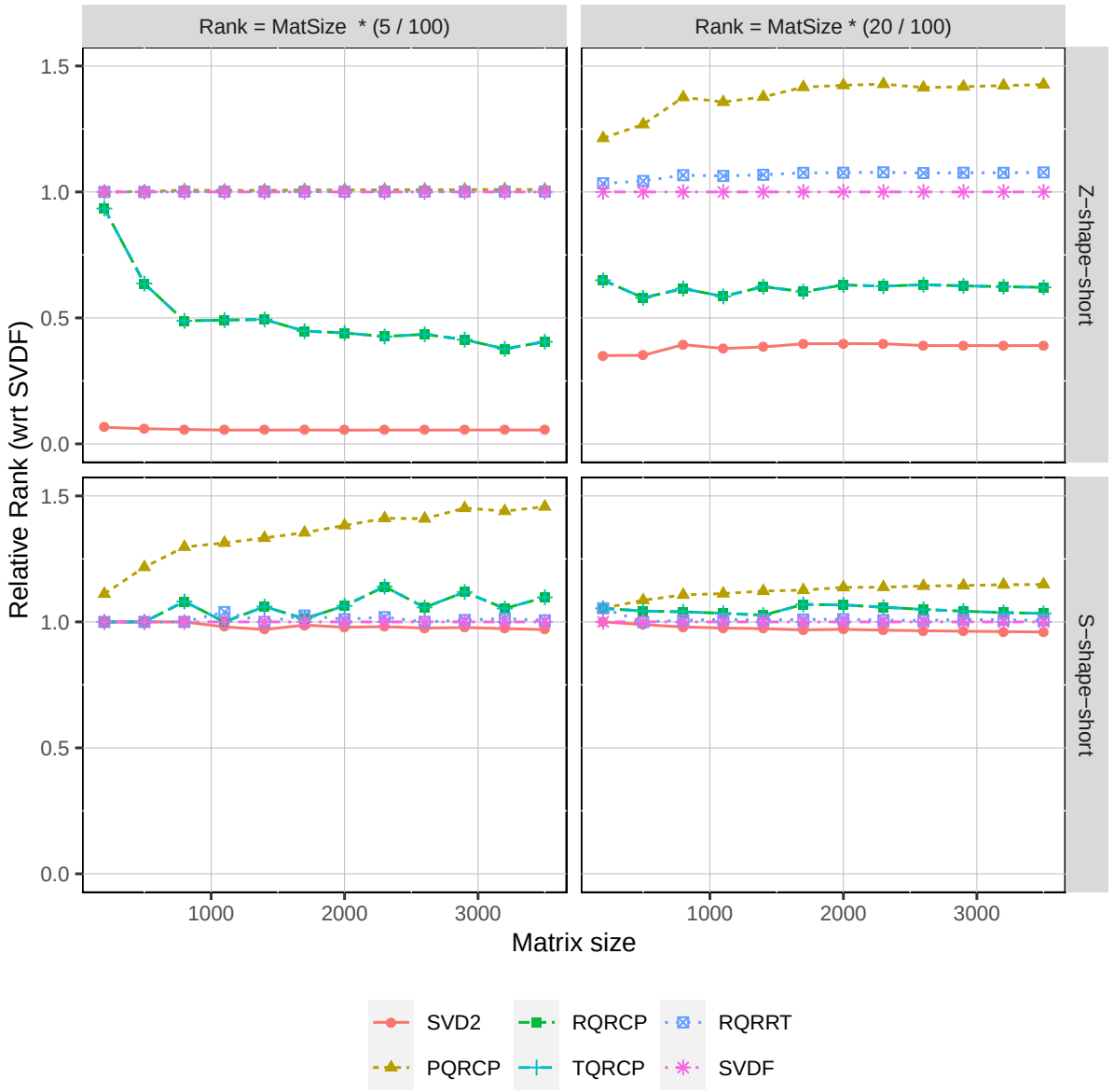


Figure 2.16: In the relative rank graphs, the relative rank with respect to SVDF, $\frac{r}{r_{SVDF}}$, for different matrix sizes and r_G values is observed. In this figure, each row illustrates a different test case, whereas each column represents a different generation rank. Here, we applied the stopping criterion within the compression methods.

with respect to SVDF, $\frac{r}{r_{SVDF}}$, is observed for different matrix sizes.

As expected, all the methods reach to the generation rank for the *k-rank* case. On the other hand, for the *Z-shape-short* and *S-shape-short* cases, the rank differences are higher since these cases have large trailing matrix norms after the generation rank as observed in Figures 2.10 and 2.11, respectively. That is, they are more challenging for our numerical stopping criterion, which uses the Frobenius norm of the trailing matrix. As these test cases result in more rank differences than other cases, we show them separately in Figure 2.16.

In the figures, the lowest possible ranks are reached by SVD2 since it uses a more strict stopping criterion than the ones that adopt the Frobenius norm of the trailing matrix.

As the rank differences are higher for the tricky case *Z-shape-short*, we can easily observe that RQRCP and TQRCP reach closer results to SVD2 compared to all other methods (including SVDF). This is caused by the higher residual norm differences within the panels for RQRCP and TQRCP as seen in Figure 2.12. The high residual norm differences can be related with the stopping criterion used in RQRCP and TQRCP, which is based on a sample matrix, whereas other methods use the original matrix norms to terminate. In addition, this difference can be related to the accumulated round-off effects of small numbers. It is also interesting to observe that the round-off effects of small numbers lead RQRRT to not always reach to the closest results to SVDF. However, as we will observe in Figure 2.17, all the methods provide a good accuracy since we work with fixed precision.

2.6.5 Accuracy

In this section, we observe the accuracy of the compression methods through our numerical stopping criterion. In Figure 2.17, the error, $\frac{||A-U_r V_r^T||_F}{\epsilon||A||_F}$, for different matrix sizes and r_G values is observed.

As seen in the figure, since for the *k-rank* case it is very trivial to reach to the generation rank, there is no observable error difference between the compression methods.

In the previous section, we observed that RQRCP and TQRCP tend to have smaller ranks than QRCP. Therefore, in Figure 2.17, they tend to have more data in the trailing matrix after the compression. Similarly, since SVD2 finds the smallest ranks in general, the data in the trailing matrix (error) tends to be more than the other methods.

When tackling very small numbers through the stopping criterion within the RQRRT method, the resulting round-off errors can slightly affect the approximation rank and error. As observed in the figure, this does not even cause one digit accuracy changes for this method, since our numerical stopping criterion guarantees a good accuracy. Therefore, we do not observe a noteworthy accuracy difference between the compression methods.

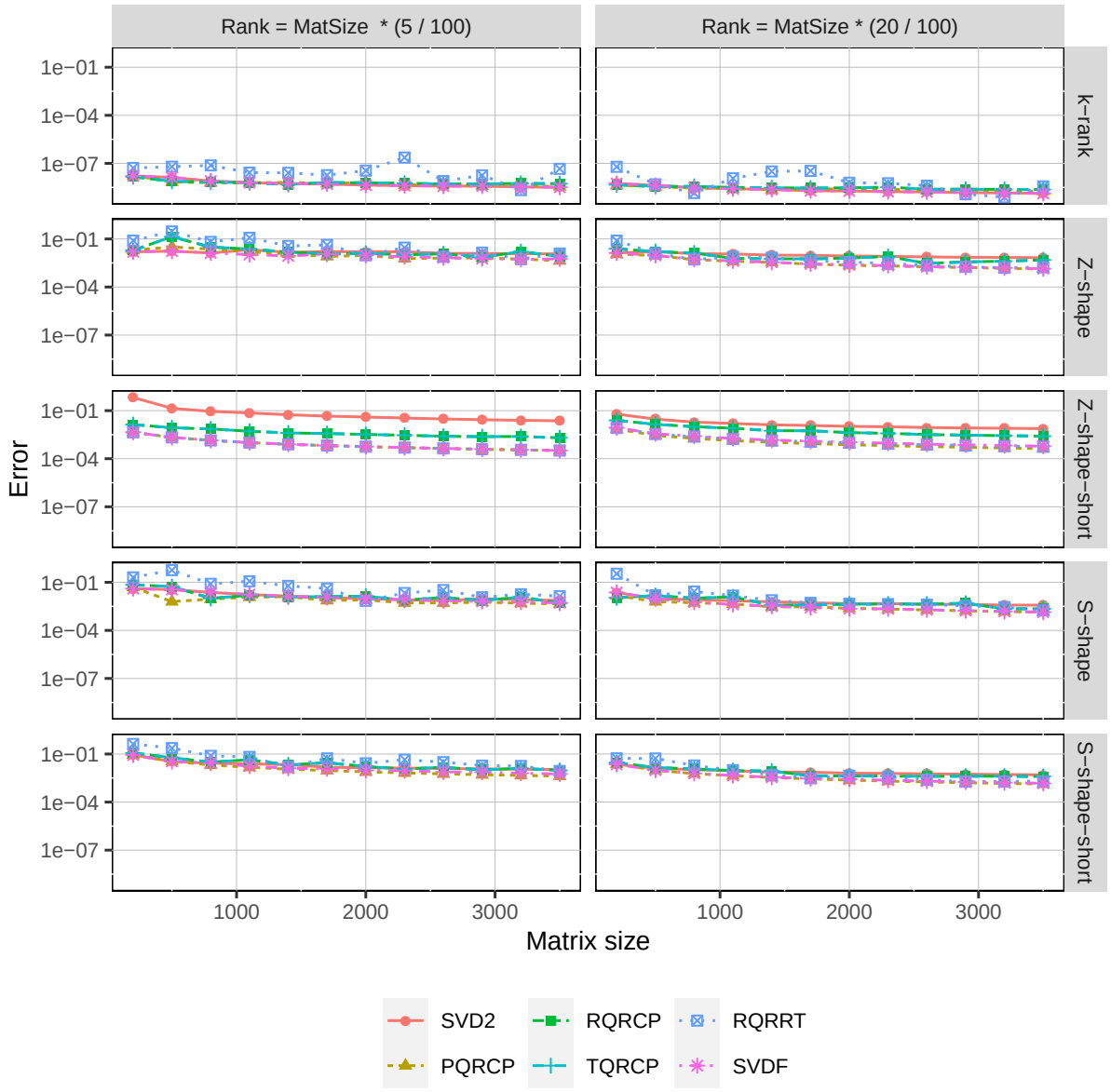


Figure 2.17: In the accuracy figures, the error, $\frac{||A-U_r V_r^T||_F}{\epsilon||A||_F}$, for different matrix sizes and r_G values is observed. Here, we applied the stopping criterion within the compression methods.

2.7 Experiments on real-life case matrices

In this section, we compare the compression methods through 810 real-life case matrices. These matrices are downloaded from https://sparse.tamu.edu/ with the criteria that their row and column dimensions are less than 2000. Among them, there are 802 matrices with real data and 8 matrices with complex data. There are both square and rectangular matrices. That is why, in this section, the matrix dimensions are noted as $m \times n$ instead of $n \times n$.

Since our stopping criterion ensures a good accuracy, as we observed on the generated matrices, we only present the performance and obtained rank results in this section. In addition, we only provide the SVDF method results (not SVD2) as a reference, since our main point is to compare the QR variants, which use the Frobenius norm of the residuals in the stopping criterion. Note that, in this section, we use the same experimental settings as in Section 2.6.

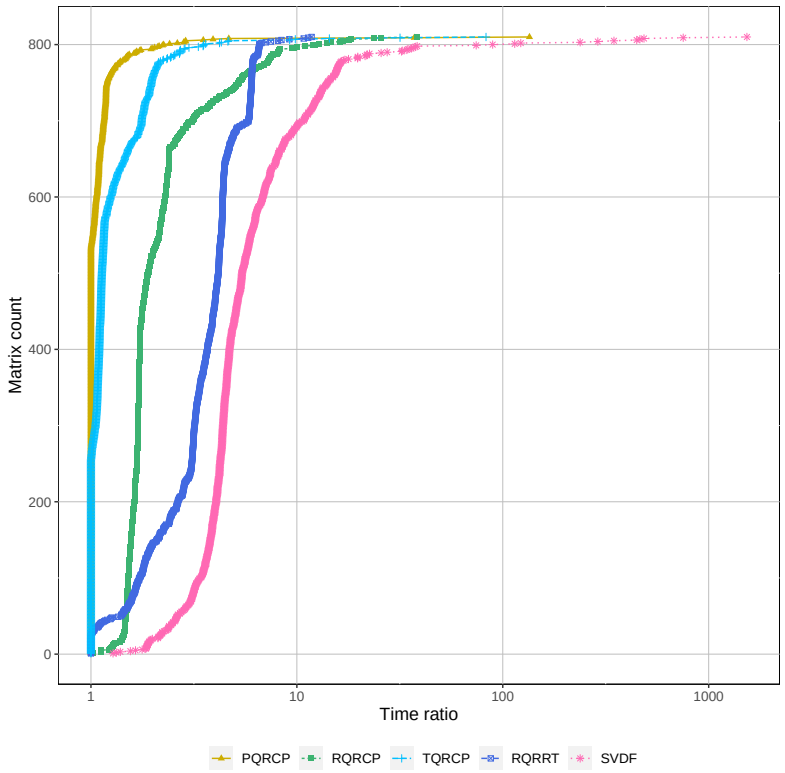


Figure 2.18: Time profiles of the real-life case test matrices. The x-axis ratios are computed as $\frac{t_{current}}{t_{min}}$ for each method, while the y-axis stands for each matrix as a number from 1 to 810. Here, the x-axis is in logarithmic scale for the sake of clarity of the figure.

In Figure 2.18, we observe the time ratio, $\frac{t_{current}}{t_{min}}$, of each method for all matrices. In this time profile, a method is better on average, if its curve is closer to the line $y = 1$.

As expected, SVDF is the slowest method for all the cases, where RQRRT is the slowest

QR variant for most of the matrices. For some very small cases, we cannot observe the expected method performances, since advantages of the methods cannot be exploited at these levels. It is interesting to observe that QRCP is the fastest method for the majority of matrices, while TQRCP is the second fastest method. The reason for this is that the real-life case matrices we used are mostly small matrices. In Section 2.6.3, we observed that TQRCP is advantageous for large matrices, so this is an expected result. RQRCP never runs as the fastest method in these experiments. As we do not use large enough matrices to benefit from this method, as well as we work in a sequential environment, this method is not promising in our context.

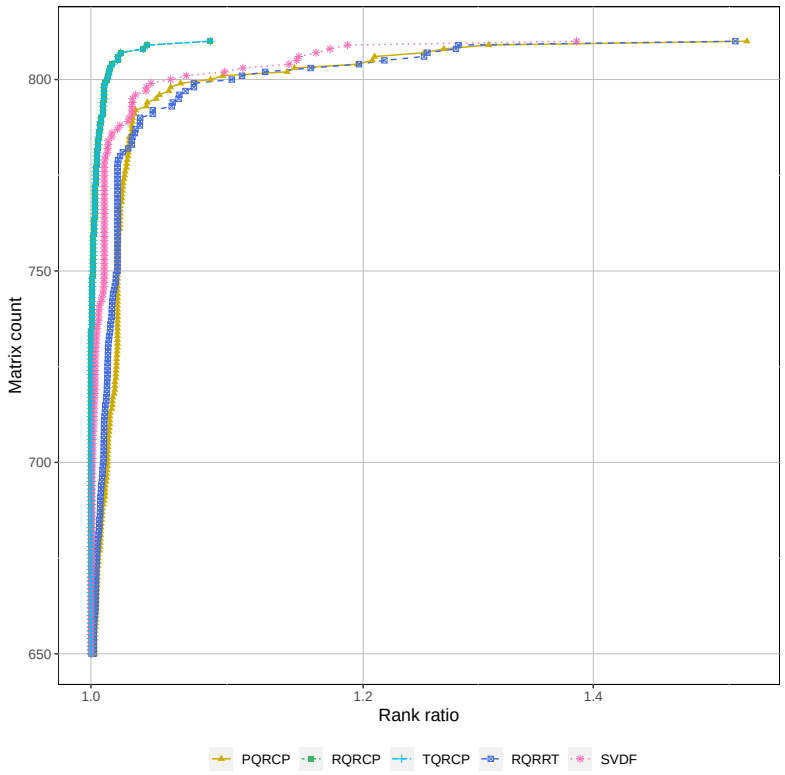


Figure 2.19: Zoomed obtained rank profiles of the real-life case test matrices. The x-axis ratios are computed as $\frac{r_{current}}{r_{min}}$ for each method, while y-axis represents each matrix as a number from 650 to 810. The y-axis values (from 0 to 649) are omitted to zoom the figure since the methods get very close results at this interval.

In Figure 2.19, we observe the compression rank ratio, $\frac{r_{current}}{r_{min}}$, of each method for some of the 810 matrices. This figure is a zoomed version of the original one in a way that it only stands for 160 matrices, which have observable difference in the figure. That is, it shows the top-most part (y-axis from 650 to 810) of the original figure. The previous y-axis values (from 0 to 649) are omitted in the figure as the methods obtain similar results at this interval.

In Figure 2.19, the values of RQRCP and TQRCP intersect, so RQRCP does not show

up. Both of these methods find the lowest compression ranks for 734 cases. Supporting the results in Section 2.6.4, even SVD finds larger ranks compared to RQRCP and TQRCP for 140 matrices because of the sample matrix based stopping criterion. Thus, RQRCP and TQRCP seem to be the most promising compression methods to find the lowest ranks possible. Considering also the performance profiles results, TQRCP seems to be better than RQRCP in sequential environments (for the matrices with dimensions less than 2000). That is, TQRCP does not only obtain the lowest ranks possible, but this left-looking method also compresses all the matrices faster than RQRCP (see Figure 2.18).

We can compare the quality of using a rotation matrix or a permutation matrix through QRCP and RQRRT. As both use a stopping criterion through the original matrix, the figure proves that RQRRT slightly improves the obtained ranks by getting closer to SVD. However, taking the performance results into account, we do not get good enough trade-off between rank and performance, in our context, through RQRRT. QRCP seems to be the least promising method to find the minimal rank. However, it is worth noting that the rank differences between the methods are not important for most of the matrices. Thus, QRCP is still promising for the small matrices, thanks to its performance.

To conclude, as good accuracy is ensured by our stopping criterion, we need to mostly focus on the obtained rank and performance results. Then, QRCP and TQRCP seem to be the most promising compression methods for us. Tuning between these methods according to the matrix size and the compressibility of the matrix (if known) can contribute to minimize the total time of the compression process, as well as the compression rank.

2.8 Discussions

In this chapter, we study and compare four different stable QR decomposition techniques, which aim to be fast alternatives to SVD when compressing dense matrices.

The most basic QR version is a panel-wise right-looking QRCP method. The second one is the randomized version of the QRCP method, so it is called randomized QRCP (RQRCP). It aims to avoid the costly data movements for large matrices when computing the column norms during the pivoting. Here, we adopt a version, where the resampling is avoided thanks to a cheaper sample matrix update formula. The third method is the left-looking version of the RQRCP method, which is called truncated randomized QRCP (TQRCP). This method reduce the computational complexity by eliminating expensive full trailing matrix updates at each iteration. Therefore, in a sequential environment, this method is the cheapest one especially for large matrices with low ranks. The last method we study is a rotational version of RQRCP. Here, we use a rotation matrix instead of column pivoting, so that we can better approximate SVD.

All the studied methods in this chapter are existing methods. However, we implement them in a similar fashion for a better comparison. In addition, we determine the compression ranks through our numerical stopping criterion at a given precision. This stopping criterion was never used for two of the studied methods, namely TQRCP and RQRRT. We conduct all the experiments in the framework of the PaStiX sparse direct

solver, and as a matter of interest, we observe the methods in a sequential environment.

In our experiments, we show that all the methods are accurate thanks to the nature of our numerical stopping criterion. Thus, we mostly focus on the performance and obtained rank results.

We observe that the obtained ranks for the RQRCP and TQRCP methods might be even better than SVDF thanks to the sample matrix based stopping criterion, while still satisfying the accuracy requirement. As TQRCP method also outperforms other methods for larger matrices, it is the best method for these matrix sizes in a sequential environment. We should keep in mind that all the compression methods obtain close ranks for the majority of the matrices. Therefore, performance results determine the best method for small matrices, which is QRCP. Then, tuning between QRCP and TQRCP is necessary according to the block sizes we use in our solver.

The QRCP method has more performance than RQRCP even for the largest matrices we used. Therefore, RQRCP is not an advantageous method for us. Nonetheless, it is still a promising method in other contexts. Here, we could not benefit from this compression method as all the experiments are in sequential and our matrix sizes might not be large enough to exploit it.

Chapter 3

Block Low-Rank Clustering

As explained in Section 1.2, direct solvers became more promising for large systems after the introduction of the low-rank representations. Thanks to these representations, the time-to-solution and memory footprint of these robust solvers can be highly reduced. Among the low-rank representations, some consider the full problem and extract the sparsity of the whole matrix from the low-rank representation, while others, like the block low-rank (BLR) representation, compress the dense blocks of the solver independently. In this work, we focus on improving the latter. More specifically, this chapter focuses on improving low-rank supernodal direct solvers for sparse systems in two different aspects.

Firstly, determining which blocks to compress and when to do it is an important problem for these solvers to be time efficient while benefiting from the lower memory consumption offered by the low-rank techniques. Therefore, we want to find a suitable admissibility criterion to offer an intermediate solution between the existing strategies that we explained in Section 1.4. For this purpose, we gather the poorly compressible blocks together by using the graph information. That is, we do not identify the existing uncompressible blocks. Instead, the identification is done through the reordering of the separator unknowns in a specific way to collect the uncompressible data together. Then, our target is to compress the highly compressible blocks at initialization (like *Minimal Memory* scenario) to improve the memory footprint, while compressing the poorly compressible ones on the fly (like *Just-In-Time* scenario) to reduce the flops overhead of the solver with a controlled memory overhead compared to *Minimal Memory*.

Secondly, as explained in 1.1.1, sparse direct solvers need to adopt an ordering strategy which ensures a good sparse structure. However, introduction of the low-rank representations into these solvers made the ordering problem more challenging. In the low-rank solvers, reordering the unknowns to improve the compressiblity can degrade the sparse structure. Therefore, in this chapter we offer an ordering (and clustering) method of the unknowns to improve both the sparse structure and compressibility of the matrix.

Before starting to explain our solutions, we provide the necessary notations and background of this chapter in Section 3.1. In Section 3.2, we mention the related work on the matrix ordering problem. In Section 3.3, we introduce and detail the Projection heuristic [Pic18], which is our starting point to solve the two problems in this chapter. In

Section 3.4, we explain our improvements on this heuristic. In Section 3.5, we discuss on the experimental results and we conclude the chapter in Section 3.6.

3.1 Background

In this section, we recall the necessary background information on the matrix ordering problem. Ordering heuristics impact many aspects of the sparse low-rank direct solvers such as the fill-in amount, the sparse structure, the compressibility or the parallelism. In order to improve all these features at the same time, we need to take advantage of several techniques. As mentioned in Section 1.1.1, we adopt the nested dissection method as the coarse level ordering, since this method reduces the fill-in and provides a good parallelism by its nature. Let us explain this recursive method in more detail.

The nested dissection is illustrated on a very simple example in Figure 3.1. Here, two levels of nested dissection procedure are applied on a regular cube (on the left). This method first splits the cube into two balanced parts through the first level separator (in grey). Note that the size of the separator should be as small as possible to minimize the fill-in. When this separator is ordered with the largest number in the elimination tree, it ensures that there is no fill-in between both sub-parts. Afterward, the resulting separated parts are also divided into two sub-parts through the second level separators (in green and red), in the same fashion. On the right part of Figure 3.1, we see the first level separator graph (in grey), where the nodes that interact with the second level separators (called traces) are colored in red and green, similarly to the left-most figure. Let us remind that both the sub-parts and the separators, which are obtained through the nested dissection algorithm, are called supernodes.

In practice, the nested dissection algorithm is applied recursively until the sub-parts are small enough to perform a local ordering heuristic on each of them. This local ordering can be applied for improving both the compressibility and the sparse structure at the same time. However, finding such an ordering is difficult.

In [PFRR17], it has been shown that permutation within each separator can be applied without impacting the fill-in ratio produced by the nested dissection. Indeed, all sparse direct solvers consider the separator blocks originated from this heuristic as dense. We rely on this property to study ordering heuristics within each separator. Before providing more details about the reordering (and clustering) techniques for the separators, let us introduce some important notations for the remainder of the chapter.

As we mentioned in Section 1.1.3, the numerical factorization is performed, in a panel-wise fashion, in three steps: factorize, solve, update. In this respect, let us represent a symmetric matrix to be factorized, A, in a block form as

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \tag{3.1}$$

where A_{22} illustrates the first level separator, while A_{11} stands for all the other unknowns. A_{12} and A_{21} are the coupling parts, which represent the interactions between

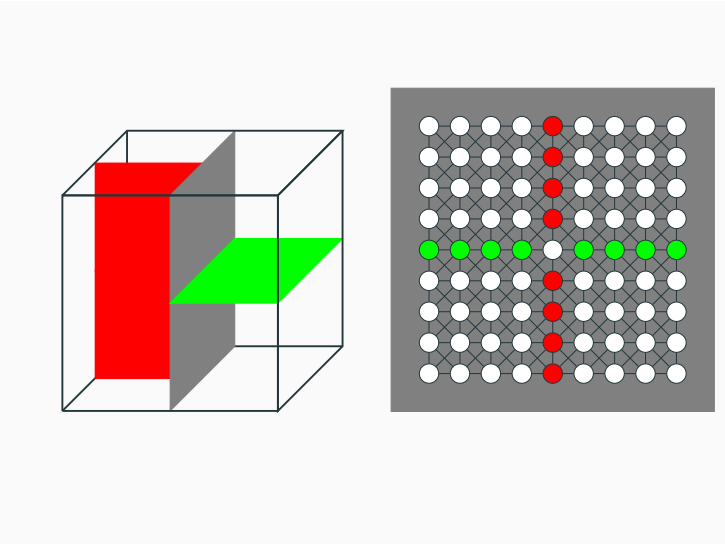


Figure 3.1: Two level nested dissection on a regular cube. On the left figure, the first level grey separator and the second level red and green separators are represented. On the right figure, the graph of the first level grey separator is shown. The red and green nodes in the right figure represent the second level red and green separator traces on the grey separator.

A_{11} and A_{22}. Using this representation, we can define the factorization steps as

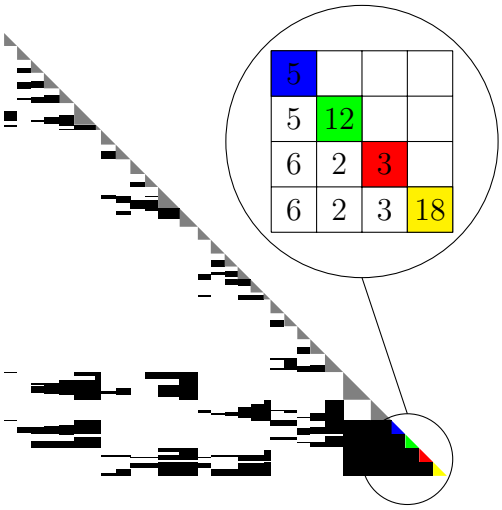
1. POTRF(A_{11})
   - This step factorizes the matrix A_{11}
2. TRSM(A_{11}, A_{21})
   - This step solves the blocks in A_{12} and A_{21}
3. HERK(A_{21}, A_{22})
   - This step updates A_{22}
4. POTRF(A_{22})
   - This step factorizes the matrix A_{22}

In the following sections, we will focus on reducing both the HERK(A_{21}, A_{22}) and the POTRF(A_{22}) operation costs through sparse structure and compressibility improvements. For the sake of simplicity, we will explain the reordering methods only on the first level separator, although it can be applied recursively at each level of the nested dissection. Now let us explain the necessary information on two promising ordering and clustering techniques: K-Way clustering in Section 3.1.1 and Traveling Salesman Problem (TSP) [ABCC06] based clustering in Section 3.1.2. Note that the corresponding related work on these two methods are detailed in Section 3.2.

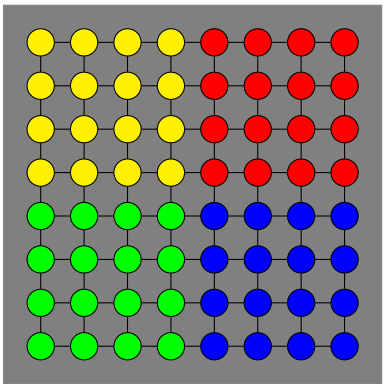
3.1.1 K-Way clustering

As mentioned before, in the sparse supernodal solvers, the clustering techniques can be performed on the separators without changing the fill-in. Performing the K-Way clustering

on a separator aims to improve its compressibility, so that the POTRF(A_{22}) cost can be reduced.



(a) Symbolic factorization



(b) The *regular* clustering of the first separator

Figure 3.2: $8 \times 8 \times 8$ Laplacian partitioned using SCOTCH and K-Way clustering on the first separator. The figure on the left stands for the symbolic factorization structure. In the zoomed figure, the total update count on each block of the first separator (A_{22}) is represented. The graph on the right shows the clustering of the unknowns inside the first separator.

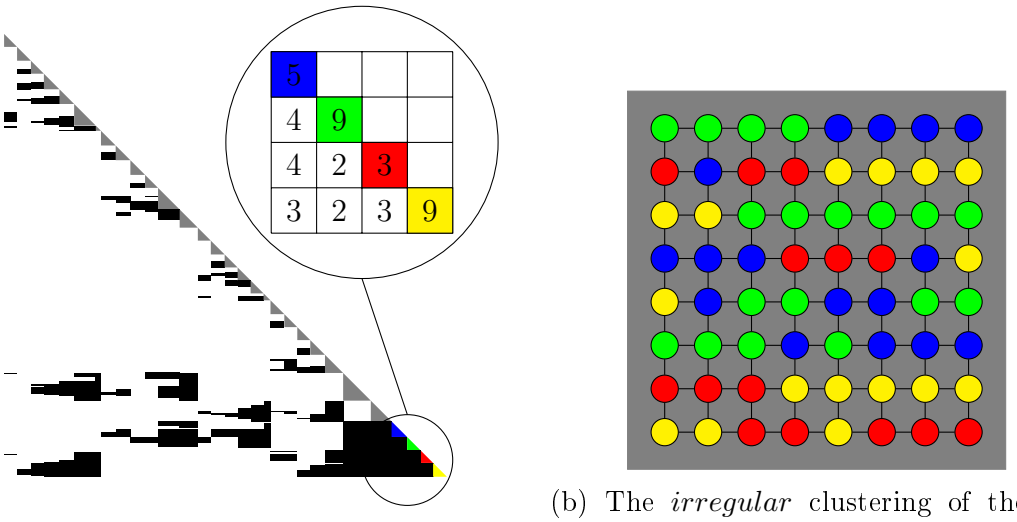
Let us explain the K-Way clustering through Figure 3.2. In the figure, we see the symbolic factorization (on the left) and the corresponding first separator graph (on the right) of an $8 \times 8 \times 8$ Laplacian, which is partitioned by SCOTCH [Pel08]. Here, the K-Way clustering is performed on the first separator, resulting in four clusters that are represented in four different colors. The zoomed part in Figure 3.2a shows the number of external updates on each corresponding block.

As we can see in Figure 3.2b, the clusters consist of close nodes of the graph and the number of neighbor clusters is minimized. From now on, we will call this kind of clustering as *regular*. Having a *regular* clustering reduces the interaction of clusters, so it improves the compressibility of the interaction blocks. However, as the sparse structure of the sparse matrix is not considered in this method, the total number of updates on each block (seen in the zoomed part of Figure 3.2a) is high, increasing the HERK(A_{21}, A_{22}) cost.

3.1.2 TSP based clustering

For improving the sparse structure in the solver, a separator clustering method which gathers similar contributions together should be used. In this respect, we adopt the TSP based reordering strategy to cluster the unknowns as in [PFRR17]. From now on, for

the sake of simplicity, we abbreviate this clustering as the TSP method. The aim of this method is to reduce the total number of off-diagonal blocks of the coupling matrix by gathering them together, which in return reduces the total number of updates. Note that this refers to the improvement of the HERK(A_{21}, A_{22}) step.



(a) Symbolic factorization

(b) The *irregular* clustering of the first separator

Figure 3.3: $8 \times 8 \times 8$ Laplacian partitioned using SCOTCH and reordering with smart splitting on the first separator. The figure on the left stands for the symbolic factorization structure. On the zoomed figure, the total update count on each block of the first separator (A_{22}) is represented. The graph on the right shows the clustering of the unknowns inside the first separator.

In Figure 3.3, we see the same information as Figure 3.2. However, in this figure, the separator clusters are generated by the TSP method. In Figure 3.3a, A_{21} has a better sparse structure (dense parts stay continuously, not spread into smaller ones), so that the total number of updates on each separator block in the zoomed figure is reduced compared to the K-Way clustering. Therefore, the efficiency of the HERK(A_{21}, A_{22}) step is improved. Moreover, since fewer and larger blocks in the coupling part might increase the compressibility of A_{21}, this method can further improve the memory footprint and computational complexity of the low-rank operations. However, as we can see in Figure 3.3b, compared to K-Way, the separator clustering is more *irregular* in a way that each color (cluster) is spread over the graph. Thus, clusters have more neighbors and larger diameters. This results in stronger cluster interactions, which reduces the separator compressibility and increases the POTRF(A_{22}) cost.

As we can see, neither the K-Way nor the TSP method provides an optimal solution. Therefore, benefiting both of these methods can lower the POTRF(A_{22}) and HERK(A_{21}, A_{22}) costs at the same time.

3.2 Related work

The related work on the low-rank solvers can be found in Chapter 1. In this section, we only discuss the literature on the clustering techniques.

In the dense low-rank solvers, the clustering aims to improve the compressibility. However, in the sparse case, in addition to the compressibility, having large enough data for the block operations is also crucial. It allows to exploit the modern architectures for a better performance. Therefore, finding an optimal clustering for the sparse low-rank solvers is a harder problem compared to the dense case.

In [RCL+18], the authors explore some ordering techniques for hierarchical solver efficiency through the STRUMPACK solver. Here, they use the geometry of the problems to obtain the cluster trees. Although, they show a huge improvement in the compressibility rate compared to the dense solver, this work is not easily extendable to our context as we work algebraically without depending on the geometry of the problems.

In [Beb08], spectral bisection based sparse matrix clustering is proposed, as well as offering the nested dissection for the low-rank clustering. The author also demonstrates the block admissibility in his work. In our context, using graph partitioning techniques on separator graphs is a similar idea to this work since we focus on the separator clustering.

In [YLRB17], an algebraic dense matrix clustering is proposed. For this purpose, the authors use a result from reproducing kernel Hilbert space theory in [HSS08]. According to this result, any SPD matrix corresponds to a Gram matrix of vectors in an unknown Gram space. As a result, they define their distances by using the matrix entries as inner products. By using these distances they generate a balanced cluster tree, where they avoid computing all the distances through sampling since it is costly for dense matrices.

The K-Way clustering is a classical low-rank clustering method to cluster the fronts or separators in sparse solvers. It is an extension from dense low-rank solvers, where it successfully improves the compressibility. The main idea of this clustering is to choose the diameter of the clusters as small as possible, and minimize the total number of cluster neighbors. In this way, the nodes with strong interactions are gathered together and the cluster interactions are minimized through a *regular* clustering. As a result, the cluster interaction blocks are more compressible, which in return improves the time-to-solution and memory footprint of the solver. In MUMPS [AAB+15, ABLM19b] and STRUMPACK [GLR+16, GLGR17], this clustering is adopted on the fronts. In our context, we perform it on the separators. Note that the K-Way clustering requires a connected graph, but fronts or separators are not always connected. Therefore, it is applied in two steps. Firstly, the vertices of the fronts or separators are reconnected with a halo distance 1 or 2 for obtaining a connected graph. Then, the K-Way clustering can be applied through partitioning libraries like METIS [KK97b] or SCOTCH [Pel08]. Although, this method provides compressible separators, it fails to provide a good sparse structure. In addition, large number of small off-diagonal blocks in this method may degrade the compressibility of the coupling part.

Let us emphasize that having large enough data sizes within the sparse solvers is very important. Methods like Reverse Cuthill-McKee (RCM) [GL81] are widely used to reduce

the number of off-diagonal blocks in the column blocks. However, this approach only considers the intra-supernode interactions, without taking the contributing supernodes into account. Therefore, we adopt the reordering method in [PFRR17], where the authors convert the optimization problem of minimizing off-diagonal blocks number into the traveling salesman problem (TSP). This method provides a good sparse structure in the coupling part by gathering the similar contributions together. In this method, the clustering, following the TSP based unknown reordering, is obtained by using the smart splitting as in [Lac15]. This smart splitting does not use strictly fixed size clusters in order to decrease the total number of off-diagonal blocks. Note that although the TSP method in [PFRR17] is more suitable to be parallelized, in [JNP21] this method is improved to highly reduce its cost. The authors improve it by reducing the size of TSP and providing a more efficient way of computing the TSP distances. However, even though the TSP reordering provides a good sparse structure, it does not take into account the compressibility of the separator A_{22}.

To conclude, the K-Way and TSP methods are promising for the sparse solver separator clustering in different perspectives. In order to obtain an optimal solution in terms of both compressibility and sparse structure, we can take advantage of these two methods at the same time. In our work, we propose to improve the Projection heuristic (see Section 3.3), which profits from both of these methods and gathers the uncompressible blocks together.

3.3 Projection heuristic

In Sections 3.1.1 and 3.1.2, the existing K-Way and TSP clustering methods and their aims are explained. In this section, we study the Projection heuristic introduced in [Pic18], which is the main focus in our work. This heuristic targets to improve both compressibility and sparse structure at the same time to outperform the existing clustering methods. In addition, it aims to determine the strong interaction blocks through the adjacency graph, which are poorly compressible. Let us start explaining this heuristic step by step.

3.3.1 Determining the traces

In this step, the vertices of the separator are ordered according to the set of contributions on them, which comes from the closest children in the elimination tree.

In Figure 3.4, we see a generic separator as an example. The figure illustrates the interactions of two levels of the closest children on this separator. These interactions represent the traces on the separator, which are resulted from the nested dissection (see Section 3.1). These traces are colored in red for the first level children, while they are in green for the second level children. There are two red traces from the first level children, and four traces from the second level ones. Each pair results in a chromosome shape, as seen in the figure.

In this step, firstly, the trace vertices (preselected vertices that represent the nodes

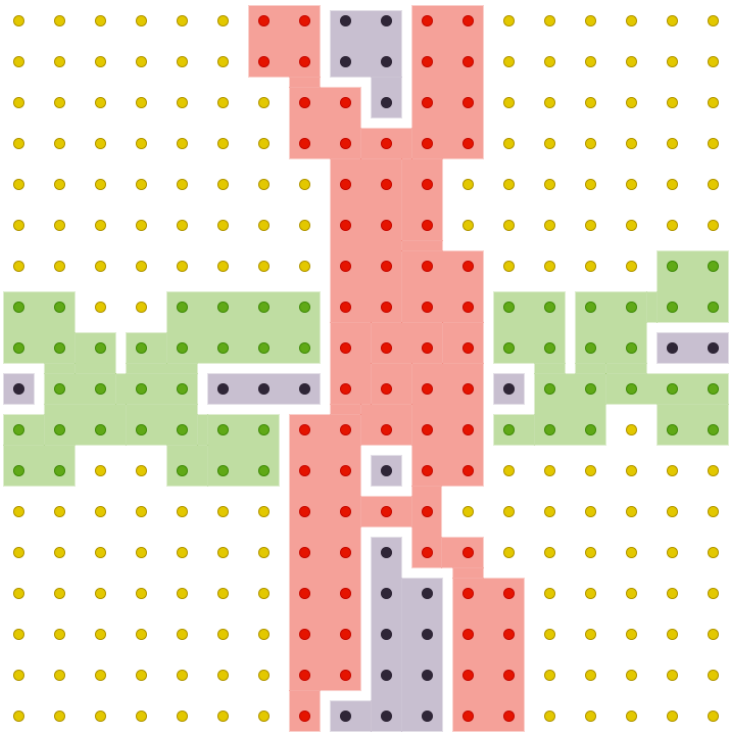


Figure 3.4: A generic separator with two levels of traces represented on it. First level separators are in red, while the second level separators are in green. There are two red traces from the first level children, and four traces from the second level ones. Each pair separator leads to a chromosome shape. Small node groups (in grey) appear between the separators.

belonging to the traces in the separator graph) are determined and ordered together. The remaining vertices are in several well-separated groups, called connected components (grey and yellow color in the figure). Each isolated connected component gets the same set of contributions from the two levels of children considered in the elimination tree, so that they have identical sparsity pattern.

Note that the connected components which have a smaller size than a given threshold (grey color in Figure 3.4) are gathered together to be large enough with respect to the minimal compressibility size. This component made out of the small ones is called as the remainder component.

Gathering the preselected vertices together and isolating the connected components provides two main benefits. Firstly, it leads to a suitable clustering for the connected components, since it increases the distance between them. Thus, the compressibility of their interactions is improved. Secondly, the preselected vertices have strong interactions. Therefore, the blocks that represent their interactions are poorly compressible, and are identified as *uncompressible* within the Projection heuristic. Let us emphasize that the *uncompressible* blocks in our context do not represent only the uncompressible blocks but also the poorly compressible blocks for the sake of simplicity.

At the end of this step, representing our separator as

$$A_{22} = \begin{pmatrix} A_{kk} & A_{ks} \\ A_{sk} & A_{ss} \end{pmatrix} \tag{3.2}$$

the preselected vertices are ordered in A_{ss} and others are ordered in A_{kk}. Here, the only compressible part is A_{kk}, while others are defined as *uncompressible*. In addition, the sub-diagonal blocks represent the connection of two consecutive column blocks. Therefore, they are also considered as *uncompressible* since they have a high probability of having strong interactions. Note that the blocks with small sizes are considered to be non-admissible for low-rank compression, due to their low impact. From now on, all the small blocks and the dense diagonal blocks are automatically defined as non-admissible and never compressed to avoid unnecessary compression.

3.3.2 K-Way and TSP methods within the Projection heuristic

Initially at this step of the heuristic, there are a few large connected components, the traces and the remainder component. In order to reduce the size of the diagonal blocks, a clustering method is applied on each of them.

Each large connected component is clustered through the K-Way method to increase compressibility of the cluster interaction blocks. However, as the traces are considered to have strong interactions, K-Way is not performed on them. They are clustered through the TSP method to improve the sparse structure. The remainder component is also clustered through the *irregular* TSP method since it consists of small components that are already split in the graph.

Note that each connected component cluster (obtained through the K-Way partitioning) is further reordered through the TSP method. Here, the TSP method only serves as a reordering method to improve the sparse structure at the cluster level, without splitting the unknowns into different clusters. Now let us continue with some important implementation details of this heuristic.

3.3.3 Implementation details

As seen in Algorithm 9, at the beginning, the separator graph is isolated from the original one with the function ObtainSubgraph(C). This routine generates direct connections at a distance 2 from the original graph to obtain the connected separator graph.

In the function ComputeTraces(C, l, d, w), the traces are determined by checking each vertex of the separator. It is applied in two steps. Firstly, for each separator vertex, it checks the direct connections with the children for each depth level (less than or equal to l). If there is a distance less than or equal to d between them, this vertex is preselected. After preselecting these vertices, the vertices inside the separator, which have distance less than or equal to w to a preselected vertex, are also set as preselected.

Algorithm 9 Clustering and ordering the unknowns within the separator C in the original heuristic.

```
1:  function OriginalProjection(C, l, d, w)
2:      ObtainSubgraph( C )
3:      ComputeTraces( C, l, d, w )
4:      IsolateConnectedComponents( C )
5:      for each connected component C_i do
6:          if |C_i| < threshold then
7:              Merge C_i into the remainder component
8:          else
9:              K-Way( blocksize, C_i )
10:             for each K-Way cluster K_j do
11:                 TSP( K_j )
12:     TSP( traces )
13:     TSP( remainder component )
```

In the function IsolateConnectedComponents(C), the connected components of the separator are isolated. It applies Breadth-First Search (BFS) algorithm before visiting each vertex one time. Every time this algorithm stops before all the vertices are visited, it means that another connected component is generated.

In the connected components loop, the small connected components are all merged together. Therefore, at the end of the loop, the large remainder component can be clustered through the TSP method. On the other hand, each large connected component is firstly clustered through the K-Way procedure, which is followed by a TSP based reordering at the cluster level. Finally, the TSP method is applied on the traces to improve the sparse structure.

In practice three main parameters are used to choose the preselected vertices: Projection depth, Projection distance, Projection width.

Projection depth, l, specifies how many levels of children should be considered to determine the traces on the separator. If this parameter is set to l, 2^l children in the elimination tree are used to determine the contributions on the vertices. Therefore, this parameter affects the number of traces. If l is too large, there will be a lot of small connected components and it will reduce the compressibility. Thus, it should be chosen small.

Projection distance, d, specifies the distance of the vertices, from a child to a separator, to be considered as part of the projection. Therefore, it affects the width of the traces. As d is getting larger, the number of preselected vertices will grow because more vertices are accepted as part of the projection.

Projection width, w, is used for widening the traces after their computation. The aim of this parameter is to get a better compressibility of the clusters on each side of the traces by better isolating them.

These parameters should be chosen carefully. That is, the blocks should be compressed

as much as possible and there should be large enough connected components for further clustering. Therefore, small parameters should be adopted to reduce the number of preselected vertices and have large connected components. On the other hand, if the traces are not sufficiently large or if there are not enough number of traces, there will not be large number of well-separated connected components. In this case, the Projection heuristic cannot be exploited sufficiently.

In this heuristic, the preselected vertices are accumulated from the children, level by level (from the closest children to the farthest), based on the Projection depth (l) parameter. For a separator of size n, if the preselected vertices are more than $\Theta(\sqrt{n})$, the preselection terminates. Thanks to this stopping criterion, the risk of preselecting too many vertices is eliminated. In addition, let us represent the blocking size by b. The Projection heuristic is only applied on separators larger than size $16b$ to sufficiently benefit from it.

It is important to emphasize that the separators, which are obtained with the partitioning tools, are not always connected. That is why, in practice the separator graph is reconnected, using a distance 1 or 2, to allow the K-Way clustering on it. We adopt a distance of 2 for being safe, although in [AAB+15] it is shown that a distance of 1 is enough for most of the graphs.

3.3.4 Discussions on the Projection heuristic

In the original work [Pic18], the author shows that the Projection heuristic can reach a 10% time improvement with only a 5% memory overhead compared to the existing K-Way method for the *Minimal Memory* strategy. However, the original heuristic is not optimal and can be improved. In this respect, we want to both validate the admissibility criterion and increase the performance.

In this chapter, one of our main focus is to empirically validate the admissibility criterion in the Projection heuristic. In Figure 3.5, we have three scenarios to identify the poorly compressible blocks. Here, pink color represents the preselected blocks. These blocks show the interaction with the preselected nodes. Therefore, they should be poorly compressible. Black color stands for the sub-diagonal blocks. As the sub-diagonal blocks represent two consecutive blocks in the matrix, they are possibly poorly compressible because these blocks represent close distances in the graph. Pink and black blocks are our interest in terms of compressibility. Brown color illustrates the preselected node cluster blocks. As seen in the figure, according to the number of preselected nodes and the blocking size, the preselected nodes can be split into different clusters. Blue color shows the diagonal blocks of the non-preselected node clusters, while the yellow color stands for all the remaining blocks.

In the left-most scenario, all off-diagonal blocks are considered compressible. In the middle figure, the sub-diagonals are considered as poorly compressible, while in the right-most one, both the sub-diagonals and the preselected blocks are identified as poorly compressible. By comparing these three scenario results, we want to validate the hypothesis on determining the poorly compressible blocks. In addition, we want

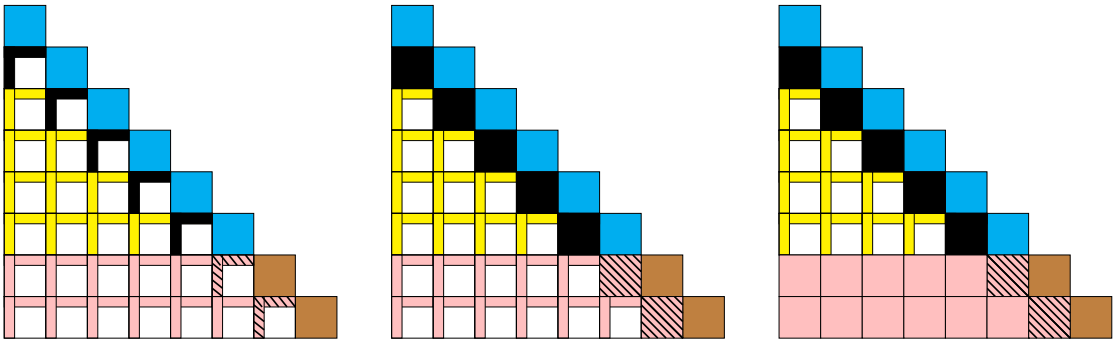


Figure 3.5: Three different scenarios to identify the poorly compressible blocks. Pink color represents the preselected blocks, which are the interaction with the preselected nodes. Brown color illustrates the preselected node cluster blocks. Black color stands for the sub-diagonal blocks. Blue color shows the diagonal blocks of the non-preselected node clusters, while the yellow color is for all the remaining blocks. In the left-most scenario all the off-diagonal blocks are compressed, while in the middle figure the sub-diagonal blocks are kept in full-rank. In the right-most scenario both the sub-diagonal and preselected blocks are represented in full rank.

to experimentally prove that compressing the *uncompressible* blocks on the fly highly improves the computational complexity. After this empirical validation step, we are interested in improving the heuristic in two ways.

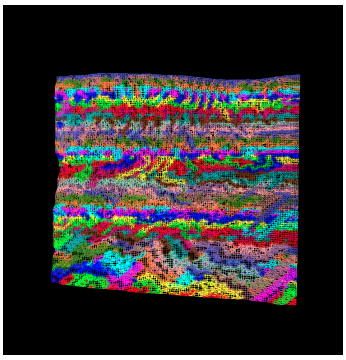
Firstly, improving the compressibility of the *uncompressible* blocks is important when they are compressed on the fly. It reduces the computational complexity of the update operations where they contribute in a low-rank form. In addition, in PaStiX, the blocks which are less compressible than a threshold are not compressed, because of their low impact. If we improve the compressibility of the *uncompressible* blocks, larger number of blocks are compressed as more blocks are eligible according to this threshold. Therefore, the resulting low-rank contributions improve the computational complexity of the operations. Thus, we propose to apply K-Way clustering on the traces for a better flops overhead of the solver. However, note that this improvement does not improve the memory peak since the *uncompressible* blocks are still allocated in full-rank at initialization.

Secondly, as the small components are not necessarily close in the graph, merging them together degrades the separator compressibility, and therefore the computational complexity. Therefore, we propose to eliminate the remainder component and merge them with their neighbors in the graph.

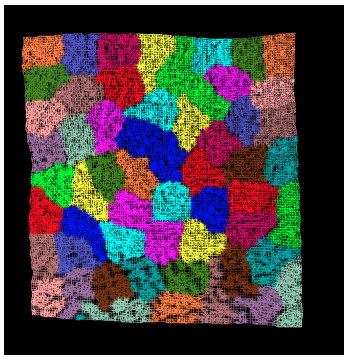
To conclude, our aim is to introduce more *regular* clusters in the Projection heuristic. In this way, we primarily expect to improve the computational complexity of factorization.

3.4 Improving the Projection heuristic

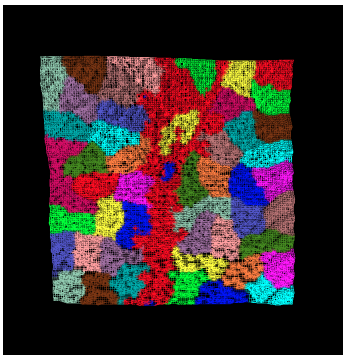
In the previous section, we discussed our motivation to change the original Projection heuristic in a way to have more *regular* separator clusters. In this section, we give the



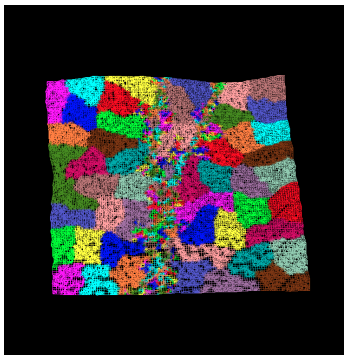
(a) TSP method applied on the separator



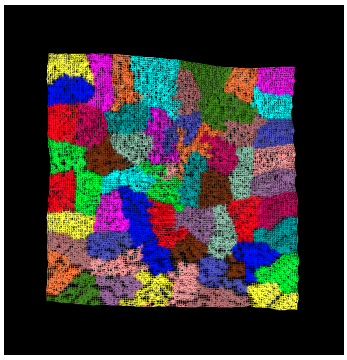
(b) K-Way method applied on the separator



(c) Traces in the Projection heuristic before clustering of the traces



(d) Final separator clustering in the original Projection heuristic



(e) Final separator clustering in our improved Projection heuristic

Figure 3.6: The first separator graphs of a $120 \times 120 \times 120$ Laplacian matrix. For the Projection heuristic, the distance and width parameters are set to 2 and 1, respectively. We can see two traces from the two children of the first depth level.

details of our changes on this heuristic. However, let us start by a quick comparison of all the clustering techniques that we study in this chapter. In Figure 3.6, we observe clusters on the first separator graphs of a $120 \times 120 \times 120$ Laplacian matrix with the proposed solutions.

In Figure 3.6a, we observe the TSP method on the whole separator. As we explained before, this method fails to cluster nicely the vertices. All clusters are spread and interleaved in the graph. Although, the outside contributions are gathered together to reduce the total number of updates, the internal clustering of the separator is not suitable to increase its compressibility.

In Figure 3.6b, we see the K-Way clustering on the whole separator, where the separator is clustered *regularly* to improve its compressibility. Note that in this clustering, each cluster is reordered through TSP at the cluster level to improve the sparse structure of the external contribution. Although the separator compressibility is better compared to Figure 3.6a, the cluster-wise TSP based reordering cannot reach a sparse structure as

good as the separator-wise reordering. However, this strategy greatly reduces the cost of the TSP based reordering by applying it to smaller clusters.

Last three figures stand for the Projection heuristic. In Figure 3.6c, we see the heuristic before applying clustering on the traces. Here, the Projection distance and width are set to 2 and 1, respectively. We can see the two first level traces on the separator (in red). The well-separated connected components are *regularly* clustered through the K-Way method. In Figure 3.6d, we see the same graph after TSP method is applied on each cluster and trace in the original heuristic. As the connected components are clustered before the reordering method, their *regular* clustering is not affected. On the other hand, as the trace clusters are obtained through the TSP method, their interaction compressibility is degraded.

Finally, in Figure 3.6e, we see our improved clustering. After Figure 3.6c is obtained similarly to the original heuristic, we merge the small connected components with their neighbors, as well as applying K-Way clustering on the traces. Following this step, the TSP based reordering is still applied on all the *regular* clusters to refine the sparse structure. Observe that our improved clustering is almost as *regular* as K-Way method, with an advantage of gathering (identifying) the high interaction nodes together.

Algorithm 10 Clustering and ordering the unknowns within the separator C in the improved heuristic.

```
1:  function PROJECTION(C, l, d, w)
2:      ObtainSubgraph( C )
3:      ComputeTraces( C, l, d, w )
4:      K-Way( blocksize, Traces )
5:      IsolateConnectedComponents( C )
6:      for each connected component C_i do
7:          if |C_i| < threshold then
8:              Merge C_i with one neighbor
9:          else
10:             K-Way( blocksize, C_i )
11:     for each cluster K_i do
12:         TSP(K_i)
```

Algorithm 10 stands for our version of the Projection heuristic. Our improvements compared to the version [Pic18] is written in blue. Let us explain this algorithm, comparing to the original heuristic.

We start by determining the preselected vertices and isolating the connected components in the same way as in the original heuristic. At this point, we apply the K-Way clustering on the traces to improve the compressibility of the trace interaction blocks. Afterward, we isolate the connected components like in the original version.

In the connected components loop, we merge the small connected components with their neighbor clusters, while they are all merged together through the remainder component in the original heuristic. Therefore, we obtain more *regular* clusters on the

separator, which improves the compressibility of the separator. We do it by checking all the separator vertices. Whenever we find a neighbor vertex, we connect it to the large component of the determined neighbor vertex. On the other hand, the large connected components are clustered through the K-Way procedure like in the original heuristic.

At this point we apply the TSP based reordering on each cluster to improve the sparse structure at the cluster level similarly to the original version.

3.5 Experiments

In this section, we aim to demonstrate the impact of our improvements on the Projection heuristic. All the experiments are performed through the BLR supernodal direct solver PaStiX [PDF+18], using the `miriel` nodes of the *Plafrim*[1] supercomputer. Each `miriel` node is equipped with two Intel `Xeon E5-2680v3` 12-cores running at 2.50 GHz and 128 GB of memory. For the time results, we use 24 threads, one per core, with the default scheduler of the PaStiX library. The Intel MKL `2020` is used for the BLAS kernels. The minimum block width and height criteria to allow compression are set to 128 and 20, respectively. In the experiments, we use only LDL^T and LU factorizations according to the input matrix features. For the SPD matrices, we avoid LL^T factorization as the positive definite property can be affected by the compression.

In the experiments, we adopt the real-life matrices, which arise from a real mesh and are given in Table 3.1. They are taken from the SuiteSparse Matrix Collection [DH11]. In addition to these matrices, we also use a Laplacian matrix of size 120^3.

Let us emphasize that for the sake of simplicity, the TSP name represents the method, where the separator clusters are obtained through the TSP based reordering with smart splitting as in [PFRR17] to gather similar contributions together. Similarly, the K-Way name stands for the K-Way low-rank clustering method followed by a TSP based reordering on each resulting cluster to improve the sparse structure.

We start our experiments by proving the effectiveness of our admissibility criterion and necessity of the late compression of the *uncompressible* blocks in Section 3.5.1. We then discuss the results of our improvements on the Projection heuristic, through the latest PaStiX version, in Section 3.5.2. Here, we observe the flops and time results on a large set of 31 real-life case matrices. Finally, in Section 3.5.3, we compare our improved heuristic to the existing methods.

3.5.1 Determining the block compression

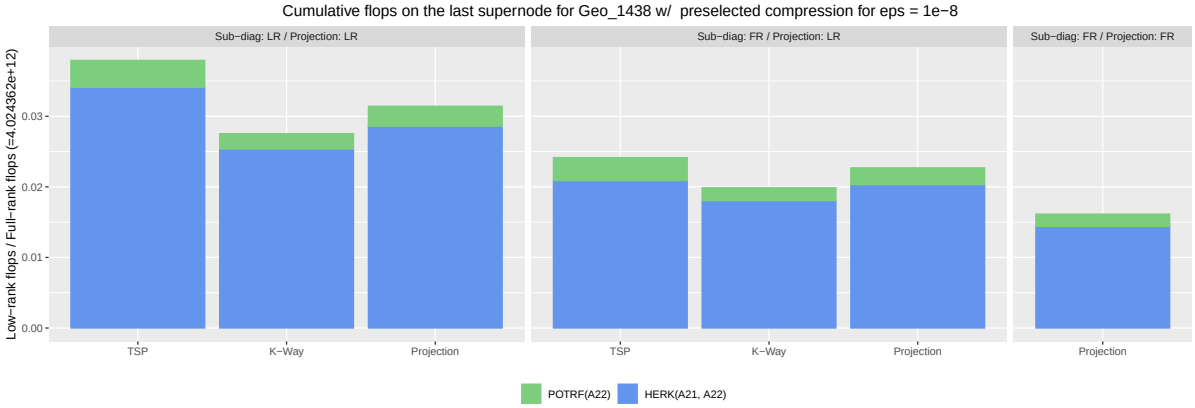
As a starting point to our experiments, our first interest is to numerically validate our admissibility criterion, as well as determining whether the poorly compressible blocks should be compressed on the fly or not.

In Figure 3.7, the three grid columns stand for the three scenarios in Figure 3.5, respectively. We observe the flops ratio with respect to the full-rank solver for the

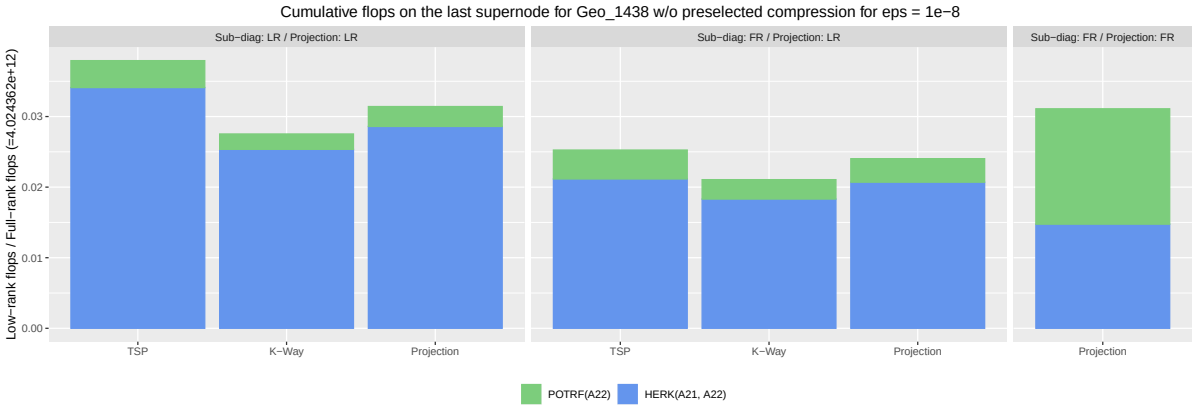
[1] https://www.plafrim.fr

Kind	Matrix	Arith.	Fact.	N	NNZ_A
2d/3d	PFlow_742	d	LL^T	742793	18940627
	Bump_2911	d	LL^T	2911419	65320659
Computational fluid dynamics	StocF-1465	d	LL^T	1465137	11235263
	atmosmodl	d	LU	1489752	10319760
	atmosmodd	d	LU	1270432	8814880
	atmosmodj	d	LU	1270432	8814880
	RM07R	d	LU	381689	37464962
Electromagnetics	dielFilterV3clx	z	LU	420408	16653308
	fem_hifreq_circuit	z	LU	491100	20239237
	dielFilterV2clx	z	LU	607232	12958252
Magnetohydrodynamics	matr5	d	LU	485597	24233141
Materials	3Dspectralwave2	z	LDL^H	292008	7307376
Model reduction	boneS10	d	LL^T	914898	28191660
	CurlCurl_3	d	LDL^T	1219574	7382096
	bone010	d	LL^T	986703	36326514
	CurlCurl_4	d	LDL^T	2380515	14448191
Structural	ldoor	d	LL^T	952203	23737339
	inline_1	d	LL^T	503712	18660027
	Flan_1565	d	LL^T	1564794	59485419
	ML_Geer	d	LU	1504002	110879972
	audikw_1	d	LL^T	943695	39297771
	Fault_639	d	LL^T	638802	14626683
	Hook_1498	d	LL^T	1498023	31207734
	Transport	d	LU	1602111	23500731
	Emilia_923	d	LL^T	923136	20964171
	Geo_1438	d	LL^T	1437960	32297325
	Serena	d	LL^T	1391349	32961525
	Long_Coup_dt0	d	LDL^T	1470152	44279572
	Cube_Coup_dt0	d	LDL^T	2164760	64685452
	Queen_4147	d	LL^T	4147110	166823197

Table 3.1: The real-life matrices in our experiments, which are taken from the SuiteSparse Matrix Collection.



(a) *Uncompressible* blocks are compressed on the fly



(b) *Uncompressible* blocks are never compressed

Figure 3.7: Flops overhead with respect to the full-rank. Each grid column represents the scenarios in Figure 3.5, respectively. Each color shows the total cost of different operations on the first separator.

*Geo*_1438 matrix at $1e^{-8}$ tolerance. Data reported on the graphs are only related to the first separator. Each color represents different operation cost, while each bar stands for a different clustering technique. The right-most configuration only changes the Projection heuristic results as the preselected blocks, which represent the trace node interactions, only exist in this method. For the Projection heuristic, the depth, distance and width parameters are set to 3, 1 and 1, respectively.

Figure 3.7a stands for the scenario where the poorly compressible blocks are compressed on the fly. Here, the left-most scenario, where all off-diagonal blocks are compressed before any operation, is the most costly one for all the clustering techniques. This is because of the costly LR2LR update kernels that explained in Section 1.3. However, in the middle figure, when the poorly compressible sub-diagonal blocks are compressed on the fly, the flops overhead of the low-rank operations is reduced. In the right-most figure, proving the hypothesis that both the sub-diagonal blocks and the

preselected blocks are poorly compressible, the Projection heuristic, where these blocks are compressed on the fly, has the best computational complexity.

In Figure 3.7b, we observe the same configuration experiments, with a scenario where the poorly compressible blocks are not compressed at all. As no block is defined as poorly compressible in the left-most figure, the result is the same as Figure 3.7a through this configuration. However, observing Figure 3.7a and Figure 3.7b, we can clearly see that compressing the poorly compressible blocks on the fly reduces the computational complexity compared to the scenario where they are not compressed at all. This improvement comes from the cheaper low-rank contributions of the *uncompressible* blocks compared to their full-rank versions.

To conclude, the Projection heuristic, where both the sub-diagonal and preselected blocks are compressed on the fly, reduces the flops overhead on the first separator computations compared to other clustering methods and configurations. Therefore, from now on, we will adopt this configuration in our experiments and call this optimized version as the OptProj heuristic.

3.5.2 Experiments on the OptProj heuristic

In this section, we compare the OptProj heuristic to the OptProj versions with our improvements. We do so by using a large set of real-life matrices through the last version of PaStiX. We set the Projection depth, distance and width parameters to 3, 1 and 1, respectively.

In Figure 3.8, the left-most figures are the factorization flops profiles, whereas the right-most figures stand for the factorization time profiles. The x-axis shows the flops and time overhead with respect to the minimum for each matrix, respectively. A method is better on average if its curve is closer to $x = 1$.

In the figure, the yellow curve stands for our OptProj heuristic, where we merged the small connected components with their neighbors. This OptProj version with Eliminated Remainder Component is abbreviated as OptProj+ERC. As we can see in the figures, the flops overhead of our contribution is very close to the basic adapted heuristic (in blue). However, in the time profiles, we can clearly see the improvement of our new heuristic version. The reason of more time improvement compared to the flops difference is because of the operation efficiency differences of *Just-In-Time* and *Minimal Memory*. That is, the small components are merged with their high interaction neighbor trace clusters, so that the *uncompressible* data is increased. As more data is compressed on the fly and the *Just-In-Time* operations are more efficient than the *Minimal Memory* operations [Pic18], the factorization time of our heuristic is improved.

In the figure, the black curve stands for our OptProj heuristic, where we performed K-Way on the Traces (abbreviated as OptProj+KWT). As we can see in the figures, since our new heuristic degrades the sparse structure, it can even have worse computational complexities compared to the basic adapted heuristic, especially at lower precisions. This is caused by the larger number of smaller coupling matrix blocks, which are not eligible for compression. On the other hand, this method separates the trace nodes well. Thus,

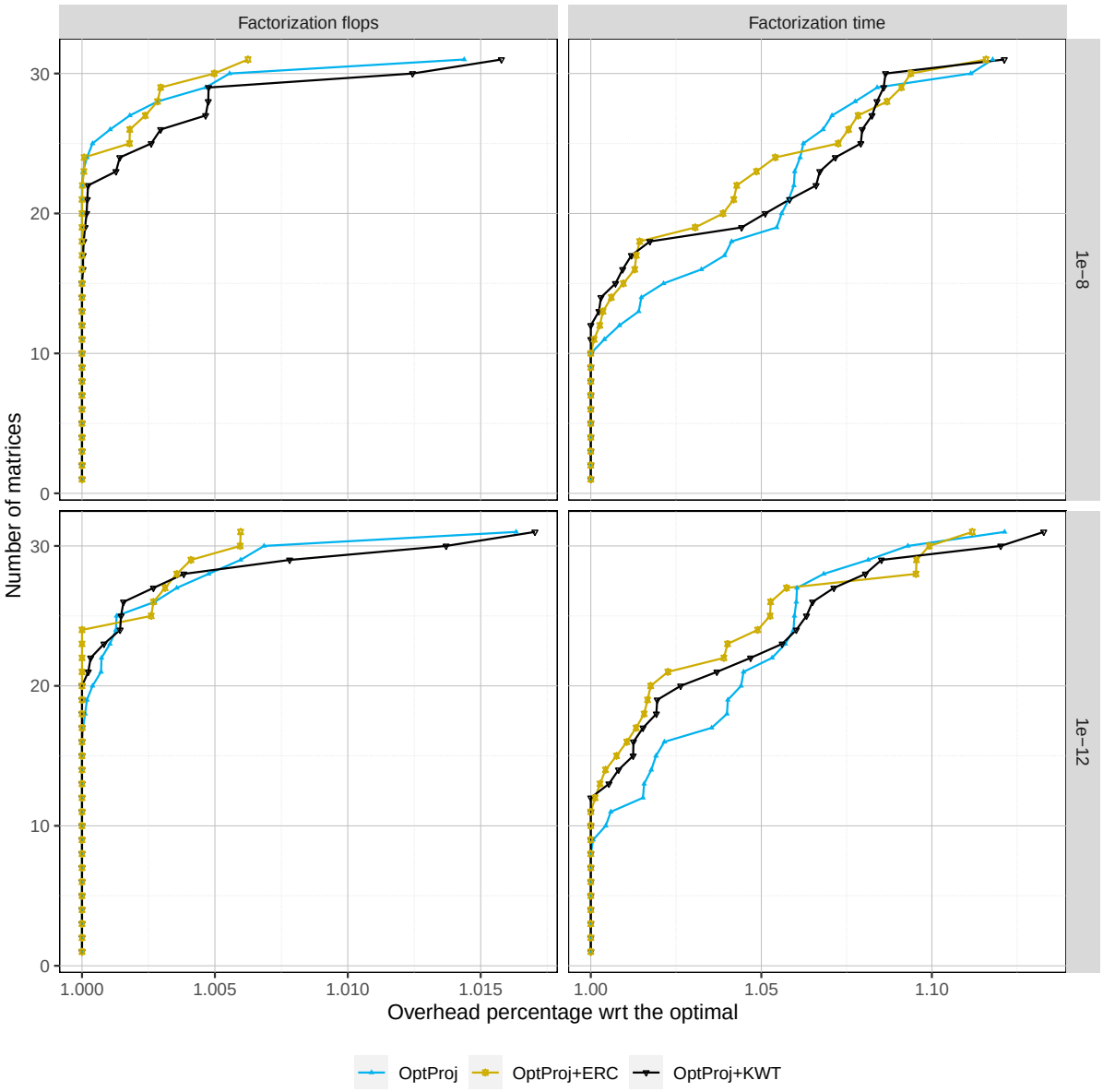


Figure 3.8: The flops and time profiles for real-life case matrices. The y-axis represents the number of matrices. At the left-most figures, the x-axis stands for the ratio $\frac{flops_{method}}{flops_{min}}$ for each matrix. Similarly, right-most figures show the relative time ratios with respect to the minimum. OptProj+ERC stands for the OptProj method with Eliminated Remainder Cluster. Similarly, OptProj+KWT represents the OptProj version, where K-Way is applied on the Traces.

it avoids the interaction between these trace clusters. Then, the resulting null blocks of the separator through the OptProj+KWT strategy allows a time improvement in the figures, despite its computational complexities in total. In the time profiles, we can see

that OptProj+KWT is the fastest method for one third of the matrices.

3.5.3 Comparing OptProj+ERC heuristic to the existing solutions

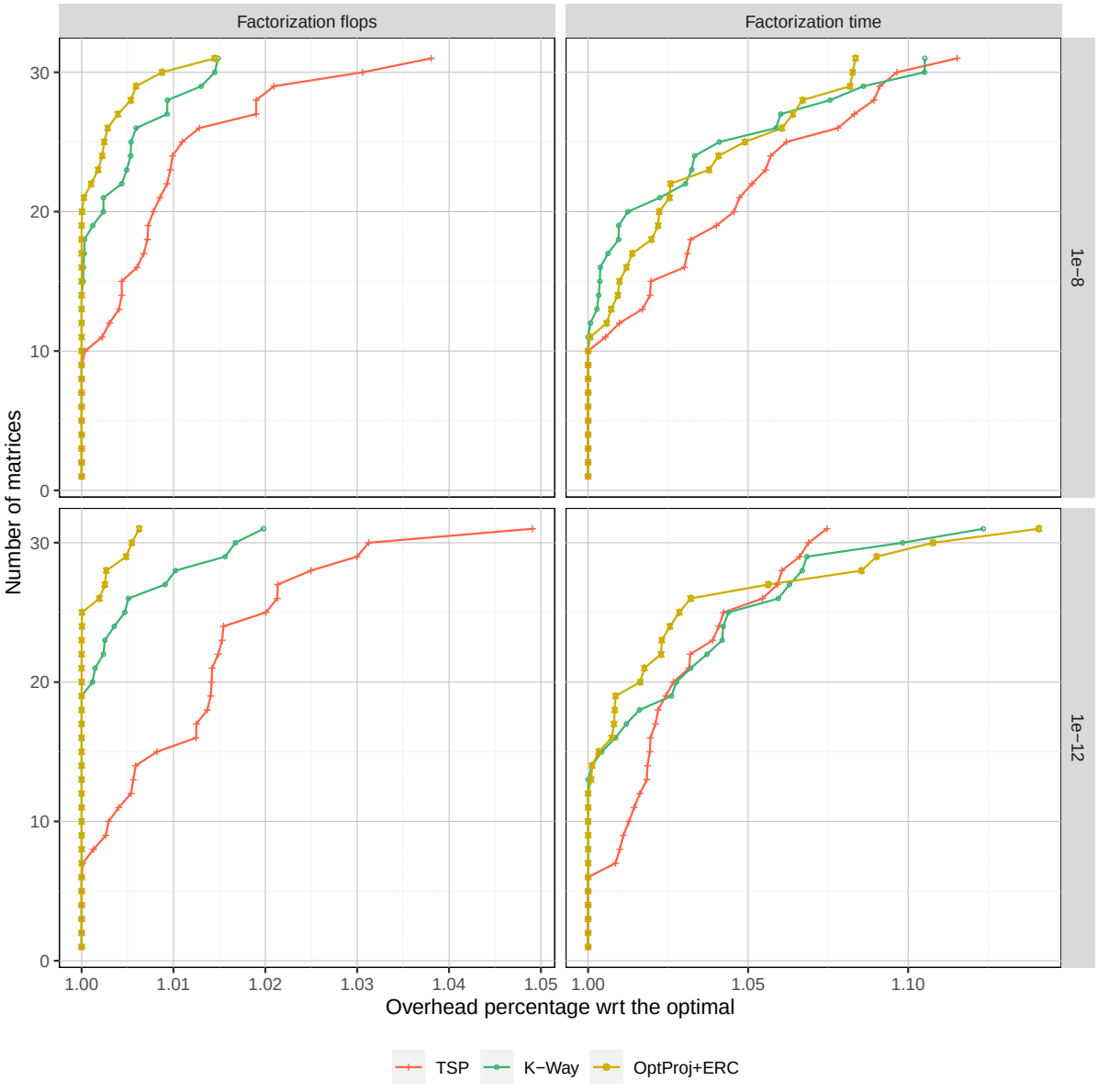


Figure 3.9: The flops/time profiles for real-life case matrices. The y-axis represents the number of matrices. The x-axis stands for the results with respect to the optimal for each matrix. The yellow curve stands for our OptProj heuristic with Eliminated Remainder Component, where the small components are merged with their neighbors.

In this section, we want to observe our contribution improvements compared to the existing methods. We have chosen the OptProj+ERC version as the most promising method through the Figure 3.8. Therefore, we will compare this version to the existing methods.

Observing the flops profiles of the Figure 3.9, our OptProj+ERC method has a better computational complexity than the existing methods on average, as we expected. As TSP method does not provide separator compressibility, there are more operations in this method. On the other hand, although K-Way has a good separator compressibility, our heuristic gathers the strong interaction blocks together and does not compress them before the factorization. In this way, our heuristic provides more compressibility ratio in the compressible blocks, while avoiding the costly low-rank updates of the strong interaction blocks. As a result, the OptProj+ERC method has the best flops profile on average.

On the other hand, when we observe the time profiles, we can see the effect of the sparse structure. The TSP method becomes more competitive in terms of factorization time, while it still cannot be the best average method because of its high flops overhead. It is interesting to observe that the OptProj+ERC heuristic is worse than the K-Way method on average, although it has a better flops profile. The reason is that determining the trace nodes is not trivial in practice. Choosing large enough traces to isolate the connected components and gather all the strong interaction unknowns together is crucial. However, when we gather the preselected vertices together, we can split the similar contribution blocks even more compared to K-Way. As a result, the OptProj+ERC heuristic cannot improve the average time overhead compared to the K-Way method at lower precisions.

3.6 Discussions

In this chapter, we improved the Projection heuristic [Pic18] to outperform the existing K-Way and TSP methods. The K-Way strategy provides *regular* clusters, where the close nodes of the graph are gathered together. This kind of clustering improves the compressibility of the separator cluster interaction blocks, which reduces the POTRF(A_{22}) cost. However, it does not provide large enough data sizes to take advantage of the underlying architectures. On the other hand, the TSP method can obtain fewer and larger off-diagonal blocks by gathering similar contributions together. As a result, it reduces the HERK(A_{21}, A_{22}) cost. Although this method provides a good sparse structure, it does not improve the separator compressibility.

The Projection heuristic is proposed to get advantage of these two methods in an optimal way and provide an efficient admissibility criterion for the solver. In this heuristic firstly the strong interaction nodes are gathered (preselected) together to be able to determine the poorly compressible blocks. As they are *uncompressible*, the corresponding nodes are clustered through the TSP method to only improve the sparse structure. On the other hand, the non-preselected node groups are clustered through K-Way to increase the compressibility of the cluster interaction blocks. However, as this method is not optimal yet, we proposed some improvements for this heuristic in our work.

The experiments on the *Geo_1438* matrix in Section 3.5.1 validated the effectiveness of the admissibility criterion of this heuristic. Moreover, these experiments showed the necessity of the late compression on the poorly compressible blocks. As a result, knowing that the *uncompressible* blocks should be compressed on the fly, we proposed to improve the compressibility of these blocks. As the poorly compressible blocks are the interaction blocks of the trace nodes, we proposed to perform a *regular* clustering (K-Way) on them. In addition, we proposed a new heuristic for the small connected components that were originally merged together. Indeed, as they are spread over the graph, their clustering is *irregular*. Therefore, we proposed to merge them with their closest neighbor for improving the regularity of the clusters, and thus their compressibility.

We provided our optimized Projection heuristic (OptProj) performance results through the last version of the PaStiX solver. In the experiments of our contributions with *regular* clusters, we could observe an improvement compared to the original version of the heuristic. However, it has a lower impact than expected compared to the existing methods through the OptProj heuristic versions. It is because choosing the preselected vertices is a difficult problem in practice. Choosing a lot of vertices can reduce the compressibility, while not choosing enough vertices can lead the connected components to not be well-separated.

As another compressibility criterion, we can also change our perspective to use the fill-in levels concept of the incomplete factorization. In the next chapter, we explain this fill-in levels based heuristic and prove the effectiveness of this method.

Chapter 4

ILU Levels Based Preselection Heuristic

Direct solvers are widely used for their robustness, despite their high cost in terms of memory and time. As mentioned in Section 1.2, low-rank representations are introduced into direct solvers to tackle these problems. Through the block low-rank (BLR) representation within sparse solvers, dense blocks of the sparse matrix are approximated into lower dimensional matrices. This approximation successfully reduces the memory and/or computational complexity problems with a controlled precision loss. A detailed study on low-rank approximations of dense matrices using various methods can be found in Chapter 2.

As explained in Section 1.3, in sparse supernodal solvers, reducing both the time and the memory footprint through the low-rank representations is challenging. In these solvers, various block sizes are involved in the block updates. The cost of the low-rank update depends on the largest block involved, contrary to the cost of full-rank update that only depends on the contribution size. That is why, the update operation might be even more expensive than the full-rank one.

The sparse supernodal direct solver PaStiX offers two opposite strategies: favor memory peak reduction over time-to-solution (*Minimal Memory*), or prefer time-to-solution by delaying the data compression (*Just-In-Time*). As explained in Section 1.4, the former suffers from the costly update operations, while the latter reduces it without improving the memory usage compared to the full-rank solver. Therefore, finding a compromise between time and memory footprint reduction becomes crucial.

Identifying the potential compressibility of each block is a key problem to benefit from both strategies in PaStiX. By compressing the most compressible blocks as soon as possible, we can significantly reduce the memory footprint of the solver. On the other hand, by delaying the compression of the poorly compressible blocks, we can get advantage of cheap update operations on them, without degrading the memory footprint much.

In Chapter 3, we study a heuristic to tackle this problem, where the unknowns are further ordered after the nested dissection algorithm. This heuristic aims at three improvements: 1) the uncompressible blocks are gathered together, 2) the remaining blocks have a better compressibility and 3) the sparse structure of the matrix is improved. The heuristic has two steps. Firstly, the data that represents strong interaction is gathered

together. The resulting strong interaction blocks represent the poorly compressible blocks (named preselected blocks), for which *Just-In-Time* is more adapted, avoiding the overhead of *Minimal Memory* with expensive low-rank updates. In addition, the remaining blocks represent weak interactions, so that they are more compressible. At the second step of the heuristic, the data is further ordered to improve the compressibility and sparse structure of the matrix. For large matrices, the sparse structure improvement is necessary to reduce the number of memory accesses, and thus the time-to-solution. Unfortunately, this heuristic does not improve the solver as much as we expected and needs to be further improved.

In this chapter, we propose a new heuristic based on incomplete factorization to define levels of admissibility (compressibility) for the blocks (see Section 1.2.1). Through this heuristic, we can successfully identify the compressible blocks and exploit the low-rank features of the solver. We prove the effectiveness of our heuristic with numerical experiments in both sequential and parallel environments. Note that the related work of this chapter can be found in Chapter 1.

In Section 4.1, we start by providing background details on the general idea behind incomplete LU factorization. In Section 4.2, we detail the new heuristic, which defines the non-compressible blocks to exploit the existing compression strategies in PaStiX. We present our numerical results in Section 4.3. Here, we first prove the applicability of our heuristic as an admissibility criterion in Section 4.3.1. In Section 4.3.2, we analyse the experiments on a large set of matrices in sequential, while in the remainder of Section 4.3, the multi-threaded experiments are discussed. In Section 4.4, we conclude on the results of this work and its perspectives.

4.1 Background

The incomplete LU (ILU) factorization is a well-known method to get a general preconditioner for the iterative solvers [HRR08]. It is an approximated version of the LU factorization, where part of the information is dropped [Saa03] and has the form $A \approx LU = LU + R$. Here, the matrix R carries the negative values of the dropped elements.

Multiple dropping heuristics have been studied through the years targeting either parallelism and efficiency of the implementation or a good numerical accuracy of the preconditioner. Among the dropping heuristics, there exist non-numerical solutions: using the position, or the fill-in levels [DD97], ILU(k), and numerical solutions: using a threshold [KK97a] value, ILU(τ). Through the years, their block-based variants have been studied to improve their efficiency [Gup17,BSV21,ARF+19]. While we may consider low-rank solvers as a solution similar to ILU(τ), we exploit in this study the block ILU(k) approach to enrich the block low-rank solver. The ILU method with the fill-in levels definition, as we use, is first suggested in 1981 [Wat81] and improved by a graph-based definition through fill-path theorem [RT78] in [HP01]. Figure 4.1 illustrates the idea of the fill-in levels that is strongly related to the ordering of the unknowns. On the left, the

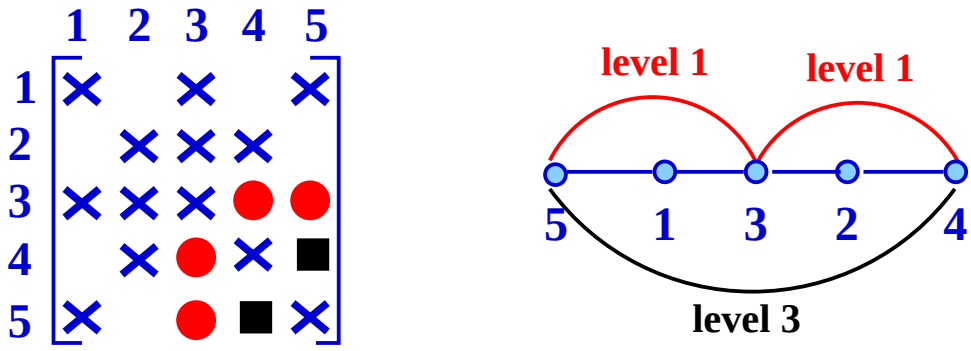


Figure 4.1: An adjacency matrix (on the left) and its associated graph (on the right). Fill-in entries that may occur during the numerical factorization are represented in red (level 1) and black (level 3).

matrix non-zeroes pattern is represented, where the blue crosses are the original entries. On the right, the graph associated with this matrix is shown. During the numerical factorization, some entries may become non-zeroes (fill-in). On Figure 4.1, these entries are represented in red and black, both on the matrix and as new edges on the graph. We can define the fill-in level as the length of the path connecting the two unknowns in the original graph. The path connecting 3 and 5 (and 3 and 4) in red goes only through 1 (resp. 2). Thus, the level of the fill-in between these two unknowns is 1. We can also see that 4 and 5 are connected at a level 3 (the path goes through 1, 3 and 2). As the fill-in level gets higher, the value of the new entry becomes smaller as it represents far interactions in the graph. That is, the fill-in levels can represent the generalized algebraic distance in the low-rank strong admissibility condition (see Section 1.2.1), without any knowledge of the geometry. Therefore, the ILU factorization can be implemented by dropping the values which have higher fill-in levels than a predefined maximum level. Similarly, this procedure can be applied in a block-wise fashion. Thus, considering the block fill-in levels as an admissibility criterion for low-rank compression, we can algebraically decide on which blocks the compression should be delayed to reduce the overhead of the fully structured updates.

4.2 The new fill-in level based compressibility heuristic

As mentioned in Section 1.4, the *Minimal Memory* strategy suffers from the complexity overhead of the LR2LR update kernel. We propose to exploit ILU fill-in levels to identify, at low cost, the blocks with large ranks. These blocks will increase the cost of the update step while providing only a small memory reduction. Once identified, it is possible to postpone the compression of these blocks as late as possible to replace the LR2LR kernels by LR2FR. This comes at the cost of a controlled memory overhead if the identification is correct. In Section 4.1, we mentioned that as the ILU fill-in levels get larger, the magnitude of the entries gets smaller. Thus, blocks with large level values should have

smaller ranks and can be kept compressed to save memory, while blocks with small level values may have high ranks and are better candidates for delayed compression.

Algorithm 11 Cholesky-based ILU fill-in levels initialization

1: **for all** block L_{ij} in L **do** ▷ Initialize the block fill-in levels in L
2: $lvl(L_{ij}) = (A_{ij} \neq 0)\ ?\ 0 : \infty$
3: **for all** column block L_{*k} in L **do** ▷ Set the block fill-in levels
4: **for all** block L_{ik} in L_{*k} **do**
5: **for all** block L_{jk} in L_{*k} (with $j > i$) **do**
6: $lvl(L_{ij}) = min(lvl(L_{ij}),\ lvl(L_{ik}) + lvl(L_{jk}) + 1)$

Algorithm 11 presents the main steps to compute the fill-in levels of the blocks. This algorithm performs the same loops as the numerical factorization focusing only on the fill-in level information and the symbolic structure of the factorized matrix L. Initially, all blocks are considered with level 0, if they are part of the original matrix A, or ∞ if they are created by fill-in (lines 1-2). Then, the main loop updates the levels according to the formula given in [Saa03], which is adapted to the block-wise algorithm (Line 6). Note that as PaStiX uses the symmetric pattern structure of $A+A^T$ even for LU factorization, Cholesky-based algorithms are presented. For strongly non-symmetric matrices, the fill-in levels of the blocks in L and U are computed separately for better identification. This very cheap algorithm is in fact fully integrated within the parallel matrix initialization to avoid an extra loop over the structure, and such that its cost is completely hidden to the user.

Algorithm 12 Cholesky BLR factorization with *maxlevel* admissibility

1: **for all** block L_{ij} in L **do** ▷ Initialize L in compressed version
2: **if** $lvl(L_{ij}) > maxlevel$ **then** ▷ Compress the highly compressible blocks
3: Compress(L_{ij})
4: **for all** column block A_{*k} in A **do** ▷ Numerical factorization
5: Factorize(A_{kk})
6: **for all** block L_{ik} in L_{*k} **do** ▷ Off-diagonal blocks of the column block
7: **if** $lvl(L_{ik}) <= maxlevel$ **then** ▷ Compress the poorly compressible blocks
8: Compress(L_{ik})
9: Solve(L_{kk}, A_{ik})
10: **for all** block L_{jk} in L_{*k} (with $j <= i$) **do**
11: Update(L_{ik}, L_{jk}, A_{ij})

Now that the fill-in levels are computed, the numerical factorization can be adapted to exploit this information. Algorithm 12 presents the proposed algorithm with a generic parameter *maxlevel*, which allows to set the new admissibility criterion (in orange color). First, lines $1-3$ compress the admissible blocks, which have a fill-in level larger than *maxlevel*. These blocks have the smallest ranks and are compressed before starting the

numerical factorization loop (like *Minimal Memory* in red). Thus, they will be involved in LR2LR updates and are the most important ones to compress to reduce the memory footprint. On the other hand, the non-admissible blocks will be involved in LR2FR updates to reduce the flops count overhead while inducing a small memory overhead. These blocks are still compressed (lines $6-8$), just after the factorization of the diagonal block (like *Just-In-Time* in blue), to reduce the cost of the following operations: solve and updates. In the remainder of the chapter, we will refer to this BLR sparse factorization as $ILU(k)$, with k the maximum level of the admissibility criterion. Note that choosing the right k value for a given problem is important. As a matter of fact, the larger the number of non-admissible blocks, the higher the memory overhead.

It is important to observe that $ILU(-1)$ is the *Minimal Memory* scenario as all the admissible blocks are compressed during the initialization. On the opposite, $ILU(\infty)$ corresponds to the *Just-In-Time* scenario as all blocks are compressed only after all the cumulative updates are performed.

As a consequence, in this chapter, we propose to improve supernodal methods by trading a small memory overhead for lower flops count and a better time-to-solution. For that purpose, we implement our solution in the sparse direct solver PaStiX [Pic18, PDF+18], which supports the BLR compression scheme.

4.3 Experiments

In the following sections, the experiments are run for a set of 31 real case matrices taken in the SuiteSparse Matrix Collection [DH11]. The experimental setup is similar to Section 3.5. The times shown are the average of 3 runs on each matrix. Data reported in the graphs are only related to the numerical factorization. The solve step is never considered as it is not impacted by this new algorithm. Interested readers can find an experimental backward error study on the *Minimal Memory* and *Just-In-Time* strategies in [Pic18] that shows the numerical stability of the methods. Here, the author shows that both methods fulfill the accuracy requirement given by users, despite the fact that *Just-In-Time* is more accurate thanks to the late compression scheme in the factorization. Note that as our new heuristic is an intermediate solution between *Minimal Memory* and *Just-In-Time*, it is also stable.

4.3.1 Compressibility statistics

We first want to validate the hypothesis that using the fill-in levels is a good heuristic to classify the blocks based on their compressibility ratios. Figure 4.2 reports the cumulative memory consumption for all the 31 studied matrices in our experiments with three different tolerance criteria and by fill-in levels. The purple bars show the memory consumption of the structurally non-admissible blocks (too small to be compressed) and is identical for all precisions. The red part corresponds to the admissible blocks. The dark red is the memory footprint when they are compressed. It naturally increases with

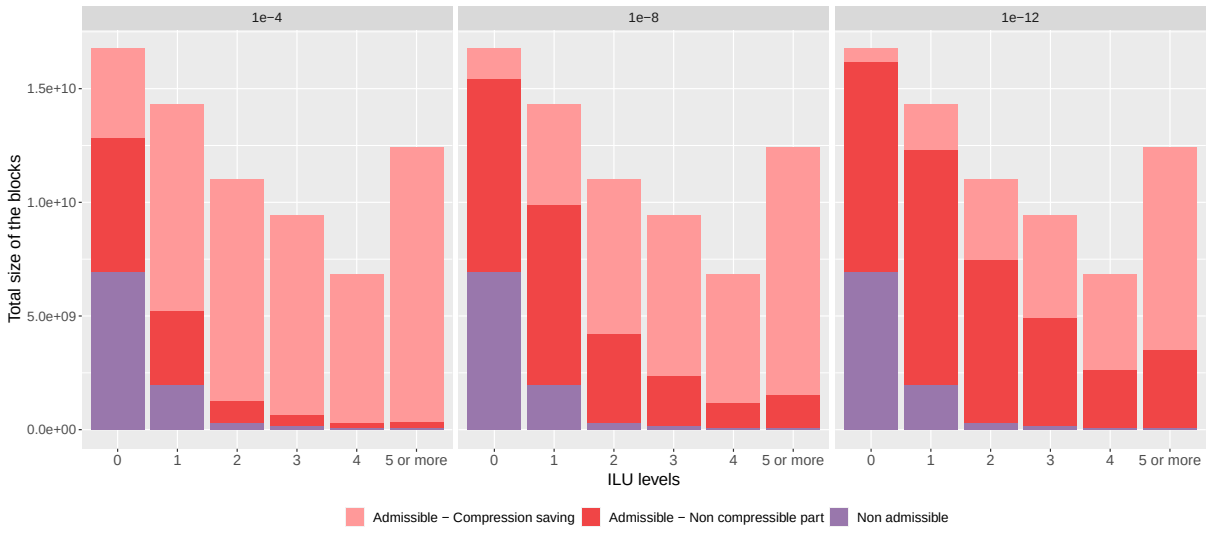


Figure 4.2: Potential memory saving based on the tolerance criterion and fill-in levels. The bars report the cumulative memory of the 31 matrices. Purple represents the memory consumed by blocks below the size criteria, and red the memory of the admissible blocks for compression. Light red is the portion that can be saved when compressed.

a higher precision. The light red shows the amount of memory that can be saved by compressing the blocks. One can observe that the lower the precision requirement, the higher the gain.

These results show that the compression ratio of the admissible blocks increases with the levels. It confirms the original hypothesis that fill-in levels can help to better tune the admissibility criterion in order to save flops for a controlled memory overhead. Furthermore, this parameter needs tuning to adapt to the tolerance. One can see that for a tolerance of $1e^{-12}$, only levels greater than 2 offer more than 40% memory savings, while all levels at $1e^{-4}$ provide more than 40% memory reduction. As a consequence, the fill-in level used to define the admissibility criterion will need to be adapted to both the tolerance and the maximum memory overhead defined by the user.

4.3.2 Impact of the fill-in level heuristic on the sequential version

This section discusses the sequential experiments. Figure 4.3 shows the memory peak, factorization flops and factorization time profiles obtained for different precisions. We study the impact of the first fill-in levels (0 to 4) with respect to *Minimal Memory* (-1) and *Just-In-Time* (∞). Each curve represents the number of matrices within a percentage overhead of the best solution for each metric and matrix.

First, as expected, the lower the fill-in level chosen for admissibility, the lower the memory peak of the solver. One can observe that the impact of the fill-in level increases as the precision decreases. This confirms the trend already observed on Figure 4.2. $ILU(\infty)$

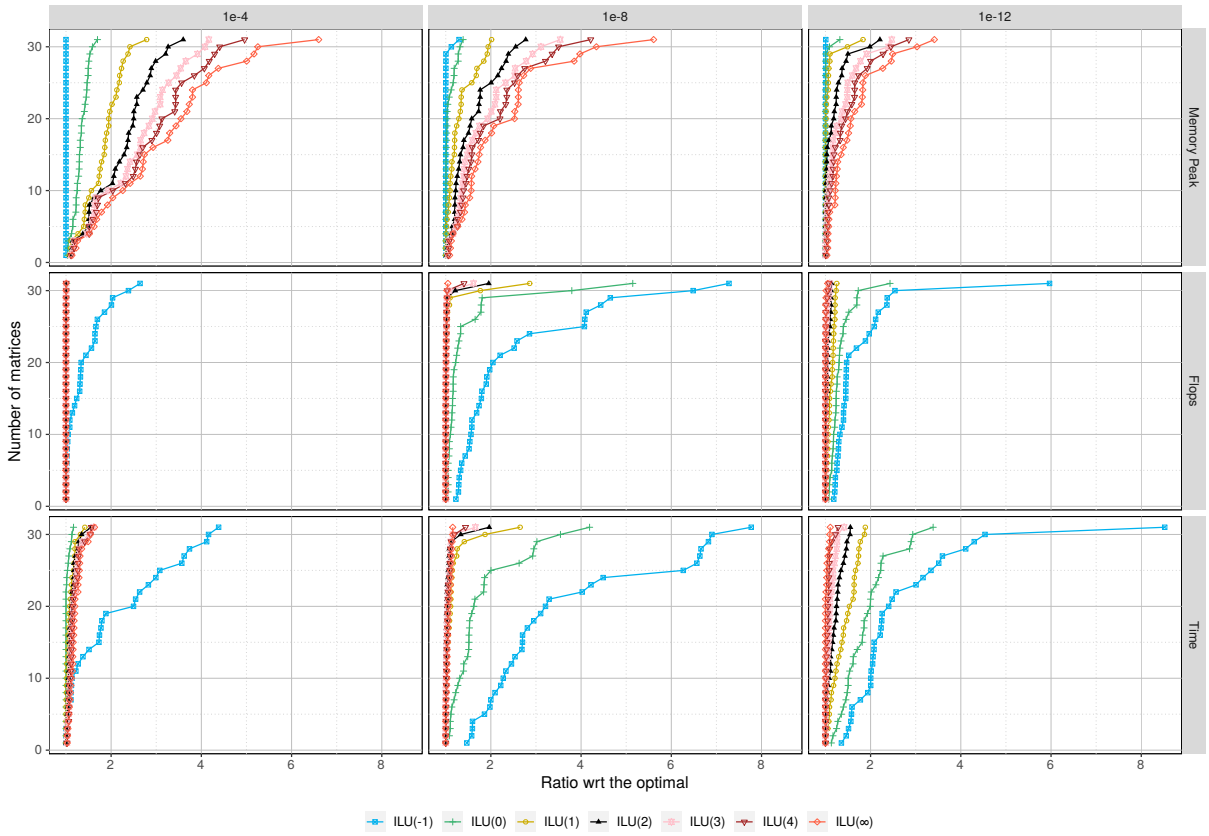


Figure 4.3: Memory peak, factorization flops and factorization time profiles with different precisions for sequential runs. Each color stands for a different $ILU(k)$ level. The x-axis shows percentages with respect to the best method for each metric and matrix, while y-axis represents the matrix count in cumulative way.

consumes up to 6.6 times more memory at $1e^{-4}$, while it drops to 3.4 times at $1e^{-12}$. Additionally, at this high precision, high levels of fill-in are able to reach the best memory peak. This means that potential flops reduction is possible without negatively impacting the memory.

Second, when observing the flops count evolution, the results are naturally reversed. The higher levels of fill-in are better to generate less flops. One can observe that for low precision ($1e^{-4}$), a level of 0 is enough to reach the same flops count as the *Just-In-Time* scenario ($ILU(\infty)$). When increasing the precision, more levels need to be considered non-admissible to lower the flops count to its minimal value. Except some corner cases, levels 1 or 2 are enough for $1e^{-8}$, and respectively 3 or 4 for $1e^{-12}$.

Finally, the time profiles follow the same trend as the flops profiles, with larger differences between the $ILU(k)$ methods. This can be explained by the disparity of the LR2LR and LR2FR efficiency, as well as their variation in number that may increase the phenomenon already observed on flops. Thus, one can observe that $ILU(0)$ is the

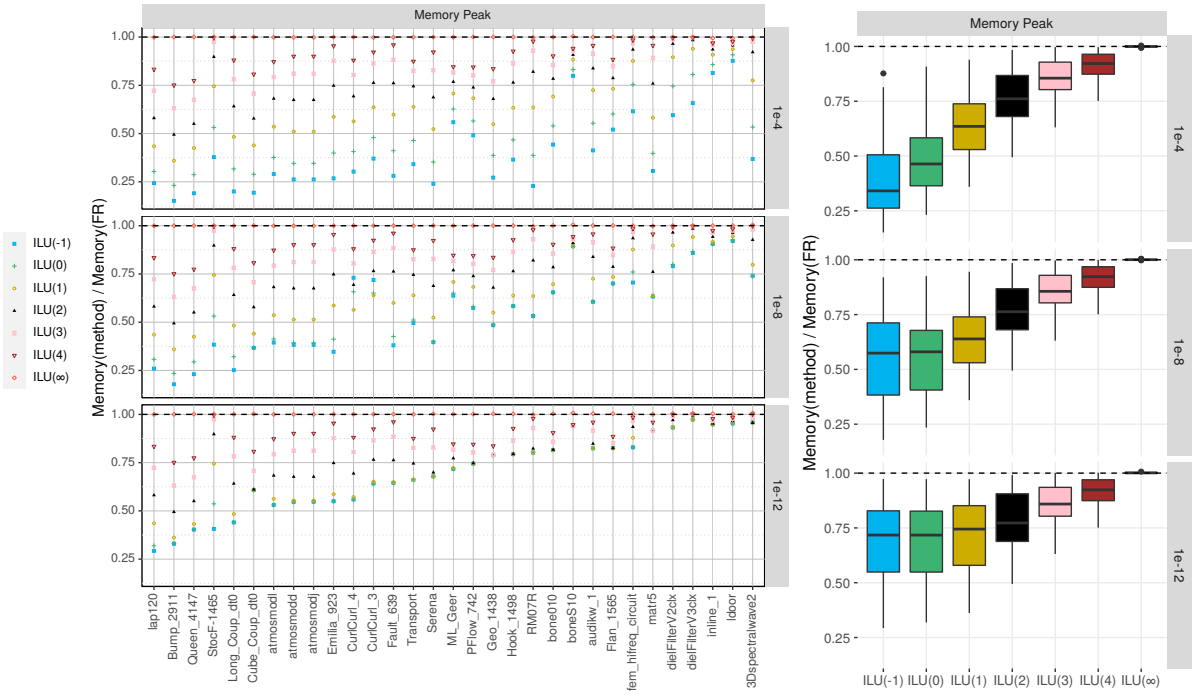


Figure 4.4: Memory peak ratio of the $ILU(k)$ heuristic with respect to full-rank on the 31 test matrices. On the left, the detailed information is presented for each matrix and precision. On the right, the information is summarized with boxplots showing minimum, maximum, median, first quartile and third quartile for each fill-in level.

best average solution at $1e^{-4}$. It even outperforms the *Just-In-Time* strategy by a factor up to 1.4×. To explain this performance, we recall that in the *Just-In-Time* strategy, many null-rank blocks are allocated and later compressed, while in the new $ILU(k)$ heuristic they may never be allocated. Indeed, they are originally null blocks and at low precision they may receive only null contributions. Thanks to these savings, $ILU(0)$ at this tolerance almost doubles the memory footprint in the worst case with respect to the *Minimal Memory* strategy, but it remains 0.23× the memory consumption of the $ILU(\infty)$ and the full-rank versions.

The observations in higher precisions are similar to the lower precision, but with higher fill-in levels. At $1e^{-8}$, only the levels above 1 compete with the *Just-In-Time* strategy in terms of time, as well as providing a controlled memory overhead with respect to the best solution.

At $1e^{-12}$, the levels higher than 3 are required to get the best factorization time, which reduces the gain one can obtain on the memory footprint. However, solutions with level 1 is a good compromise at this precision. It is up to 8.5× faster than $ILU(-1)$, with only up to 1.49× more memory usage.

Figure 4.4 presents the detailed ratios of the memory peak of the different levels of admissibility with respect to the full-rank solution. On the left figure, the matrices are

ordered by families and by increasing the memory ratio of $ILU(-1)$ at a tolerance of $1e^{-12}$. As we can observe, the compression ratio of all matrices varies a lot with the matrices and the tolerances. It also reflects the fact that increasing the fill-in level does not necessarily mean an extra memory usage in high precision problems as the dots are merged together. On the right figure, this trend is summarized with boxplots representing the average gain. One can observe that the *Just-In-Time* ($ILU(\infty)$) strategy has the same memory peak as the full-rank version, and that the lower the level, the lower the memory peak. One can also observe that increasing the level of admissibility at low precision seems to greatly increase the memory footprint with respect to the best one. However, the impact remains moderate for high precision and it can be afforded to highly reduce the time-to-solution, as observed in Figure 4.3.

The summary on the right of Figure 4.4 confirms the fact that the memory consumption increases with the higher fill-in levels and with the higher precision. The results at $1e^{-12}$ for the new heuristic at levels 1 and 2 show interesting results. As a matter of fact, they provide in average 25% memory improvement compared to the full-rank version. Note that at this precision, even $ILU(\infty)$ does not manage to efficiently accelerate the full-rank version on the five right-most cases, which are poorly compressible. On the other matrices, in addition to this memory saving, levels 1 and 2 are also faster than the full-rank version.

To conclude, it is difficult to give a single level as the optimal solution. However, depending on the problem, as well as the precision and memory restrictions, the level can be tuned to provide a solution that outperforms the *Minimal Memory* strategy in terms of time, for a small controlled memory overhead. Moreover, it can even have a speedup compared to the *Just-In-Time* strategy, while reducing the memory footprint.

4.3.3 Impact of the fill-in level heuristic on the multi-threaded version

This section presents the results of the previous experiments in a multi-threaded environment with 24 threads. Figure 4.5 shows the time profiles of the multi-threaded numerical factorization on the set of 31 matrices. Memory peak and flops are not reported as they are identical to the sequential ones.

We can observe that the impact of the new heuristic is even greater in the multi-threaded environment. The shift in the memory usage induced by the heuristic improves the memory bandwidth in the multi-threaded context and allows getting better performance. The $ILU(k)$ heuristic performs better respectively with a level of 0, 2 and 4, for tolerances of $1e^{-4}$, $1e^{-8}$, and $1e^{-12}$. The proposed solution outperforms the *Just-In-Time* strategy as it can be especially seen at $1e^{-4}$. In this parallel context, the large memory reduction improves the memory contention of the threads, while it merely degrades the flops count with respect to the $ILU(\infty)$. $ILU(0)$, which initially stores only the blocks of the original matrix A, clearly outperforms the other versions at low precisions. When increasing the precision ($1e^{-8}$ or $1e^{-12}$), the extra flops count for small values of k degrades the performance. However, one can observe that $ILU(2)$ and $ILU(4)$

(at $1e^{-8}$ and $1e^{-12}$, respectively) are good competitors with $ILU(\infty)$ in terms of time, while they still induce memory savings as opposed to $ILU(\infty)$.

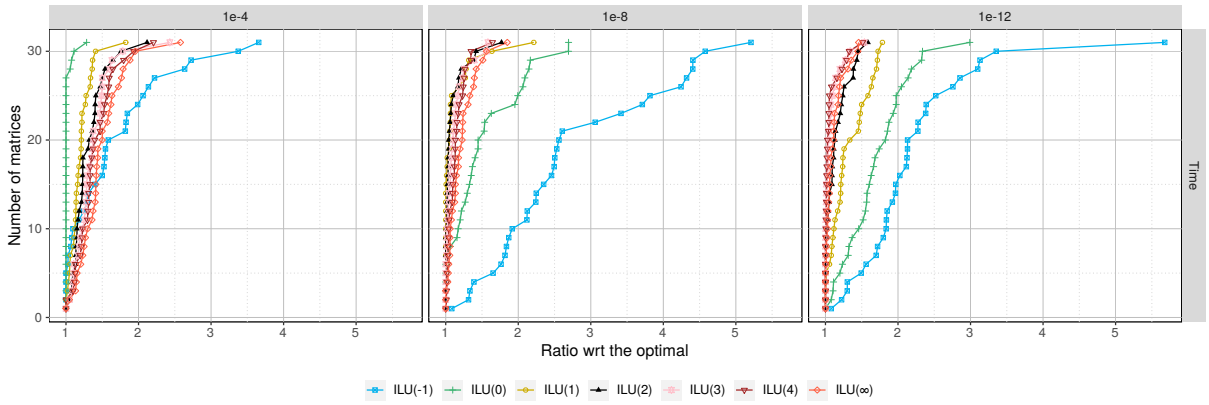
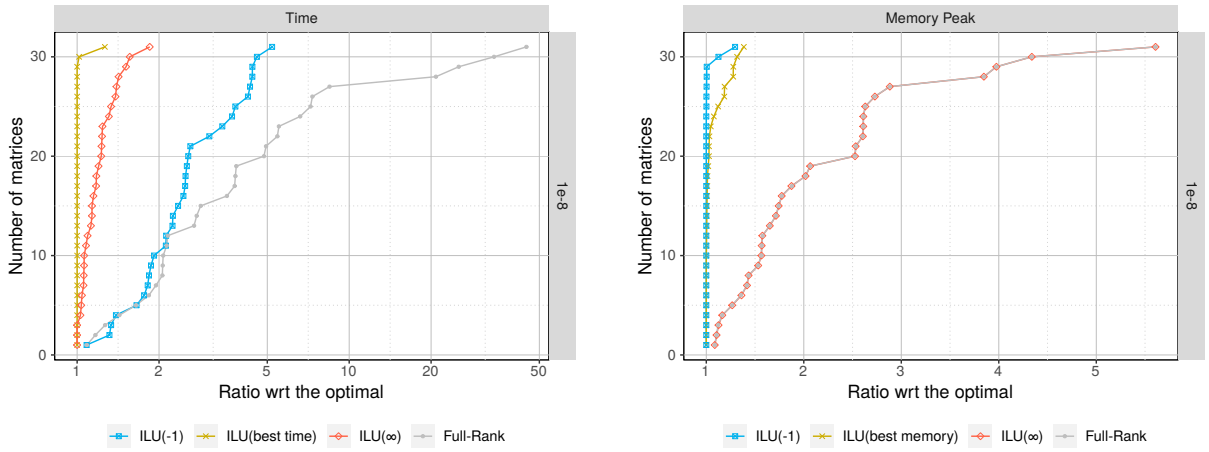


Figure 4.5: Time profiles of the multi-threaded runs for different precisions. Each color shows a different $ILU(k)$ level result.



(a) Execution time of $ILU(k)$ heuristic compared to existing methods, where the level is chosen between 0 and 4 to obtain the best time.

(b) Memory peak of $ILU(k)$ heuristic compared to existing methods, where the level is chosen between 0 and 4 to obtain the smallest memory consumption.

Figure 4.6: Best results achievable for time and memory with $ILU(k)$ heuristic, tuning the fill-in level between 0 and 4, at $1e^{-8}$ and using 24 threads.

To better highlight the high gain obtained with our new heuristic, we compare it to the full-rank solver and to the previous low-rank strategies existing in the PaStiX solver. Figure 4.6 presents the time profiles of the multi-threaded numerical factorization with 24 threads on the left, as well as the memory peak at $1e^{-8}$ precision on the right. On

Figure 4.6a, $ILU(best\,time)$ refers to the best factorization time obtained with the new heuristic, where the fill-in level is taken in the range $ILU(0)$ to $ILU(4)$ included. Similarly, in Figure 4.6b the $ILU(best\,memory)$ stands for the best memory consumption reachable with a fill-in level in the same range. Note that on Figure 4.6b, the memory consumption of the full-rank solver and the $ILU(\infty)$ are mingled, as they are identical. The left figure shows that the new heuristics is up to 45 times faster than the full-rank solver, and is up to almost 2 times faster than the previous fastest implementation $ILU(\infty)$ except for the two `CurlCurl` matrices which would require higher fill-in levels to get better performance due to the high ranks of their contributions.

Overall, the proposed heuristic, if correctly tuned, is the fastest method for all matrices and outperforms the $ILU(\infty)$ solution while providing an effective memory footprint reduction. Furthermore, the memory footprint reduction is similar to the one obtained with the $ILU(-1)$ strategy in two thirds of the cases (using a fill-in level of 0 in most cases), and reaches at most a 39% overhead in the last set of matrices. One can notice that in two cases (the two `CurlCurl` matrices), the new heuristic outperforms the $ILU(-1)$ strategy thanks to a slight change in the order of allocation of the blocks. In these specific cases, the best memory peak is even obtained with a fill-in level of 1 that better delayed the workspace allocation to compress the blocks of the matrix.

4.3.4 Study of the Serena matrix in multi-threaded environment

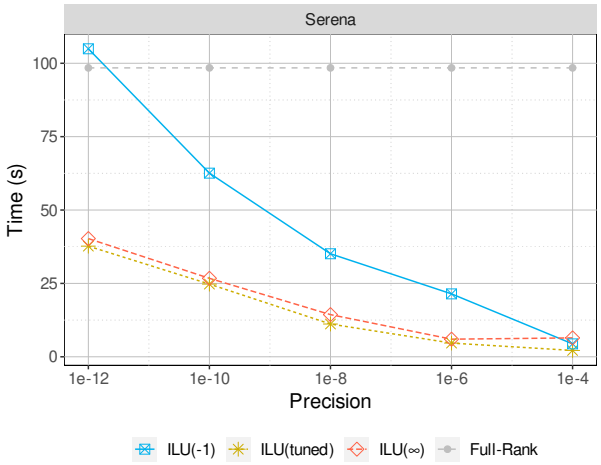


Figure 4.7: Serena matrix time results of different precisions with 24 threads. The $ILU(tuned)$ corresponds to the smallest level, which runs faster than $ILU(\infty)$.

In this section we focus the study on one of the large matrices from the collection: the `Serena` matrix. The size of the matrix is $N = 1\,391\,349$ for $32\,961\,525$ of non zeroes and it requires 28.6 TFlops to be factorized in full-rank, which makes it a good average test case of the collection. Figure 4.7 presents a performance study of different compression tolerances with 24 threads. The ILU(tuned) corresponds to the use of levels chosen for

Tolerance	$ILU(tuned)$
$1e^{-4}$	$ILU(0)$
$1e^{-6}$	$ILU(2)$
$1e^{-8}$	$ILU(2)$
$1e^{-10}$	$ILU(4)$
$1e^{-12}$	$ILU(4)$

Table 4.1: Corresponding levels for $ILU(tuned)$ at each precision.

NbThreads	$1e^{-4}$			$1e^{-8}$			$1e^{-12}$			Full-Rank
	$ILU(-1)$	$ILU(0)$	$ILU(\infty)$	$ILU(-1)$	$ILU(2)$	$ILU(\infty)$	$ILU(-1)$	$ILU(4)$	$ILU(\infty)$	
1	26.2	21.1	26.5	330.7	109.7	111.5	1154.9	468.1	446.4	1025.9
4	7.4 (3.5)	6.6 (3.2)	9.7 (2.7)	91.8 (3.6)	32.8 (3.3)	35.7 (3.1)	318.2 (3.6)	129.2 (3.6)	124.7 (3.6)	279.3 (3.7)
6	5.5 (4.8)	4.8 (4.4)	6.8 (3.9)	64.5 (5.1)	24.2 (4.5)	27.6 (4.0)	230.6 (5.0)	92.8 (5.0)	91.7 (4.9)	203.2 (5.0)
12	3.5 (7.5)	2.5 (8.4)	4.7 (5.6)	35.1 (9.4)	13.8 (7.9)	16.7 (6.7)	131.7 (8.8)	52.8 (8.9)	55.4 (8.1)	137.1 (7.5)
24	4.4 (6.0)	2.2 (9.7)	6.4 (4.1)	35.1 (9.4)	11.2 (9.8)	14.4 (7.8)	104.9 (11.0)	37.7 (12.4)	40.3 (11.1)	98.4 (10.4)

Table 4.2: Factorization times for the Serena matrix. Speedup with respect to the sequential runs are written inside parentheses.

each tolerance as reported in Table 4.1. The full-rank result is shown in the figure as a reference for the speedup observation. In conclusion, the new heuristic outperforms the former two solutions on all precisions with a small fill-in level of 0 to 4, relatively to the precision. As shown previously, at these levels, this solution even provides an important memory gain as opposed to the fastest existing solution $ILU(\infty)$.

Table 4.2 reports the detailed factorization times of the Serena matrix with different numbers of threads and tolerances. The values inside parentheses present the speedup compared to the sequential run of each method at the corresponding precision. The $ILU(k)$ heuristic levels are chosen as in Table 4.1. The level selection has been limited to the range 0 to 4 to ensure a useful memory saving compared to $ILU(\infty)$. That is why, with a small number of threads at $1e^{-12}$, our method cannot run faster than $ILU(\infty)$. However, the $ILU(k)$ heuristic benefits from a higher scalability, which allows it to quickly outperform other solutions as the number of threads increases. These results, while not reported in this chapter, are similar to the ones observed on the set of matrices used for the experiments. To conclude, the $ILU(k)$ heuristic, through the correct tuning of the levels, solves the original issue of the flops overhead of $ILU(-1)$. Furthermore, thanks to its better memory footprint, it provides a better scalability in parallel environments and outperforms the fastest original solutions.

4.4 Discussions

The behavior of sparse supernodal direct solvers using low-rank compression highly depends on when the compression is performed. On one hand, all admissible blocks can be compressed before the factorization (as it happens with the *Minimal Memory* / $ILU(-1)$ strategy). It allows high memory savings, but in the specific case of supernodal methods,

it induces an expensive flops overhead during the low-rank updates. On the other hand, admissible blocks can be compressed after they have received all their updates (as for the *Just-In-Time* / $ILU(\infty)$ strategy). It reduces significantly the flops count as in dense solvers, and thus the execution time. However, the memory peak of allocating all the L matrix blocks in full-rank, before the numerical factorization, still exists. In order to reduce this memory peak, the allocation of the blocks should be carefully delayed to their first access.

In this chapter, we proposed a new heuristic to estimate the compressibility of each block and constructed an algorithm that is a compromise between the two strategies mentioned previously. The new heuristic, named $ILU(k)$, identifies poorly compressible blocks similarly to the ILU method, which identifies the most important data. It relies on the block ILU fill-in levels to define an *algebraic* distance to compute low-rank admissibility of the blocks. The purpose of defining the admissibility is to propose an intermediate solution that accelerates the *Minimal Memory* solution, while it slightly increases the memory consumption. Moreover, it gives the chance to tune the levels according to the precision, the matrix properties, and the characteristics of the machine to better exploit the advantages of both the *Minimal Memory* and the *Just-In-Time* strategies.

The experiments that we conducted on a large set of 31 real matrices demonstrated that the $ILU(k)$ heuristic manages to identify efficiently the most compressible blocks. The solution proposed runs up to 5.2 times faster than *Minimal Memory* with only a 1.38 times increase of memory usage for high precision in both sequential and multi-threaded environments. Moreover, due to the elimination of the null blocks before the numerical factorization, the $ILU(k)$ heuristic is also able to run 1.4 times faster than the *Just-In-Time* strategy in sequential, with a much lower memory consumption (0.23 times less). In the multi-threaded environment, it even goes up to 1.84 times faster and outperforms it for most of the cases. We showed that through correctly tuned fill-in levels, the $ILU(k)$ heuristic can be used as an improved version of the existing strategies. It improves the numerical factorization in terms of both memory and time, and it improves the scalability for parallel environments.

Despite the high gain of the $ILU(k)$ heuristic, it remains difficult to know beforehand which level will provide the best improvement. Section 4.3 has shown that only the first levels are worth consideration, and that a clear trend appears on the level to use depending on the tolerance.

Conclusion and Future Work

Recently, the introduction of low-rank representations into sparse direct solvers allowed to successfully reduce their memory and/or time complexities. Block low-rank (BLR) format, compared to other alternatives, provides more flexibility in parallel environments and it is simpler to implement. We believe BLR format is more advantageous in supernodal solvers in terms of memory since the fronts inherent to the multifrontal solvers are not allocated in these solvers.

In this work, we focused on improving the sparse supernodal BLR solver PaStiX. This solver adopts the nested dissection method to obtain the coarse level matrix ordering to generate more sparsity in the factorized matrix and provide better parallelism. Following the nested dissection procedure, it uses the BLR approximation scheme on the resulting large dense blocks of the factorized matrix structure, exploiting the existing sparsity. We contributed PaStiX in three ways, each of which is explained in a devoted chapter. Now let us provide the conclusions and perspectives of each chapter (so each improvement). Then, we explain the long-term future work.

Compression kernels In Chapter 2, we studied and compared four different QR based compression methods as an alternative to the costly SVD kernel. We implemented all the methods in a similar fashion with similar numerical stopping criteria. Let us emphasize that determining the rank through a numerical criterion is new for the TQRCP and RQRRT methods that we studied. As the blocks of BLR representation are not large enough to take advantage of parallelism when compressing them, PaStiX adopts sequential compression kernels. Therefore, we only observed the useful matrix sizes in our context and in a sequential environment.

We tested the QR based compression kernels by using some generated matrices. Here, we saw that each method ensures a good stability and accuracy. Thus, we mainly concentrated on the obtained ranks and performance results through both the generated and the real-life case matrices. These results showed that the QRCP method is the most promising QR alternative for the BLR format blocks.

In the current PaStiX solver, compression is performed on the blocks of size less than 300. In the future, the blocks could be much larger after the hierarchical scheme implementation. Thus, although QRCP seems to be the best kernel within the up-to-date PaStiX version, the TQRCP can outperform it within the future solver. Therefore, comparing these two kernels should be our perspective after the hierarchical scheme

implementation.

In Chapter 2, we proved the efficiency of the QRCP methods on obtaining small enough ranks. Thus, we do not need to adopt more expensive approaches (like RQRRT) to reach closer results to SVD. For this reason, our further studies should mainly target to improve the performance, rather than the obtained ranks, compared to the QR methods we observed.

Block low-rank clustering In Chapter 3, we studied the Projection heuristic [Pic18]. This heuristic reorders the separator nodes to increase the separator compressibility and reduce the total number of updates within the solver. The former is applied through *regular* clusters, while the latter is done by gathering similar contributions together. It additionally aims to identify the poorly compressible (*uncompressible*) blocks, which represent the interaction blocks of the trace nodes.

In this work, we improved the Projection heuristic by configuring it correctly and clustering the separators in a *regular* way. We proposed two changes to obtain these *regular* clusters to increase the separator compressibility. Firstly, we merged the small node groups to a large neighbor cluster in the graph. In this way, the remainder component that includes the small node groups, which are spread along the graph, is eliminated. Secondly, we performed the K-Way (*regular*) clustering on the traces. Indeed, as we decided to compress the trace interaction blocks on-the-fly, improving their compressibility is also crucial. In our experiments, we showed an improvement compared to the original heuristic. However, we could not obtain the results we expected.

In this chapter, we implemented the Projection heuristic, where only the small components are merged to their neighbors (OptProj+ERC). Additionally, we tested the heuristic, where only the traces are clustered through the K-Way partitioning (OptProj+KWT). However, we did not conduct experiments on the version, which implements both improvements at the same time. This should be our perspective at this point.

This chapter is devoted to improve the clustering after the coarse level nested dissection partitioning. Alternatively, we can focus on improving the nested dissection method itself. For example, in order to have similar contributions from children to the separators, the nested dissection procedure can be changed in a way to align the traces on the separators better [Pic18]. As a result, we can increase the compressibility through well-separated large connected components, by getting advantage of the symmetric structure.

ILU levels based preselection heuristic In Chapter 4, our purpose is to identify the poorly compressible blocks with a different angle than Chapter 3. In this respect, we used the fill-in levels concept of the incomplete factorization in a block-wise manner. Then, according to our heuristic, a block is highly compressible if it has a high fill-in level. Otherwise, it is *uncompressible*. In both sequential and multi-threaded environments, we proved the efficiency of our heuristic.

In the experiments, we adopted predefined values (k) to determine how many levels are considered poorly compressible. Through the experiments with different k values,

we concluded that this value should be tuned according to memory and time restrictions of the applications. That is, smaller values provide reduced memory usage, while larger ones lead to a faster solver. However, determining the most advantageous level value is difficult. Therefore, as a perspective, we would like to introduce tuning techniques to automatically infer the best value of k depending on the given tolerance, as well as the properties of the machine and the matrix used.

In the future, we can use our ILU fill-in levels based compressibility information within the performance model of [MMPV22] as a parameter. In this way, we can take advantage of both heuristics at the same time.

As another future work, we can also identify the *uncompressible* blocks with another admissibility criterion. For example, we can set our criterion based on the k-way cluster distances. Then, the smaller the distance, the stronger the interaction (so *uncompressible*).

What is next? As mentioned before, thanks to the non-hierarchical clustering with similar size blocks, the BLR representation is simpler to implement and more convenient in parallel environments. For example, it is easier to arrange suitable block sizes to fit into the memory of a shared-memory implementation. Similarly, we can reach a good load balance in a distributed environment through the blocks of similar size in the BLR representation. However, hierarchical schemes provide better asymptotical complexities compared to flat ones. Therefore, some intermediate solutions between these techniques, like BLR-H [Ida18] and Multilevel-BLR [ABLM19a], are proposed to take advantage of both formats at the same time.

In BLR-H, the BLR scheme is used at the coarse level to obtain the blocks to be distributed. In this way, an efficient load-balanced distribution of the matrix is targeted. Then, the distributed parts are represented in a hierarchical way to improve the memory and computational complexity of the solver. In Multilevel-BLR, the BLR scheme is applied recursively to the full-rank diagonal blocks of the previous BLR representation. As a result, both BLR-H and Multilevel-BLR can provide complexities close enough to the hierarchical solvers (although not as good as them) and they are more promising to be efficient in the parallel context.

We already mentioned that our next step is to implement hierarchical schemes within the PaStiX solver to improve the time and the memory complexities. After this step, our long-term perspective should be to apply a hybrid scheme, like BLR-H and Multilevel-BLR. In this way, we can exploit both hierarchical and flat formats at the same time. Here, we aim to offer a sparse direct solver which provides the best intermediate solution between the most efficient in terms of time/memory usage and the most flexible one in parallel environments, compared to the other alternatives. Through these improvements, we target larger linear systems arising from different applications that need fast, accurate and memory efficient solutions in parallel environments.

As a short term perspective, we should conduct a detailed study of our solver in mixed precision. We know that it is possible to control the accuracy, time and memory of direct solvers through different floating-point arithmetic. For example, adopting low precision arithmetics (like single or half precision) can highly speed up the solver and reduce the

storage. However, low precision does not lead to high accuracy solutions. Alternatively, using higher precisions (like double precision) provides high accuracy despite the expensive time and storage requirements. We can use a mixed-precision solver to both ensure the required accuracy and reduce the solver cost. In this respect, we need a detailed comparison of our low-rank solver to the mixed-precision solver. Additionally, we need a clever implementation of the mixed-precision strategy within our low-rank solver to further improve it in terms of time and storage, while ensuring a given accuracy.

As another perspective, we can improve the low-rank feature of our solver to better adapt to the domain decomposition application needs. Domain decomposition methods [QV99, DW94, Mat08] can be used to reduce the time-to-solution of sparse linear systems. They are applied by using the underlying geometry and the partial differential equation before the discretization step. Here, the main idea is to split the domain into subdomains, each of which has a new boundary condition on the interface. Then, the subdomains are solved separately in a way that the solution continuity is ensured through the interface. The solution of each discretized subdomain is performed through some algebraic solvers like direct methods, iterative methods or hybrid methods. When we reorder a subdomain matrix into interior and interface subparts, the former subpart is sparse and the latter is dense. Then, in the interior part, it is more convenient to adopt a robust solver with high accuracy. Thus, a low-rank solver with high accuracy threshold can be used for these parts. On the dense interface parts, we prefer an iterative solver to improve the time-to-solution, where a preconditioner is needed for the fast convergence. For the preconditioning aim, a low-rank direct solver with low accuracy threshold can be applied. As we can observe, we can use a low-rank direct solver in both the interior and interface parts within a domain decomposition application. However, the parameters of the solver should be adapted depending on the matrix parts that it is used on. In this respect, we should automatically tune the low-rank parameters of the PaStiX solver within the domain decomposition applications.

Bibliography

[AAB+15] Patrick R. Amestoy, Cleve Ashcraft, Olivier Boiteau, Alfredo Buttari, Jean-Yves L'Excellent, and Clément Weisbecker. Improving Multifrontal Methods by Means of Block Low-Rank Representations. *SIAM Journal on Scientific Computing*, 37(3):A1451–A1474, 2015. 18, 24, 72, 77

[ABB+07] 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. 34

[ABCC06] David L. Applegate, Robert E. Bixby, Vašek Chvatál, and William J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006. 69

[ABLM19a] 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. *SIAM Journal on Scientific Computing*, 41(3):A1414–A1442, May 2019. 105

[ABLM19b] Patrick R Amestoy, Alfredo Buttari, Jean-Yves L'Excellent, and Théo Mary. Performance and scalability of the block low-rank multifrontal factorization on multicore architectures. *ACM Transactions on Mathematical Software (TOMS)*, 45(1):2, 2019. 72

[AD16] AmirHossein Aminfar and Eric 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. 17

[ADD96] 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. 11

[AGG+13] 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*, 60-61:23–30, 2013. CIVIL-COMP: Parallel, Distributed, Grid and Cloud Computing. 9

[ALMK17] Kadir Akbudak, Hatem Ltaief, Aleksandr Mikhalev, and David Keyes. Tile low rank cholesky factorization for climate/weather modeling applications on manycore architectures. pages 22–40, 2017. 15

[ALS+18] Sameh Abdulah, Hatem Ltaief, Ying Sun, Marc G. Genton, and David E. Keyes. Exageostat: A high performance unified software for geostatistics on manycore systems. *IEEE Transactions on Parallel and Distributed Systems*, 29(12):2771–2784, 2018. 15

[ARF+19] Hartwig Anzt, Tobias Ribizel, Goran Flegar, Edmond Chow, and Jack Dongarra. ParILUT - a parallel threshold ILU for GPUs. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 231–241. IEEE, 2019. 90

[ATNW11] 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. 21

[BBD+13] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Herault, and Jack J. Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013. 21

[Beb08] Mario Bebendorf. Hierarchical matrices. *Lecture notes in computational science and engineering, v.63 (2008)*, 63, 2008. 72

[BG05] Steffen Börm and Lars Grasedyck. Hybrid cross approximation of integral operators. *Numerische Mathematik*, 101:221–249, 08 2005. 29

[BPS05] Michael W. Berry, Shakhina A. Pulatova, and G. W. Stewart. Algorithm 844: Computing sparse reduced-rank approximations to sparse matrices. *ACM Trans. Math. Softw.*, 31(2):252–269, jun 2005. 29

[BR03] M. Bebendorf and S. Rjasanow. Adaptive low-rank approximation of collocation matrices. *Computing*, 70:1–24, 2003. 29

[BSV21] Matthias Bollhöfer, Olaf Schenk, and Fabio Verbosio. A high performance level-block approximate LU factorization preconditioner algorithm. *Applied Numerical Mathematics*, 162, 01 2021. 90

[CB15] 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. 17, 24

[CDGS10] S. Chandrasekaran, P. Dewilde, M. Gu, and N. Somasunderam. On the numerical rank of the off-diagonal blocks of schur complements of discretized elliptic pdes. *SIAM Journal on Matrix Analysis and Applications*, 31(5):2261–2290, 2010. 16

[CDHR08] Yanqing Chen, Timothy A. Davis, William W. Hager, and Sivasankaran Rajamanickam. Algorithm 887: Cholmod, supernodal sparse cholesky factorization and update/downdate. *ACM Trans. Math. Softw.*, 35(3), oct 2008. 24

[Cha87] Tony F. Chan. Rank revealing qr factorizations. *Linear Algebra and its Applications*, 88-89:67 – 82, 1987. 29

[CPA+20] Qinglei Cao, Yu Pei, Kadir Akbudak, Aleksandr Mikhalev, George Bosilca, Hatem Ltaief, David Keyes, and Jack Dongarra. Extreme-scale task-based cholesky factorization toward climate and weather prediction applications. *Proc. Platform for Advanced Scientific Computing Conference*, pages 1–11, 2020. 15

[DB08] Zlatko Drmac and Zvonimir Bujanovic. On the failure of rank-revealing qr factorization software – a case study. *ACM Trans. Math. Softw.*, 35, 07 2008. 37

[DD97] Timothy A Davis and Iain S Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. *SIAM Journal on Matrix Analysis and Applications*, 18(1):140–158, 1997. 9, 90

[DDCHD90] 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. 12, 31

[DER86] Iain S Duff, Albert M Erisman, and John K Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, Inc., USA, 1986. 9

[Dev12] Jay L. Devore. *Probability and Statistics for Engineering and Science - 8th edition*. Brooks/Cole Publishing Co., 2012. 41

[DG15] Jed A. Duersch and M. Gu. True blas-3 performance qrcp using random sampling. *arXiv: Numerical Analysis*, 2015. 29, 34, 40, 41, 42, 43

[DH11] 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. 81, 93

[DR83] Iain S. Duff and John Ker Reid. The multifrontal solution of indefinite sparse symmetric linear. *ACM Trans. Math. Softw.*, 9:302–325, 1983. 14

[DW94] M. Dryja and O. B. Widlund. Domain decomposition algorithms with small overlap. *SIAM Journal on Scientific Computing*, 15(3):604–620, 1994. 106

[EY36] C. Eckart and G. Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218, 1936. 29

[Fav09] Mathieu Faverge. *Ordonnancement hybride statique-dynamique en algèbre linéaire creuse pour de grands clusters de machines NUMA et multi-cœurs*. Theses, Université Sciences et Technologies - Bordeaux I, December 2009. 22

[Geo73] A. George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973. 11

[GH08] J. Gaidamour and P. Hénon. A parallel direct/iterative solver based on a schur complement approach. In *2008 11th IEEE International Conference on Computational Science and Engineering*, pages 98–105, 2008. 9

[GHLN88] 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. 9

[Gil11] A. Gillman. *Fast Direct Solvers for Elliptic Partial Differential Equations*. PhD thesis, University of Colorado, 2011. 24

[GKLB08] Lars Grasedyck, Ronald Kriemann, and Sabine Le Borne. Parallel black box h-lu preconditioning for elliptic boundary value problems. *Comput. Vis. Sci.*, 11(4–6):273–291, sep 2008. 16, 23

[GKLB09] L. Grasedyck, R. Kriemann, and S. Le Borne. Domain decomposition based $h - lu$ preconditioning. *Numerische Mathematik*, 112(4):565–600, 2009. 23

[GL81] A. George and J. W. H. Liu. Computer Solution of Large Sparse Positive Definite Systems. *Prentice-Hall, Englewood Cliffs, NJ*, 1981. 9, 72

[GLGR17] Pieter Ghysels, Xiaoye Sherry Li, Christopher Gorman, and François-Henry Rouet. A robust parallel preconditioner for indefinite systems using hierarchical matrices and randomized sampling. *IEEE International Parallel and Distributed Processing Symposium*, pages 897–906, 2017. 24, 72

[GLR+16] Pieter Ghysels, Xiaoye S. Li, François-Henry Rouet, Samuel Williams, and Artem Napov. An Efficient Multicore Implementation of a Novel HSS-Structured Multifrontal Solver Using Randomized Sampling. *SIAM Journal on Scientific Computing*, 38(5):S358–S384, 2016. 17, 24, 72

[Gup17] Anshul Gupta. Enhancing Performance and Robustness of ILU Preconditioners by Blocking and Selective Transposition. *SIAM Journal on Scientific Computing*, 39(1):A303–A332, 2017. 90

[GvL13] G. H. Golub and C. F. van Loan. Matrix computations. JHU Press, 2013. 9, 29, 30

[Hac99] Wolfgang Hackbusch. A Sparse Matrix Arithmetic Based on $\mathcal{H}$-Matrices. Part I: Introduction to $\mathcal{H}$-Matrices. *Computing*, 62(2):89–108, 1999. 17, 23

[Hac15] W. Hackbusch. *Hierarchical Matrices: Algorithms and Analysis*, volume 49. Springer, 12 2015. 16, 23

[HB02] Wolfgang Hackbusch and Steffen Börm. Data-sparse Approximation by Adaptive $\mathcal{H}^2$-Matrices. *Computing*, 69(1):1–35, 2002. 17

[HMT11] Nathan Halko, Per-Gunnar Martinsson, and Joel A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53(2):217–288, 2011. 15, 29, 35

[HP01] D. Hysom and A. Pothen. A scalable parallel algorithm for incomplete factor preconditioning. *SIAM J. Sci. Comput.*, 22:2194–2215, 2001. 90

[HRR02] 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. 13

[HRR08] P. Hénon, P. Ramet, and J. Roman. On finding approximate supernodes for an efficient block-ILU(k) factorization. *Parallel Computing*, 34(6):345–362, 2008. Parallel Matrix Algorithms and Applications. 90

[HSS08] Thomas Hofmann, Bernhard Schölkopf, and Alexander J. Smola. Kernel methods in machine learning. *The Annals of Statistics*, 36(3):1171 – 1220, 2008. 72

[Ida18] Akihiro Ida. Lattice h-matrices on distributed-memory systems. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 389–398, 2018. 105

[JNP21] Mathias Jacquelin, Esmond G. Ng, and Barry W. Peyton. Fast implementation of the traveling-salesman-problem method for reordering columns within supernodes. *SIAM Journal on Matrix Analysis and Applications*, 42(3):1337–1364, 2021. 73

[KK97a] G. Karypis and V. Kumar. Parallel threshold-based ilu factorization. *proceedings of the IEEE/ACM SC97 Conference*, 1997. 9, 90

[KK97b] George Karypis and Vipin Kumar. Metis—a software package for partitioning unstructured graphs, partitioning meshes and computing fill-reducing ordering of sparse matrices. 01 1997. 11, 72

[Lac15] 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. 73

[LFB+14] Xavier Lacoste, Mathieu Faverge, George Bosilca, Pierre Ramet, and Samuel Thibault. Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, pages 29–38, 2014. 22

[LN14] R. Luce and E. Ng. On the minimum flops problem in the sparse cholesky factorization. *SIAM Journal on Matrix Analysis and Applications*, 35(1):1–21, 2014. 11

[LWM+07] Edo Liberty, Franco Woolfe, Per-Gunnar Martinsson, Vladimir Rokhlin, and Mark Tygert. Randomized algorithms for the low-rank approximation of matrices. *Proceedings of the National Academy of Sciences*, 104(51):20167–20172, 2007. 29

[Mar15] Per-Gunnar Martinsson. Blocked rank-revealing qr factorizations: How randomized sampling can be used to avoid single-vector pivoting. *arXiv preprint arXiv:1505.08115*, 2015. 34, 44, 45, 49

[Mar17] T. Mary. *Block Low-Rank multifrontal solvers: complexity, performance, and scalability*. PhD thesis, Toulouse University, Toulouse, France, November 2017. 24, 25, 37

[Mat08] Tarek Mathew. *Domain Decomposition Methods for the Numerical Solution of Partial Differential Equations*, volume 61. 01 2008. 106

[MMPV22] Loris Marchal, Thibault Marette, Grégoire Pichon, and Frédéric Vivien. Trading performance for memory in sparse direct solvers using low-rank compression. *Future Generation Computer Systems*, 130:307–320, 2022. 25, 105

[MQOH17] Per-Gunnar Martinsson, Gregorio Quintana-Orti, and Nathan Heavner. randutv: A blocked randomized algorithm for computing a rank-revealing utv factorization. *ACM Transactions on Mathematical Software*, 45, 03 2017. 29

[MQOHvdG17] Per-Gunnar Martinsson, Gregorio Quintana-Orti, Nathan Heavner, and Robert van de Geijn. Householder qr factorization with randomization for column pivoting (hqrrp). *SIAM Journal on Scientific Computing*, 39(2):C96–C115, 2017. 34, 40

[MT11] Per-Gunnar Martinsson and Mark Tygert. A randomized algorithm for the decomposition of matrices. *Applied and Computational Harmonic Analysis*, 30:47–68, 01 2011. 29

[MV16] Per-Gunnar Martinsson and Sergey Voronin. A randomized blocked algorithm for efficiently computing rank-revealing factorizations of matrices. *SIAM Journal on Scientific Computing*, 38(5):S485–S507, 2016. 34

[PDF+18] 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. 10, 21, 23, 81, 93

[Pel08] Francois Pellegrini. Scotch and libScotch 5.1 User's Guide. August 2008. User's manual, 127 pages. 11, 70, 72

[PFRR17] Grégoire Pichon, Mathieu Faverge, Pierre Ramet, and Jean Roman. Reordering Strategy for Blocking Optimization in Sparse Linear Solvers. *SIAM Journal on Matrix Analysis and Applications*, 38(1):226 – 248, 2017. 68, 70, 73, 81

[Pic18] G. Pichon. *On the use of low-rank arithmetic to reduce the complexity of parallel sparse linear solvers based on direct factorization technique*. PhD thesis, Universite de Bordeaux, Bordeaux, France, 2018. 10, 21, 67, 73, 77, 80, 84, 87, 93, 104

[QOSB98] Gregorio Quintana-Ortí, Xiaobai Sun, and Christian H. Bischof. A blas-3 version of the qr factorization with column pivoting. *SIAM Journal on Scientific Computing*, 19(5):1486–1494, 1998. 31, 33

[QV99] A. Quarteroni and A. Valli. *Domain Decomposition Methods for Partial Differ-ential Equations*. Oxford University Press, 1999. 106

[RBH12] Sivasankaran Rajamanickam, Erik G. Boman, and Michael A. Heroux. Shylu: A hybrid-hybrid solver for multicore platforms. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 631–643, 2012. 9

[RCL+18] Elizaveta Rebrova, Gustavo Chávez, Yang Liu, Pieter Ghysels, and Xiaoye Sherry Li. A study of clustering techniques and hierarchical

matrix formats for kernel ridge regression. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 883–892, 2018. 72

[RT78] D. J. Rose and R. E. Tarjan. Algorithmic aspects of vertex elimination on directed graphs. *SIAM J. Appl. Math.*, 34(1):176–197, January 1978. 90

[Saa03] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, USA, 2nd edition, 2003. 9, 90, 92

[Sch82] Robert Schreiber. A new implementation of sparse gaussian elimination. *ACM Trans. Math. Softw.*, 8(3):256–276, sep 1982. 11

[TB97] Lloyd Trefethen and David Bau. *Numerical linear algebra*. SIAM, 1997. 30

[TW67] 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, November 1967. 11

[Wat81] J. W. Watts. A conjugate gradient truncated direct method for the iterative solution of the reservoir simulation pressure equation. *Society of Petroleum Engineers Journal*, 21:345–353, 1981. 90

[Wil71] James Hardy Wilkinson. *The algebraic eigenvalue problem*. Springer-Verlag, New York, 1971. 9

[XGL17] Jianwei Xiao, Ming Gu, and Julien 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. IEEE, 2017. 34, 40

[Xia13a] J. L. Xia. Randomized sparse direct solvers. *SIAM Journal on Matrix Analysis and Applications*, 34(1):197–227, 2013. 17, 24

[Xia13b] Jianlin Xia. Efficient structured multifrontal factorization for general large sparse matrices. *SIAM Journal on Scientific Computing*, 35(2):A832–A860, 2013. 21

[YL10] Ichitaro Yamazaki and Xiaoye S. Li. On techniques to improve robustness and scalability of a parallel hybrid linear solver. In *Proceedings of the 9th International Conference on High Performance Computing for Computational Science*, VECPAR'10, page 421–434, Berlin, Heidelberg, 2010. Springer-Verlag. 9

[YLRB17] Chenhan D. Yu, James Levitt, Severin Reiz, and George Biros. Geometry-oblivious fmm for compressing dense spd matrices. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017. 72

Communication with proceedings

[Kor+21a] E. Korkmaz, M. Faverge, G. Pichon, and P. Ramet. "Deciding Non-Compressible Blocks in Sparse Direct Solvers using Incomplete Factorization". *HiPC 2021 - 28th IEEE International Conference on High Performance Computing, Data, and Analytics*. Bangalore, India: IEEE, Dec. 2021, pp. 1–10. DOI: 10.1109/HiPC53243.2021.00024.

Communication without proceedings

[Kor+19a] E. Korkmaz, M. Faverge, G. Pichon, and P. Ramet. "Rank Revealing QR Methods for Sparse Block Low Rank Solvers". *ComPAS'19*. Anglet, France, June 2019.

[Kor+19b] E. Korkmaz, M. Faverge, G. Pichon, and P. Ramet. "Rank Revealing QR Methods for Sparse Block Low Rank Solvers". *Sparse days*. Toulouse, France, July 2019.

[Kor+21b] E. Korkmaz, M. Faverge, G. Pichon, and P. Ramet. "Rank Revealing QR Methods for Sparse Block Low Rank Solvers". *Inria Skoltech Moliere Associated Team meeting*. online event, July 2021.

[Kor+22a] E. Korkmaz, M. Faverge, G. Pichon, and P. Ramet. "Deciding Non-Compressible Blocks in Sparse Direct Solvers using Incomplete Factorization". *Sparse days*. Saint Girons, France, June 2022.

Research report

[Kor+22b] E. Korkmaz, M. Faverge, G. Pichon, and P. Ramet. "Reaching the Quality of SVD for Low-Rank Compression Through QR Variants". 2022.