Utilisation de la compression Block Low-Rank pour accélérer un solveur direct creux supernodal

Grégoire Pichon

Compas'2017 : Parallélisme / Architecture / Système
Sophia Antipolis, France, du 27 au 30 juin 2017

Utilisation de la compression Block Low-Rank pour accélérer un solveur direct creux supernodal

Grégoire Pichon

Inria, University of Bordeaux, CNRS (Labri UMR 5800), Bordeaux INP, Talence, France

---

Résumé
La résolution de systèmes linéaires creux est une opération de base dans la modélisation de nombreux problèmes physiques, comme l'électromagnétique ou l'astrophysique. Ce papier se focalise sur le solveur supernodal PaStiX et propose l'introduction de techniques de rang faible pour réduire la complexité en temps et en mémoire du solveur. Plus précisément, le schéma de compression Block Low-Rank (BLR) est utilisé pour exploiter le rang faible des blocs apparaissant dans la résolution directe de systèmes creux. Une première approche, appelée *Minimal Memory*, permet de réduire l'empreinte mémoire du solveur jusqu'à un facteur 4.4 sur une architecture multi-threadée, composée de 24 threads et de 128 Go de mémoire. Une seconde stratégie, appelée *Just-In-Time*, permet de réduire le temps de résolution jusqu'à un facteur 3.3. Ces deux approches, couplées avec deux techniques classiques de compression – décomposition en valeurs singulières (SVD) et l'algorithme Rank-Revealing QR (RRQR) – sont comparées en termes de temps, consommation mémoire et stabilité numérique.

Mots-clés : Solveurs creux directs, Block Low-Rank, PaStiX, architectures multi-threadées.

---

1. Introduction

De nombreuses applications scientifiques utilisent des modèles qui reviennent à résoudre des systèmes linéaires de la forme $Ax = b$, où la matrice A est grande et creuse. Pour résoudre ces problèmes, une approche classique est d'utiliser un solveur direct qui factorise la matrice en un produit de matrices triangulaires avant de résoudre des systèmes triangulaires.
Cependant, cette approche est limitée par le coût de la factorisation, que cela soit au niveau de la consommation mémoire ou du temps de résolution. Si d'autres approches, comme les méthodes itératives, permettent de résoudre rapidement des systèmes associés à une équation bien spécifique, il existe actuellement peu de solutions pour résoudre de grands systèmes dans une approche complètement algébrique. De récents travaux ont tiré profit de représentations de rang faible des blocs qui apparaissent durant la factorisation, en les compressant sous divers formats, comme Block Low-Rank (BLR), $\mathcal{H}$, $\mathcal{H}^2$, HSS, HODLR...
L'objectif principal de ce travail est l'introduction de la technique BLR dans le solveur PaStiX [14], en ne tirant pas profit des propriétés du problème (géométrie...) pour conserver un solveur algébrique. Deux stratégies ont été développées. La première, *Minimal Memory*, compresse l'ensemble de la matrice creuse avant toute opération numérique. Elle implique des opérations complexes entre structures de rang faible mais permet de réduire au maximum la consommation mémoire, permettant de résoudre des systèmes plus gros. A l'inverse, la stratégie *Just-In-Time* se focalise sur la réduction du temps de factorisation en compressant au plus

tard les blocs. Le nouveau solveur prend alors en entrée une tolérance requise par l'utilisateur, et peut être utilisé en tant que solveur direct à cette précision ou en tant que préconditionneur pour atteindre la précision machine.
La contribution de ce travail est la mise en place d'un tel solveur pour une approche supernodale tout en conservant le caractère algébrique. Les algorithmes développés suivent le même niveau de parallélisme que la version originale de PaStiX.
Dans la Section 2, le fonctionnement basique des solveurs directs creux supernodaux est décrit. La Section 3 présente l'état de l'art, avant d'entrer dans les détails des algorithmes proposés dans la Section 4. La Section 5 présente les résultats expérimentaux, avant de discuter de futurs axes de recherche dans la Section 6.

2. Solveurs directs creux supernodaux

L'approche classique des solveurs directs se décompose en quatre étapes : 1) numérotation des inconnues, 2) factorisation symbolique, 3) factorisation numérique et 4) résolution de systèmes triangulaires. A noter que dans le reste du document, nous ne considérerons que des systèmes creux à structure symétrique.
L'objectif de la première étape est de minimiser le remplissage – les éléments nuls devenant non nuls – qui apparaît durant la factorisation numérique, afin de réduire consommation mémoire et coût calculatoire associés à la résolution du système. Afin de réduire le remplissage tout en garantissant un bon niveau de parallélisme, la dissection emboîtée [8] est majoritairement utilisée à travers des bibliothèques de partitionnement comme Metis [16] ou Scotch [20]. Chaque ensemble de sommets correspondant à un séparateur est appelé un supernoeud.
A partir de ce partitionnement des inconnues, la seconde étape prédit la forme de la matrice factorisée et l'arbre d'élimination des inconnues. Cette structure est composée d'un bloc diagonal pour chacun des supernoeuds et de blocs extra-diagonaux qui représentent les interactions entre les supernoeuds.
La troisième étape tire profit de la factorisation symbolique pour réaliser une factorisation en place. L'élimination d'un supernoeud est découpée en trois étapes : 1) *Factorization* du bloc diagonal dense, 2) *Solve* sur les blocs extra-diagonaux du supernoeud et 3) application d'un *Update* sur une partie des supernoeuds non éliminés.

3. État de l'art

Hackbusch [12] a introduit la factorisation $\mathcal{H}$-LU pour les matrices denses, qui a par la suite été étendue aux matrices creuses, en utilisant la dissection emboîtée pour tirer profit du creux. Cependant, aucune factorisation symbolique n'est exploitée : comme l'a montré [10], qui compare l'approche aux solveurs creux, la consommation mémoire est parfois plus importante. Dans [11], cette approche est utilisée de manière algébrique, et [17, 19] proposent une implémentation au dessus d'un support d'exécution.
La compression Hierarchically Off-Diagonal Low-Rank (HODLR) a été utilisée dans un solveur multifrontal [3] pour accélérer l'élimination des gros fronts. Elle a été totalement étendue au creux dans [4] en utilisant des techniques de présélection de lignes et de colonnes pour permettre un procédé d'*Update* efficace. Un solveur supernodal utilisant un format similaire a été présenté dans [6], mais requiert une estimation des rangs et n'est donc pas totalement algébrique.
Le format Hierarchically Semi-Separable (HSS) a été utilisé dans [24] pour des problèmes 2D et en tenant compte de la géométrie. [23] a proposé un solveur géométrique plus général pour

accélérer la factorisation mais sans gain mémoire. [9] a mis en place un solveur algébrique utilisant la technique de *randomized sampling* pour accélérer l'*Update*. L'arithmétique $\mathcal{H}^2$ [13] a été utilisée dans plusieurs solveurs [15, 21, 22, 25].
Un solveur dense utilisant la compression BLR a été présenté dans [5]. L'approche est proche de notre stratégie *Minimal Memory* mais pour des matrices denses, où la mise à jour *low-rank* concerne des matrices de taille similaire et peut être réalisée efficacement.
Finalement, un solveur creux utilisant BLR a été développé pour le solveur MUMPS, avec une approche multifrontale [1]. Une étude théorique de la complexité a également été réalisée [2]. Dans ces travaux, les blocs de contribution ne sont pas compressés, comme dans notre stratégie *Just-In-Time*. Par ailleurs, les opérations *low-rank* concernent les fronts qui sont des matrices denses, contrairement à notre approche supernodale où les contributions sont directement appliquées.

4. Solveur Block Low-Rank

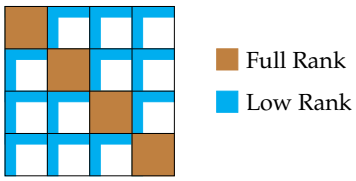

FIGURE 1 – Compression Block Low-Rank.

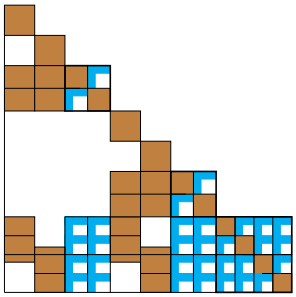

FIGURE 2 – Factorisation symbolique.

Le format BLR est un format de compression non hiérarchique, à l'inverse des autres formats présentés dans la Section 3. Si on considère une matrice dense, comme dans la Figure 1, le format BLR découpe (*clustering*) la matrice en un ensemble de blocs plus petits. Les blocs diagonaux sont conservés denses et les blocs extra-diagonaux sont compressés sont une forme *low-rank* uv^t, obtenue avec des techniques de compression comme SVD ou RRQR.
Notre approche consiste à appliquer ce format à la structure symbolique présentée en Section 2. Les supernoeuds suffisamment larges sont découpés en un ensemble de supernoeuds plus petits. A partir de cette factorisation symbolique raffinée, les blocs extra-diagonaux sont compressés sous un format *low-rank*, comme présenté dans la Figure 2.
Adapter le solveur original à cette nouvelle structure revient à remplacer les noyaux opérant sur des blocs denses par des noyaux opérant sur des blocs au format *low-rank*. Cependant, différentes variantes de l'algorithme peuvent être obtenues en changeant le moment où les blocs sont compressés.
La Figure 3 présente le graphe de tâches associé à la factorisation classique (*full-rank*) d'une matrice de 3×3 blocs. Cela correspond aux opérations suivantes :

1. *Factorization* du premier bloc diagonal (`GETRF`)
2. *Solve* sur les quatre blocs extra-diagonaux du premier supernoeud (`TRSM`)
3. *Update* de la matrice sous-jacente de taille 2×2 blocs (`LR2GE`)

4. *Factorization* du second bloc diagonal
5. *Solve* sur les deux blocs extra-diagonaux du second supernoeud
6. *Update* de la matrice sous-jacente de taille 1×1 bloc
7. *Factorization* du troisième bloc diagonal

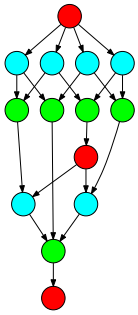

FIGURE 3 – Full-rank.

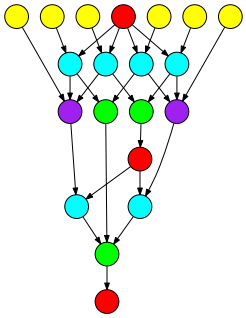

FIGURE 4 – *Minimal Memory*.

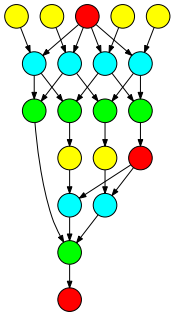

FIGURE 5 – *Just-In-Time*.

La première stratégie, *Minimal Memory*, présentée dans la Figure 4, consiste à compresser la matrice creuse A et réalise la factorisation avec des blocs *low-rank*. Chaque opération sur un bloc *low-rank* est remplacée par un noyau opérant sur la forme low-rank. Le coût de l'opération `TRSM` est réduite, alors que l'opération *LR2LR* permet de faire l'addition entre deux matrices *low-rank*. Avec cette stratégie, des gains mémoire sont possibles car les blocs de la structure symbolique ne sont jamais alloués sous leur forme dense. Cependant, l'opération *LR2LR* entraîne un surcoût calculatoire théorique confirmé par les expériences en Section 5.
La seconde stratégie, *Just-In-Time* présentée dans la Figure 5, compresse les blocs au plus tard pour éviter le surcoût de l'addition *low-rank*. Ainsi, lorsqu'un bloc à reçu toutes ses mises à jour (*Update*), *i.e.* avant que l'opération `TRSM` ne soit appliquée à ce bloc, il est compressé. De cette manière l'opération d'*Update* de matrices *low-rank* vers des matrices denses est accélérée. Cependant, en compressant les blocs au plus tard, le gain mémoire sera plus faible qu'avec la stratégie précédente (voire nul si tous les blocs de la matrice ont été initialement alloués de manière dense).

5. Expériences

Les expériences ont été réalisées sur la machine *Plafrim* [1], en utilisant un noeud `miriel`, composé de deux INTEL `Xeon E5-2680 v3` 12-coeurs à 2.50 GHz et de 128 Go de mémoire. La bibliothèque INTEL MKL `2016` a été utilisée pour les BLAS et la SVD. Le noyau RRQR est issu du solveur BLR-MUMPS [1], et étend les sous-routines de Rank-Revealing QR bloquées de LAPACK 3.6.0 (xGEQP3). La version de PASTIX utilisée dans nos expériences est disponible sous un dépôt git public [2], et l'implémentation originale multi-threadée utilise l'ordonnancement statique présenté dans [18].

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

Le partitionnement a été réalisé avec SCOTCH 5.1.11 [20], en définissant la taille minimale des blocs à 15 (paramètre `cmin`), et en autorisant l'agrégation de colonnes tant que le remplissage supplémentaire n'excède pas 8% (paramètre `frat`). Dans nos expériences, les blocs de taille supérieure à 256 sont découpés en un ensemble de sous-blocs de taille minimale 128. De cette nouvelle partition, les blocs de largeur supérieure ou égale à 128 et de hauteur supérieure ou égale à 20 sont mis sont forme *low-rank*.
Les expériences ont été réalisées sur un ensemble de matrices 3D extraites de The SuiteSparse Matrix Collection [7] :

— *Atmosmodj* : atmospheric model (1 270 432 dofs)

— *Audi* : structural problem (943 695 dofs)

— *Hook* : model of a steel hook (1 498 023 dofs)

— *Serena* : gas reservoir simulation (1 391 349 dofs)

— *Geo1438* : geomechanical model of earth (1 437 960 dofs)

Et la matrice `lap120` correspond à un Laplacien (stencil de 7 points) de taille 120^3.
Les précisions présentées correspondent à l'erreur inverse sur b : $\frac{||Ax-b||_2}{||b||_2}$.
L'ensemble des expériences a été réalisé en utilisant les 24 threads disponibles. La version SVD est une référence en termes de qualité (donc pour la consommation mémoire), mais n'est pas présentée au niveau des performances. Trois tolérances d'entrée sont étudiées pour chaque expérience, afin de couvrir un large spectre. Une étape de raffinement itératif est ajoutée.

5.1. Performance

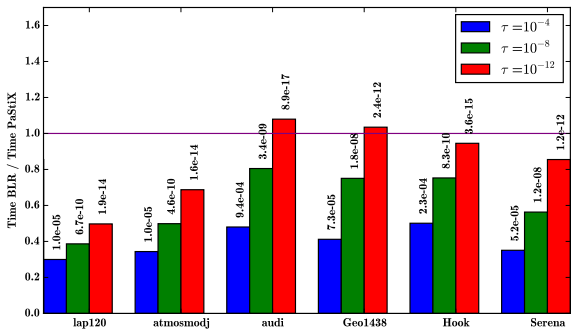

FIGURE 6 – *Just-In-Time* avec RRQR

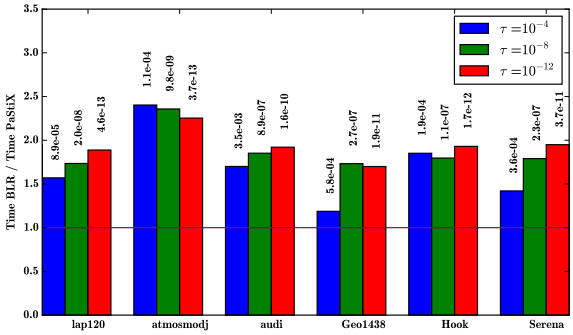

FIGURE 7 – *Minimal Memory* avec RRQR

La Figure 6 démontre que la stratégie *Just-In-Time* couplée à la compression RRQR permet de réduire le temps de factorisation dans quasiment tous les cas de tolérance. Des applications qui requièrent une faible précision peuvent donc bénéficier d'une accélération jusqu'à un facteur 3.3. A l'inverse, la Figure 7 montre qu'il est plus difficile pour la stratégie *Minimal Memory* d'être compétitive avec un solveur direct sans compression. Le surcoût est en moyenne d'un facteur 1.8, mais la tolérance a moins d'effet sur le temps de factorisation.
Pour les deux stratégies, la solution en sortie est acceptable vis-à-vis de la précision demandée en entrée. On peut noter qu'elle est un peu moins précise pour la stratégie *Minimal Memory*, car plus d'approximations sont réalisées lors la recompression *low-rank*. Ces résultats démontrent

que le solveur développé permet de sauvegarder algébriquement les informations les plus importantes.
A noter que le temps de résolution des systèmes triangulaires est également réduit.

5.2. Consommation mémoire

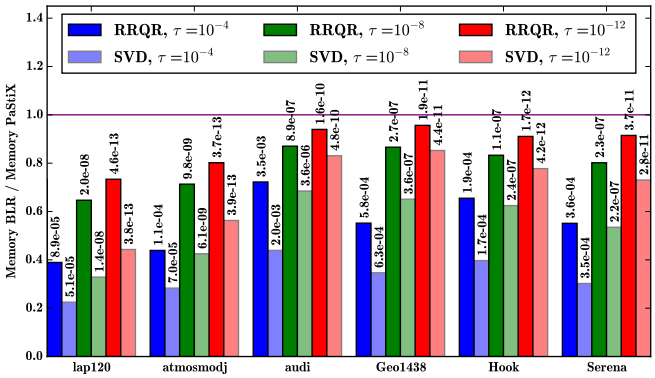

FIGURE 8 – Taille des facteurs avec la stratégies *Minimal Memory*.

Si l'étude de performance a démontré que la stratégie *Minimal Memory* était moins performante que le solveur original, elle permet des gains pratiques sur la consommation mémoire et le pic mémoire du solveur. Nous l'expérimentons en décrivant le gain mémoire sur des problèmes qui passent déjà en mémoire, puis sur une résolution de Laplaciens de taille variable. La Figure 8 présente les gains mémoires sur le stockage des facteurs (très proche du pic mémoire pour une approche supernodale) par rapport à la version *full-rank* de PASTIX. Dans ce cas, nous comparons également les deux techniques de compression, la SVD permettant d'atteindre, à une précision donnée, le meilleur taux de compression. On peut observer que l'utilisation de la SVD permet des gains plus intéressants, ainsi qu'une meilleure qualité en sortie. Plus la précision demandée est faible (10^{-12}), plus la consommation mémoire est importante. Avec une précision moins stricte, les gains mémoire peuvent dépasser 50%.
Une autre étude de scalabilité sur des Laplaciens à permis de montrer que le solveur original pouvait passer en mémoire pour résoudre au maximum un Laplacien de taille 160^3. Pour une précision de 10^{-8}, un Laplacien de taille 190^3 a pu être résolu, alors que pour une précision de 10^{-4}, la résolution d'un Laplacien de taille 230^3 a pu être réalisée sous la contrainte mémoire de 128 Go.

6. Conclusion et travaux futurs

Nous avons introduit la technique de compression BLR dans le solveur supernodal PASTIX afin de tirer profit des blocs de rang faible apparaissant durant la factorisation d'une matrice creuse. Une première stratégie a permis de réduire l'empreinte mémoire du solveur et une seconde stratégie entraîne une réduction du temps de résolution.
Dans des travaux futurs, nous étudierons des techniques de renumérotation en jouant sur la dissection emboîtée pour maximiser la compressibilité des blocs. L'utilisation de la technique de compression HODLR sera également étudiée pour permettre des gains plus conséquents.

L'étape d'*Update* sera plus particulièrement étudiée, afin de permettre simultanément d'obtenir des gains en temps et en mémoire. Une autre piste de recherche pour mettre en place une stratégie hybride tirant profit des deux approches, pourrait consister à utiliser un support d'exécution en introduisant une contrainte mémoire. Ainsi, un nombre réduit de blocs seraient compressés avec la stratégie *Minimal Memory* afin de passer en mémoire, tout en maximisant le nombre de blocs compressés avec la stratégie *Just-In-Time* pour gagner en efficacité.

Remerciements

Ces travaux sont financés par la DGA à travers une bourse de thèse DGA/Inria. Les expériences présentées dans ce papier ont été réalisées sur la plateforme expérimentale PLAFRIM.

Bibliographie

1. Amestoy (P.), Ashcraft (C.), Boiteau (O.), Buttari (A.), L'Excellent (J.-Y.) et Weisbecker (C.). – Improving Multifrontal Methods by Means of Block Low-Rank Representations. *SIAM Journal on Scientific Computing*, vol. 37, n3, 2015, pp. A1451–A1474.
2. Amestoy (P.), Buttari (A.), L'Excellent (J.-Y.) et Mary (T.). – *On the Complexity of the Block Low-Rank Multifrontal Factorization*. – Research Report nRT–2016–03–FR, IRIT, mai 2016.
3. Aminfar (A.), Ambikasaran (S.) et Darve (E.). – A fast block low-rank dense solver with applications to finite-element matrices. *Journal of Computational Physics*, vol. 304, 2016, pp. 170–188.
4. Aminfar (A.) et Darve (E.). – 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*, 2016.
5. Anton (J.), Ashcraft (C.) et Weisbecker (C.). – A Block Low-Rank Multithreaded Factorization for Dense BEM Operators. – In *SIAM Conference on Parallel Processing for Scientific Computing (SIAM PP 2016)*, Paris, France, avril 2016.
6. Chadwick (J. N.) et Bindel (D. S.). – An Efficient Solver for Sparse Linear Systems Based on Rank-Structured Cholesky Factorization. *CoRR*, vol. abs/1507.05593, 2015.
7. Davis (T. A.) et Hu (Y.). – The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, vol. 38, n1, décembre 2011, pp. 1 :1–1 :25.
8. George (A.). – Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, vol. 10, n2, 1973, pp. 345–363.
9. Ghysels (P.), Li (X. S.), Rouet (F.-H.), Williams (S.) et Napov (A.). – An Efficient Multicore Implementation of a Novel HSS-Structured Multifrontal Solver Using Randomized Sampling. *SIAM Journal on Scientific Computing*, vol. 38, n5, 2016, pp. S358–S384.
10. Grasedyck (L.), Hackbusch (W.) et Kriemann (R.). – Performance of H-LU preconditioning for sparse matrices. *Computational methods in applied mathematics*, vol. 8, n4, 2008, pp. 336–349.
11. Grasedyck (L.), Kriemann (R.) et Le Borne (S.). – Parallel black box $\mathcal{H}$-LU preconditioning for elliptic boundary value problems. *Computing and Visualization in Science*, vol. 11, n4-6, 2008, pp. 273–291.
12. Hackbusch (W.). – *Hierarchical Matrices : Algorithms and Analysis*. – Springer Series in Computational Mathematics, 2015.
13. Hackbusch (W.) et Börm (S.). – Data-sparse Approximation by Adaptive H2-Matrices. *Computing*, vol. 69, n1, 2002, pp. 1–35.
14. Hénon (P.), Ramet (P.) et Roman (J.). – PaStiX : A High-Performance Parallel Direct Solver

for Sparse Symmetric Definite Systems. *Parallel Computing*, vol. 28, n2, janvier 2002, pp. 301–321.

15. Ho (K. L.) et Ying (L.). – Hierarchical interpolative factorization for elliptic operators : differential equations. *Communications on Pure and Applied Mathematics*, 2015.
16. Karypis (G.) et Kumar (V.). – METIS : A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices, 1995.
17. Kriemann (R.). – H-LU factorization on many-core systems. *Computing and Visualization in Science*, vol. 16, n3, 2013, pp. 105–117.
18. Lacoste (X.). – *Scheduling and memory optimizations for sparse direct solver on multi-core/multi-gpu cluster systems*. – Talence, France, Thèse de PhD, Bordeaux University, février 2015.
19. Lizé (B.). – *Résolution directe rapide pour les éléments finis de frontière en électromagnétisme et acoustique : h-matrices. parallélisme et applications industrielles.* – Thèse de PhD, École Doctorale Galilée, juin 2014.
20. Pellegrini (F.). – Scotch and libScotch 5.1 User's Guide, août 2008. User's manual, 127 pages.
21. Pouransari (H.), Coulier (P.) et Darve (E.). – Fast hierarchical solvers for sparse matrices using extended sparsification and low-rank approximation. *arXiv preprint arXiv :1510.07363v3*, 2015.
22. Sushnikova (D. A.) et Oseledets (I. V.). – "Compress and eliminate" solver for symmetric positive definite sparse matrices. *arXiv preprint arXiv :1603.09133v3*, 2016.
23. Wang (S.), Li (X. S.), Rouet (F.-H.), Xia (J.) et De Hoop (M. V.). – A Parallel Geometric Multifrontal Solver Using Hierarchically Semiseparable Structure. *ACM Trans. Math. Softw.*, vol. 42, n3, mai 2016, pp. 21 :1–21 :21.
24. Xia (J.), Chandrasekaran (S.), Gu (M.) et Li (X.). – Superfast Multifrontal Method For Large Structured Linear Systems of Equations. *Siam Journal on Matrix Analysis and Applications*, vol. 31, 2009, p. 1382–1411.
25. Yang (K.), Pouransari (H.) et Darve (E.). – Sparse hierarchical solvers with guaranteed convergence. *arXiv preprint arXiv :1611.03189*, 2016.