Solveurs rapides pour l'aeroacoustique haute frequence

Marek Felšöci

université de BORDEAUX

Inria

THÈSE PRÉSENTÉE

POUR OBTENIR LE GRADE DE

DOCTEUR
DE L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE

MATHÉMATIQUES ET INFORMATIQUE

INFORMATIQUE

Par **Marek FELŠÖCI**

Solveurs rapides pour l'aéroacoustique haute-fréquence

Sous la direction de : **Luc GIRAUD**
Co-directeur : **Guillaume SYLVAND**
Co-encadrant : **Emmanuel AGULLO**

Soutenue le 22 février 2023

Président du jury : M. Christian PEREZ

Membres du jury :

M. Emmanuel AGULLO	Chargé de recherche	Inria	Invité
M. Alfredo BUTTARI	Directeur de recherche	CNRS	Invité
Mme. Stéphanie CHAILLAT	Chargée de recherche	CNRS	Examinatrice
M. Jean-Yves L'EXCELLENT	Ingénieur de recherche	Mumps Technologies, ENS Lyon	Rapporteur
M. Luc GIRAUD	Directeur de recherche	Inria	Directeur
M. David GOUDIN	Ingénieur	Atos	Examinateur
M. Konrad HINSEN	Directeur de recherche	CNRS	Examinateur
M. Christian PEREZ	Directeur de recherche	Inria	Examinateur
M. Ulrich RÜDE	Professeur	Friedrich Alexander Universität	Rapporteur
M. Guillaume SYLVAND	Ingénieur de recherche	Airbus Central R&T	Co-directeur

Solveurs rapides pour l'aéroacoustique haute-fréquence

Résumé : Dans l'industrie aéronautique, l'aéroacoustique est utilisée pour modéliser la propagation d'ondes sonores dans les flux d'air enveloppant un avion en vol. Il est alors possible de simuler le bruit produit par un avion au niveau du sol lors du décollage et de l'atterisage afin d'assurer le respect des normes environnementales et de permettre la conception de futurs modèles d'avion. Contrairement à la plupart des autres simulations complexes en physique, la méthode consiste en la résolution de systèmes linéaires couplés creux/denses. Pour produire un résultat réaliste, le nombre d'inconnues dans le système peut être extrêment important ce qui fait de sa résolution un défi de taille.

Dans ce travail, nous nous focalisons sur la conception et l'évaluation d'algorithmes pour résoudre de grands systèmes linéaires de ce genre. D'un côté, nous proposons des algorithmes utilisant l'interface de programmation (API) existante de solveurs directs creux et denses riches en fonctionnalités et optimisés. Grâce à ces algorithmes, nous arrivons à contourner les défauts majeurs d'un usage basique de ces solveurs et profiter pleinement de leurs fonctionnalités avancées telles que la compression numérique, le calcul *out-of-core* et le parallélisme en mémoire distribuée. D'un autre côté, nous évaluons une API de solveur alternative qui s'appuie sur un couplage de solveurs directs à base de tâches utilisant le même moteur d'exécution. Une API personnalisée permet d'améliorer la composabilité et de simplifier l'échange de données entre les solveurs pour une utilisation plus efficace de resources de calcul. Tandis que l'introduction de ces changements substantiels dans des solveurs aux fonctionnalités avancés et maintenus par la communauté ne peut se faire qu'à long terme à cause de la complexité de leur code source (quelques centaines de milliers de ligne de code), nous avons pu implémenter une preuve de concept de cette approche dans un prototype réduit.

Outre la contribution principale, nous avons consacré un important effort à la reproductibilité de notre travail. À cette fin, nous avons exploré les principes de la programmation lettrée ainsi que les outils logiciels associés pour garantir la reproductibilité des environnements expérimentaux et des expériences numériques elles-mêmes sur différentes machines et sur des périodes de temps étendues.

Mots-clés : matrices creuses et denses, $\mathcal{H}$-matrices, grands systèmes linéaires, méthode directe, solveurs parallèles, compression de rang faible, out-of-core, méthode des éléments finis (FEM), méthode des éléments finis de frontière (BEM), couplage FEM/BEM, reproductibilité, programmation lettrée

---

Fast solvers for high-frequency aeroacoustics

Abstract: In the aeronautical industry, aeroacoustics is used to model the propagation of acoustic waves in air flows enveloping an aircraft in flight. One can thus simulate the noise produced at ground level by an aircraft during the takeoff and landing phases, in order to validate that the regulatory environmental standards are met and support the design of future planes. Unlike most other complex physics simulations, the method resorts to solving coupled sparse/dense systems. In order to produce a result that is physically realistic, the number of unknowns in the system to solve can be extremely important, which makes its treatment a computational challenge.

In this work, we focus on the design and evaluation of algorithms for solving large linear systems of this kind. On the one hand, we propose algorithms using the existing Application Program Interface (API) of fully-featured and well-optimized sparse and dense direct solvers. Thanks to these algorithms, we can get around major shortcomings of a straightforward usage the solvers and fully benefit from their advanced features such as numerical compression, out-of-core computation and distributed memory parallelism. On the other hand, we investigate an alternative solver API relying on the coupling of task-based direct solvers built on top of the same runtime. A custom API allows for a better composability and an easier data exchange between the solvers leading to a more efficient usage of computing resources. While introducing such substantial changes into fully-featured community-driven solvers can only be a long-time work due to their complex codebase (several hundred thousands lines of code), we were able to implement a proof-of-concept of this approach within a non fully-featured prototype.

Beyond the main contribution, we put a strong emphasis on the reproducibility of our work. To this end, we explore the principles of literate programming and software tools for ensuring the reproducibility of both experimental environments and numerical experiments themselves across different machines and extended periods of time.

Keywords: sparse and dense matrices, $\mathcal{H}$-matrices, large linear systems, direct method, parallel solvers, low-rank compression, out-of-core, Finite Elements Method (FEM), Boundary Elements Method (BEM), FEM/BEM coupling, reproducibility, literate programming

Unité de recherche
Inria Bordeaux Sud-Ouest, 200 Avenue de la Vieille tour, 33405 Talence, France.

Acknowledgements

In the first place, I would like to thank my advisors Emmanuel Agullo, Luc Giraud et Guillaume Sylvand for these more than three years of fruitful collaboration that led to the accomplishment of this thesis, allowed me to learn a lot of new exicitng things and granted me a precious experience for my future life. I truly enjoyed our collaboration and I hope that our paths will cross again soon.

I thank also Olivier Beaumont and Emmanuelle Saillard, who were in my monitoring committee, for their constructing remarks during the committee meetings and especially for their support and advice regarding my future career choices.

I would then like to address my acknowledgements to Jean-Yves L'Excellent and Ulrich Rüde for dedicating their time and expertise to the review of this manuscript and for their constructive feedback on the latter. Many thanks again to Jean-Yves L'Excellent as well as to Patrick Amestoy for taking time to discuss in depth all the remarks and ideas that emerged since the submission of the manuscript. I would also like to thank all the other members of the jury, Stéphanie Chaillat, David Goudin, Konrad Hinsen and Christian Pérez. I thank Alfredo Buttari for accepting the invitation too. Our collaboration during the thesis represent a very pleasant and enriching experience for me.

I am very grateful to Jérôme Robert from Airbus for our numerous exchanges as well as for his committement, his time and his expertise that made it possible to implement the algorithms developed within this thesis. I would also like to thank Jean-Marie Couteyen for his help with the Airbus benchmarking software that represents one of the key building blocks of this work.

I thank Hervé Mathieu for his involvement and assistance in the deployment of the energy profiling `energy_scope` tool as well as Amina Guermouche and Bastien Tagliaro for all of the help and expertise dedicated to our common effort of studying the energy consumption of the methods proposed in this thesis.

Many thanks from me go also to the SED and the PlaFRIM teams, especially to François Rué and Florent Pruvost. Thank you for your valuable technical help, assistance, insights and enjoyable, often cheerful, discussions and coffee breaks.

I would like to thank again Emmanuel Agullo whose 'kindly insisting encouragements' at the beginning of the thesis led me to discover a whole new domain of reproducibility, open-source software and related considerations. It was a more than enriching experience. I had the occasion to learn a lot of new things and tools, to change my mind on some crucial questions but also to know a whole new community of wonderful people. If I would have to cite one person among them, it would certainly be Ludovic Courtès. All of the efforts towards improving the reproducibility of our work would not be possible without his great expertise and patience. Thank you, Ludo', for all of our exchanges and insighful discussions.

I am also very greatful to Karim for his support and our interesting discussions which helped me to go through the toughest last months of the thesis when my working hours were often

aligned with those of evening security officers.

I reserve a huge 'thank you' to my fellow colleagues from Inria and LaBRI for these great past three years and all the cheerful moments we have spent together during the coffee breaks, beer and wine 'tastings', apéroplages as well as many others. Thank you all! Thank you for your friendship and your support!

Nakoniec prejdem do svôjho rodného jazyka, tej ľubozvučnej slovenčiny. Veľká vďaka patrí aj celej mojej rodine za ich nikdy neutíchajúcu podporu v tomto dobrodružstve a v podstate vo všetkom, čo som si zatiaľ v živote zaumienil. Vzhľadom na nedávnu radostnú rodinnú udalosť by som rád túto prácu venoval novopečenému vnúčikovi Matúšovi!

Contents

Résumé en français

Nous nous intéressons à la résolution de très grands systèmes d'équations linéaires $Ax = b$ avec la particularité d'être composés à la fois d'une partie creuse et d'une partie dense. Dans la partie creuse, la plupart des coefficients sont nuls, ce qui peut être exploité pour optimiser le calcul. Dans la partie dense, la proportion de coefficients nuls n'est pas suffisamment importante pour en tirer partie. Les systèmes de ce genres apparaissent dans différents domaines scientifiques tels que l'électromagnétisme, l'acoustique, l'étude des interactions entre le sol et les structures ou la prédiction génomique. Dans le cas présent, les systèmes linéaires émergent des simulations numériques dans un contexte industriel où l'on couple deux types de méthodes des éléments finis, c'est-à-dire la Méthode des éléments finis (FEM) et la Méthode des élements finis de frontière (BEM). Il en résulte un système linéaire couplé creux/dense FEM/BEM avec une matrice de coefficients A ayant un partitionnement distinctif 2×2 où le bloc $(1, 1)$ est une large sous-matrice creuse associée à la FEM, $(2, 1)$ une plus petite sous-matrice creuse représentant le couplage FEM/BEM et $(2, 2)$ une plus petite sous-matrice dense associée à la BEM. Dans ce travail, nous considérons la plupart du temps des systèmes couplés où A est symétrique. Néanmoins, nous considérons également un cas industriel spécifique où A est non-symétrique. Afin de produire un résultat physiquement réaliste, le nombre d'inconnues dans le système à résoudre peut être extrêment important (des millions d'inconnues BEM, des centaines de millions d'inconnues FEM) ce qui fait de son traitement un vrai défi.

La motivation initiale pour cette thèse réside dans le domaine de l'aéroacoustique, c'est-à-dire l'étude du couplage entre les phénomènes acoustiques et la mécanique des fluides. Dans l'industrie aéronautique, cette discipline est utilisée pour modéliser la propagation d'ondes acoustiques dans les flux d'air enveloppant un avion en vol. En particulier, ceci permet de simuler le bruit produit par un avion au niveau du sol pendant les phases de décollage et d'atterissage. Au-delà de son importance technique pour attester du respect des normes environementales en vigueur, c'est aussi un défi sociétal et économique. Les fabricants sont incités à minimiser le niveau de bruit de leurs avions afin de réduire son impact sur la santé. Lors de la conception de futurs avions destinés à desservir les aéroports de banlieu, ils ont donc souvent recours aux simulations numériques de ce genre de phénomènes. Les modèles physiques sous-jacents sont exprimés sous forme d'équations aux dérivés partielles. Par conséquent, la conception d'un modèle numérique nécessite une approximation du modèle physique d'origine sur un domaine d'intérêt et en utilisant une technique de discrétisation adaptée. Les ondes aéroacoustiques sont susceptibles d'interférer avec différents types de supports. Dans le flux de jet émis par les réacteurs, le support de propagation, c'est-à-dire l'air, est très hétérogène en termes de température, densité, etc. Ailleurs, nous considérons le support comme étant homogène. Pour calculer la propagation des ondes à travers le support hétérogène, c'est-à-dire le flux de jet des réacteurs, nous utilisons la FEM conduisant à un système linéaire creux. Pour le support homogène, c'est-à-dire à la frontière de l'avion et du flux de jet, nous nous appuyons sur la BEM conduisant à un système dense. Le couplage de ces méthodes donne donc un système linéaire creux/dense FEM/BEM que nous cherchons à résoudre (chapitre 1).

Les deux classes de méthodes que nous pouvons utiliser pour résoudre un système linéaire

sont les méthodes itératives et directes. D'un côté, le but des méthodes itératives est de trouver une approximation de la solution en commençant par une hypothèse initiale et en avançant de façon itérative. Si le calcul converge suffisamment près de la solution exacte en un nombre raisonnable d'itérations, une méthode itérative peut garder son avantage en performance vis-à-vis une méthode directe. De l'autr côté, les méthodes directes sont souvent basées sur une étape initiale de factorisation de A en un produit de matrices rendant la résolution du système plus facile. Dans le cas d'un système symétrique, la matrice A peut être factorisée sous forme $A = LL^T$ où L est une matrice triangulaire inférieure et T marque l'opérateur de transposition. Dans la cas d'une matrice non-symétrique, A peut être factorisée sous forme $A = LU$ où L et U sont des matrices triangulaires inférieure et supérieure, respectivement. La faible proportion de coefficients non-nuls dans une matrice creuse motive l'optimisation de son empreinte mémoire en évitant le stockage de zéros. Le caractère creux permet également de réduire la compléxité algorithmique des opérations matricielles associées. Néanmoins, la factorisation d'une matrice creuse est susceptible d'introduire de nouveaux coefficients non-nuls dans la matrice d'origine. Ce phénomène s'appelle le *remplissage* [66]. Si le remplissage est important, les méthodes directes peuvent conduire à une consommation de mémoire très élevée. Toutefois, lorsqu'elles rentrent dans la mémoire, elles sont extrêment robustes du point de vue numérique et indispensables dans le cadre d'un contexte industriel où elles sont régulièrement mises en œuvre pour résoudre des systèmes allant des tailles modérées à des tailles relativement grandes.

Dans cette thèse, nous nous focalisons sur les méthodes directes et étudions les possibilités de conception de schémas efficaces pour la résolution de systèmes linéaire couplés creux/denses. Les solveurs directs creux et denses de l'état de l'art fournissent des briques de construction que nous pouvons utiliser. La plupart des solveurs proposent des briques de construction basiques telles que la factorisation creuse, la factorisation dense et les opérations de résolution associées. Nous pouvons les appliquer directement sur les trois sous-matrices (quatre dans un cas non-symétrique) de A et composer un solveur direct creux/dense. Certains solveurs bien équipés nous permettent d'exprimer qu'une partie des inconnues dans le système linéaire est liée à la sous-matrice dense. Dans ce cas, le solveur creux a la connaissance de la matrice A dans son intégralité et peut effectuer des opérations creuses de manière plus efficace. Indépendemment du choix des briques de construction, nous appelons ces couplages simples de solveurs directs creux et denses de l'état de l'art les couplages de base. Avec la taille grandissante du système linéaire couplé et plus particulièrement de sa partie dense, les couplages de bases peuvent très vite venir à manquer de mémoire. Pour faire face à ce problème, nous pouvons nous appuyer sur certaines fonctionnalités avancées souvent implémentées dans les solveurs directs bien équipés. Par exemple, dans le cadre d'un seul ordinateur à mémoire partagée, la compression numérique peut être utilisée pour réduire l'empreinte mémoire ainsi que le temps du calcul. Puis, grâce aux techniques *out-of-core* (déplacement de la mémoire vive vers le disque des données qui ne sont pas utilisées dans le calcul actuel), nous pouvons d'avantage réduire la consommation mémoire. Finalement, le calcul parallèle sur plusieurs ordinateurs à la fois peut nous permettre de résoudre des systèmes encore plus grands en distribuant la charge de données et de travail parmi ces ordinateurs. Tandis que les briques de construction individuelles des solveurs directs creux et denses peuvent bénéficier de ces fonctionnalités avancées, leur exploitation à l'articulation entre les opérations creuses et denses n'est pas triviale. L'interface de programmation (API) des solveurs n'a simplement pas été conçue pour ce genre d'utilisation. En effet, peu importe le choix des briques de construction ou des fonctionnalités avancées, le résultat intermédiaire passé du solveur creux au solveur dense est sous forme d'une matrice non-compressée entièrement stockée en mémoire vive. Cette matrice a la taille du bloc dense (2, 2) dans le système d'origine ce qui représente un important désavantage pour les couplages de base. À cause de leur dimension (la taille du bloc dense en particulier), les problèmes auxquels nous nous intéressons ne peuvent pas être traités en utilisant un couplage de base.

La résolution de systèmes linéaires couplés creux et denses est cruciale au-delà du domaine de l'aéroacoustique. Plusieurs approches basées sur des méthodes directes ont été étudiées dans la litérature [61, 74, 129, 73, 116]. Dans [74, 129] l'objectif principal est de bénéficier du parallélisme en mémoire distribuée dans la partie dense du système pour réduire son coût en matière du temps de calcul et de la consommation mémoire tandis que la partie creuse ne nécessite même pas d'être parallélisée. Cependant, [61, 73, 116] considèrent des systèmes linéaires avec des parties creuses plus larges et proposent des schémas pour traiter en parallèle et la partie dense et la partie creuse. Le chapitre 2 fournit une analyse plus détaillée des approches connexes. Toutefois, nous soulignons que dans les études citées les tailles de problèmes traitées restent relativement petites. Ils peuvent donc être résolus grâce à un couplage de base des solveurs creux et denses de l'état de l'art comme nous l'avons vu auparavant. De plus, à notre connaissance, aucune des approches disponibles pour résoudre des systèmes linéaires couplés ne bénéficie de la compression numérique, du calcul *out-of-core* ou n'essaie de mettre en œuvres ces techniques en mémoire distribuée.

Le but de cette thèse est de concevoir et évaluer de nouveaux algorithmes permettant d'aller au-delà des limites actuelles des approches de résolution de l'état de l'art (chapitre 2) aussi bien sur un seul ordinateur en mémoire partagée que sur une grappe de machines pour le calcul de haute performance. D'un côté, nous proposons des algorithmes se basant sur l'API existante des solveurs directs creux et denses bien équipés et bien optimisés. Les couplages de bases qui s'appuient sur ces solveurs ne nous permettent de tirer pleinement avantage ni de la compression numérique, ni du calcul *out-of-core*, ni du parallélisme en mémoire distribuée dans le cadre du processus de résolution. Nos algorithmes visent à composer des briques de construction des solveurs sur des sous-matrices du système bien sélectionnées afin de pouvoir appliquer ces fonctionnalités avancées de manière efficace (chapitres 3, 4 and 5) dans les parties creuses et denses du système ainsi qu'à l'articulation entre les deux. En plus des expériences spécifiquement destinées à évaluer l'impact de ces fonctionnalités, nous avons mené une étude multi-métrique (chapitre 6) des algorithmes pour analyser le processus de calcul plus en détails. De l'autre côté, nous proposons un couplage de solveurs alternatif s'appuyant sur des solveurs directs choisis à base de tâches utilisant le même moteur d'exécution (chapitre 7). Dans ce contexte, nous explorons les possiblités d'adaptation de l'API des solveurs directs. Le but est d'assurer une meilleure composabilité et un passage plus simple de données entre le solveur creux et le solveur dense ainsi qu'un usage des ressources de calcul plus efficace. Tandis que changer des algorithmes et ajouter une nouvelle API dans un solveur direct bien équipé et développé par la communauté représente un travail à long terme à cause de la compléxité de son code source (plusieurs centaines de milliers de lignes de code), cette approche peut être implémentée dans le cadre d'un prototype moins bien équipé au prix de renoncer à certaines des fonctionnalités avancées.

En plus de la contribution scientifique principale, nous mettons un fort accent sur la réproductibilité de notre travail. Nous faisons usage d'un gestionnaire de paquets transactionnel garantissant la réproductibilité d'envrionnements logiciels à travers différentes machines et grappes de calcul de haute performance. En outre, nous nous appuyons sur les principes de la programmation lettrée dans un effort de maintenir une documentation exhaustive, claire et accessible de nos expériences et des environnements associés. Nous fournissons ce genre de journal de laboratoire pour toutes les expériences présentées dans cette thèse sous forme de rapports techniques exhaustifs tels que [30]. Un chapitre entier de la thèse (Chapitre 8) décrit notre méthodologie et propose un exemple fonctionnel minimal d'une étude expérimentale reproductible et des instructions permettant sa reproduction.

Ce manuscrit est organisé comme suit. Le chapitre 1 décrit le contexte de cette thèse ainsi que l'origine des systèmes linéaires couplés creux/denses FEM/BEM. Il fournit également une introduction aux méthodes de résolution de systèmes linéaires en général et détaille la solution

des systèmes linéaires couplés creux/denses FEM/BEM en particulier. Ensuite, le chapitre 2 présente l'état de l'art des approches existantes pour résoudre ce genre de systèmes linéaires et précise le positionnement de cette thèse vis-à-vis des travaux connexes et de l'état de l'art.

Dans le chapitre 3, nous introduisons deux nouvelles classes d'algorithmes se basant sur la compression numérique pour le traitement de systèmes linéaires couplés creux/denses FEM/-BEM relativement grands sur un seul ordinateur en mémoire partagée. Ils s'appuient sur les briques de construction des solveurs directs creux et denses bien équipés de l'état de l'art. Tandis que la compression numérique peut être directement appliquée sur des structures de données internes, le résultat intermédiaire passé du solveur creux au solveur dense est toujours stocké entièrement en mémoire vive sous forme d'une matrice dense non-compressée. Ceci devient rapidement une limitation pour le traitement de grands systèmes couplés. Les algorithmes proposés représentent un schéma alternatif pour l'application des briques de construction des solveurs permettant l'usage de la compression numérique pour la résolution de systèmes couplés FEM/BEM.

Le même problème nous empêche d'avoir recours au calcul *out-of-core* même si celui-ci peut-être appliqué aux structures de données internes des solveurs creux et denses de la même façon que la compression numérique. Le chapitre 4 présente alors l'extension des algorithmes proposés au calcul *out-of-core*. De plus, pour pouvoir traiter des systèmes couplés encore plus grands, nous adaptons les algorithmes pour le calcul en mémoire distribuée dans le chapitre 5.

Grâce à l'étude de multiples métriques de performance au même temps, nous pouvons mieux comprendre le comportement d'une implémentation et repousser ses limites dans le but de traiter de plus gros cas. La considération de l'empreinte carbone dans l'industrie en général et dans les centres de calcul en particulier amène la communauté scientifique et l'industrie à s'intéresser non seulement aux métriques habituelles telles que le temps de calcul, la consommation de la mémoire vive et de l'espace disque mais aussi à la consommation de l'énergie. Dans le chapitre 6, nous explorons le profil énergétique de la solution d'un système linéaire couplé FEM/BEM et évaluons la façon dont la consommation de la puissance varie avec le temps de calcul, le nombre d'opérations à virgule flottante, la quantité de mémoire utilisée et les choix algorithmiques disponibles au niveau du solveur.

Les algorithmes proposés dans le chapitre 3 nous donnent la possibilité de traiter des systèmes couplés FEM/BEM considérablement plus grands comparé aux couplages de base des solveurs de l'état de l'art. Cependant, ces algorithmes restent sous-optimaux et en termes du temps de calcul et en termes de la consommation mémoire. Dans le chapitre 7, ceci nous conduit à la conception d'un prototype de schéma d'implémentation s'appuyant sur des solveurs directs creux et denses à base de tâches utilisant le même moteur d'exécution. Le but est d'arriver à une implémentation idéale du point de vue de la performance ainsi que de l'exploitation de la symétrie et du caractère creux du système linéaire à résoudre.

Enfin, dans le chapitre 8, nous nous intéressons aux questions de reprodcutiblité. Nous y explorons les principes de la programmation lettrée ainsi que les outils logiciels destinés à garantir la reproductibilité et des environnements expérimentaux et des expériences elles-mêmes à travers différentes machines et de longs intervalles de temps.

Introduction

We are interested in the solution of very large linear systems of equations $Ax = b$, with the particularity of having both sparse and dense parts. In the sparse part, most of the coefficients are zeros which can be used to optimize the computation. In the dense part, the proportion of zero coefficients is not high enough to take advantage of it. Systems of this kind appear in various scientific fields such as electromagnetism, acoustics, study of structure-soil interaction or genomic prediction. In our case, the linear systems arise from numerical simulations in an industrial context when we couple two types of finite elements methods, namely the volume Finite Element Method (FEM) and the Boundary Element Method (BEM). This leads to a coupled sparse/dense FEM/BEM linear system with a coefficient matrix A having a distinctive 2×2 partitioning where the block $(1, 1)$ is a large sparse submatrix resulting from FEM, $(2, 1)$ a smaller sparse submatrix representing the FEM/BEM coupling and $(2, 2)$ a smaller dense submatrix resulting from BEM. In this work, most of the time we consider coupled systems where A is symmetric. However, we consider also a specific industrial situation where A is non-symmetric. In order to produce a result that is physically realistic, the number of unknowns in the system to solve can be extremely important (millions of BEM unknowns, hundreds of millions of FEM unknowns), which makes its treatment a computational challenge.

The original motivation for this thesis comes from the domain of aeroacoustics, which is the study of the coupling between acoustic phenomena and fluid mechanics. In the aeronautical industry, this discipline is used to model the propagation of acoustic waves in air flows enveloping an aircraft in flight. In particular, it allows one to simulate the noise produced at ground level by an aircraft during the takeoff and landing phases. Beyond its technical importance to validate that the regulatory environmental standards are met, it is also both a societal and an economic challenge. Manufacturers are enticed to reduce the noise level of their aircrafts for alleviating the impact on health. While designing future planes that can get authorized to access suburban airports, they can rely on numerical simulations of such aeroacoustic phenomena. Underlying physical models are expressed using partial differential equations. Therefore, prior to computing a numerical model, an approximation of its original physical expression is made over a limited domain using a suitable discretization technique. Aeroacoustic waves may interfere with various media. In the jet flow created by reactors, the propagation media, i.e. the air, is highly heterogeneous in terms of temperature, density and so on. Elsewhere, we approximate the media as homogeneous. To compute acoustic wave propagation in heterogeneous media, i.e. the jet flow, we use FEM leading to a sparse linear system. For homogeneous media, i.e. on the boundary of the aircraft and of the jet flow, we rely on BEM leading to a dense system. The coupling of these methods results in a sparse/dense FEM/BEM linear system we seek to solve (Chapter 1).

The two main classes of methods we can employ to solve a linear system are iterative and direct methods. On the one hand, the goal of iterative methods is to find an approximation of the solution starting from an initial guess and progressing iteratively. If it converges close enough to the exact solution in a reasonable amount of iterations, an iterative method can preserve its performance advantage over a direct approach. On the other hand, direct methods

are often based on an initial factorization step of A into a product of matrices making the linear system easier to solve. In the case of a symmetric linear system, the matrix A may, for instance, be factorized under the form $A = LL^T$ where L is a lower triangular matrix and T denotes the transpose operator. In the case of a non-symmetric matrix, A can be factorized under the form $A = LU$ where L is a lower-triangular and U an upper-triangular matrix. The low proportion of non-zero coefficients in a sparse matrix motivates the optimization of its memory footprint by avoiding any unnecessary storage of zeros. The sparse pattern also allows for reducing the algorithmic complexity of associated matrix operations. However, the factorization of a sparse matrix is likely to introduce new non-zero coefficients into the original matrix. This phenomenon is called *fill-in* [66]. If the fill-in is important, direct methods may lead to an important memory consumption. Nevertheless, when they fit in memory, they are extremely robust from a numerical point of view and represent a must-have in an industrial context where they are commonly employed to solve moderate to relatively large problems.

In this thesis, we focus on direct methods and investigate the opportunities to design efficient schemes for solving coupled sparse/dense linear systems. State-of-the-art sparse and dense direct solvers provide building blocks we can make use of. Most of the solvers provide baseline building blocks such as sparse factorization, dense factorization and the corresponding solve operations. We can directly apply them on the three submatrices (four in a non-symmetric case) of A and compose a coupled sparse/dense solver. Some fully-featured sparse direct solvers allow us to express that part of the unknowns in the linear system is associated with the dense submatrix. In this case, the sparse solver is aware of the entire coefficient matrix A and can perform the sparse operations in a more efficient way. Independently from the choice of building blocks, we refer to these straightforward couplings of state-of-the-art sparse and dense direct solvers as to vanilla couplings. However, with growing size of the coupled linear system and especially of the dense part, vanilla solver couplings may quickly run out of memory. To cope with this, we can resort to some advanced functionalities often implemented in fully-featured direct solvers. For instance, considering the scope of a single shared-memory workstation, numerical compression can be used to reduce the memory footprint as well as the time of computations. Thanks to out-of-core techniques (moving currently unused data to disk), one can further reduce the memory consumption. Parallel computation on multiple workstations then allow for solving even larger problems by distributing the load of data and computations among the workstations. While individual building blocks of the sparse and the dense direct solver can benefit from these advanced features, it is not trivial on the articulation between sparse and dense operations. The Application Program Interface (API) of the solvers was simply not designed for this kind of usage. Indeed, no matter the choice of building blocks or advanced features, the intermediate result passed from the sparse to the dense solver is in the form of a non-compressed dense matrix entirely stored in memory. This matrix has the size of the dense block (2, 2) in the original system which still represents a major drawback for vanilla solver couplings. Due to their dimension (the size of the dense block in particular), the problems we are interested in cannot be processed using a vanilla sparse and dense solver coupling.

Solution of coupled sparse/dense linear systems is crucial beyond the scope of aeroacoustics. Multiple approaches based on direct methods have been investigated in the literature [61, 74, 129, 73, 116]. In [74, 129], the main objective is to take advantage of distributed-memory parallelism within the dense part of the system to reduce its cost in terms of computation time and memory consumption while the sparse part does not even require parallel processing. However, [61, 73, 116] consider linear systems with larger sparse parts and propose schemes for processing both sparse and dense operations in parallel. We refer to Chapter 2 for a more detailed review. However, we highlight that, in the cited studies, the tackled problem size remains relatively small, which makes it possible to handle the system through a vanilla coupling of the state-of-the-art sparse and dense direct solvers as seen above. Moreover, to

the best of our knowledge, none of the available approaches for solving coupled systems benefit from numerical compression, out-of-core computation or try to use these techniques in a distributed-memory environment.

The goal of this thesis is to design and evaluate novel algorithms allowing for bypassing the current limits of the state-of-the-art solution approaches (Chapter 2) as well on a single multicore workstation as on high-performance distributed-memory parallel computers. On the one hand, we propose algorithms using the existing API of fully-featured and well-optimized sparse and dense direct solvers. Vanilla couplings based on top of these solvers do not allow us to fully take advantage of numerical compression, out-of-core computation and distributed-memory parallelism within the solution process. The aim of our algorithms is therefore to compose the solver building blocks on carefully chosen submatrices so as to efficiently apply these advanced features (chapters 3, 4 and 5) in both the sparse and the dense parts of the coupled system as well as on their articulation. In addition to the experiments meant to specifically evaluate the impact of these features, we conduct a multi-metric study (Chapter 6) of the algorithms to analyze the computation process in further details. On the other hand, we propose an alternative solver coupling relying on selected task-based direct solvers built on top of the same runtime (Chapter 7). In this context, we explore the possibilities of adapting the API of the direct solvers. The goal is to ensure a better composability and an easier data passing between the sparse and the dense solvers as well as a more efficient usage of computing resources. While changing the algorithms and adding a new API into a fully-featured community-driven direct solver would represent a long-time work due to the complex codebase of the latter (up to several hundred thousands lines of code), this approach can be implemented within a non fully-featured prototype at the cost of renouncing to some of the advanced features.

In addition to the main scientific contribution, we put a strong emphasis on ensuring the reproducibility of our work. We make use of a functional transactional package manager allowing for a complete reproducibility of software environments across different machines and high-performance computing clusters. Also, we rely on the principles of literate programming in an effort to maintain an exhaustive, clear and accessible documentation of our experiments and of the associated environment. We provide this kind of documentation for all the experiments presented in this thesis in the form of exhaustive technical reports such as [30]. However, an entire chapter of the thesis (Chapter 8) describes our methodology and proposes a minimal working example of a reproducible experimental study and associated reproducing guidelines.

The present manuscript is organized as follows. Chapter 1 describes the context of the thesis and the origin of the coupled sparse/dense FEM/BEM linear systems. It also provides an introduction to the solution methods for linear systems in general and details the solution of the coupled sparse/dense FEM/BEM linear systems in particular. Chapter 2 then draws a state of the art of existing approaches for solving this kind of linear systems and specifies the positioning of the thesis with respect to related work and the state of the art.

In Chapter 3, we introduce two new classes of algorithms relying on numerical compression techniques allowing for processing relatively large coupled sparse/dense FEM/BEM systems on a single shared-memory workstation. They are based on the building blocks of fully-featured state-of-the-art sparse and dense direct solvers. While numerical compression can be applied on internal data structures out-of-the-box, the intermediate result passed from the sparse to the dense solver is always stored entirely in memory in a non-compressed dense form. This quickly becomes a limitation for the processing of large coupled systems. The proposed algorithms represent an alternative scheme for applying the solver building blocks which enables the usage of numerical compression for the solution of coupled FEM/BEM systems.

The same limitation also prevents us from resorting to out-of-core computation although it can be applied on internal data structures of both the sparse and the dense direct solvers, just

as numerical compression. Chapter 4 thus presents the extension of the proposed algorithms to out-of-core computation. Furthermore, to allow for the processing of even larger coupled systems, we adjust the algorithms for distributed-memory parallelism in Chapter 5.

Thanks to the study of multiple performance metrics at the same time, one can better understand the behavior of an implementation and push its limits in order to handle larger cases. The consideration of carbon footprint issues in industry in general and in computing centers in particular leads the research community and the industry to consider not only the usual metrics such as the computation time, the consumed RAM and the disk space but also the energy consumption. In Chapter 6, we explore the energy profile of the solution of a coupled FEM/BEM linear system, and assess how the power consumption varies with the computation time, the flop rate, the amount of memory used and with the available algorithmic choices at the solver level.

Thanks to the algorithms proposed in Chapter 3, we can process considerably larger coupled FEM/BEM systems compared to a vanilla coupling of state-of-the-art solvers. However, the algorithms remain suboptimal in terms of both the computation time and the memory consumption. In Chapter 7, this leads to the design of a prototype implementation scheme relying on task-based sparse and dense direct solvers sharing the same runtime. The goal is to get on the track of an ideal implementation from the point of view of performance and the exploitation of sparsity and symmetry of the system.

Finally, in Chapter 8, we address the questions of reproducibility. To this end, we explore the principles of literate programming and software tools for ensuring the reproducibility of both experimental environments and numerical experiments themselves across different machines and extended periods of time.

Chapter 1

Context

Within this thesis, we focus on the solution of a particular kind of linear systems, i.e. the coupled FEM/BEM linear systems, arising from the simulations of aeroacoustic problems such as the propagation of sound waves around the aircrafts. The goal of the present chapter is to explain the origin of these linear systems and how to solve them numerically. We thus introduce the studied phenomenon and its initial continuous physical model in Section 1.1. In Section 1.2, we discuss the discretization of the physical model into a numerical one. Then, in Section 1.3, we give the expression of the latter using a system of linear equations. Solving the system finally leads to the result of the simulation. Our main interest resides precisely in the associated numerical solution methods. We thus explain methods for solving linear systems in general and coupled FEM/BEM systems in particular in sections 1.5 and 1.6, respectively.

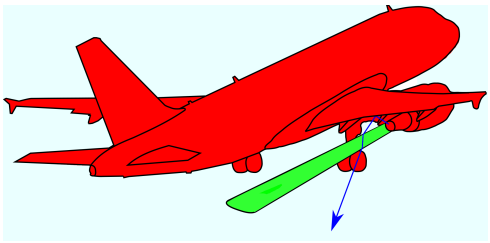
1.1 Continuous model

We are interested in a particular kind of aeroacoustic phenomenon and study the propagation of sound waves produced by an aircraft at take-off (see Figure 1.1). It can be seen as an acoustic wave propagation problem and expressed using Partial Differential Equations (PDE).

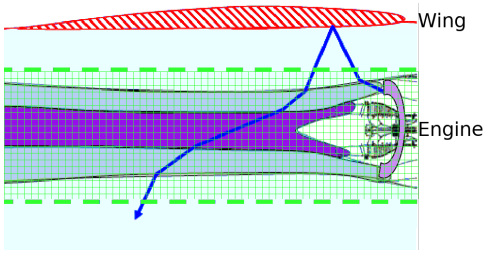
Figure 1.1: Airbus A350-900 XWB [5] at take-off producing a jet of exhaust gas traversed by an acoustic wave (blue arrow) from the engine, reflected on the wing and crossing the jet flow.

There are multiple media aeroacoustic waves may interfere with. The model takes into consideration aircraft's and engine's surfaces as well as exhaust gas and air flows (see Figure 1.2a). On the other hand, some aspects such as engine interior or retro-action sound waves are omitted to prevent the model from being unnecessarily too complex. Eventually, the model allows us to

study how acoustic waves produced by a jet engine propagate throughout a heterogeneous environment. Although some waves may only go through, for example, ambient air considered to be a homogeneous medium, others may traverse environments with varying parameters. Here, the homogeneous Helmholtz PDE is used to model the domains considered as homogeneous, such as the air surrounding the aircraft (see the light blue part in Figure 1.2b). On the contrary, the jet of exhaust gas produced by the aircraft's engine must not be considered as such. Both temperature and velocity vary inside of the jet flow depending on the engine operation condition (see the green-striped part in Figure 1.2b). This is represented thanks to an anisotropic second-order PDE.



(a) A global view where the red part represents the boundary of the media considered as homogeneous and the green part represents the media considered as heterogeneous.



(b) A zoom on the left figure where dashed lines delimit the green-striped domain considered heterogeneous including cold jet flow (dark blue) passing through the ducted fan, slower hot jet flow (violet) passing through the engine's core and a part of ambient air (light blue) for discretization to yield a simpler cylindrical 3D form.

Figure 1.2: An aeroacoustic wave (dark blue arrow) produced by the aircraft's engine, reflected on the wing and traversing the jet of exhaust gas and the ambient air (light blue).

1.2 Discrete model

The continuous physical model introduced in Section 1.1 is expressed using Partial Differential Equations (PDE) which are likely to involve concepts that can not be modelled on computers such as equations of integral functions. Therefore, prior to representing the model numerically, an approximation of its original physical expression must be made over a limited domain using an appropriate discretization technique. In this case, heterogeneous media are discretized using the Finite Elements Method (FEM) [49, 69, 108, 130] and the Boundary Elements Method (BEM) [46, 115, 126] is applied on media considered as homogeneous (see Figure 1.2).

The idea behind FEM is to partition the target domain into smaller parts referred to as *finite elements* such as tetrahedrons. Eventually, to each of these elements, a local linear equation is attributed, which approximates the original PDE on the selected part of the domain. The set of these linear equations forms a linear system that approximates the original continuous model to the extent of the domain of interest. Here, we model the jet flow beginning at the rear part of the aircraft's engine up to a limited distance from the engine and for a concrete frequency of the problem. On the other hand, the goal of BEM is to solve a given problem only on the *boundary values* of the considered domain. As the media it is applied on is considered as homogeneous, the method does not mesh over the entire domain, only over its surface. By definition, discretization of a three-dimensional problem using BEM makes it two-dimensional.

Both methods may be used together thanks to a coupling technique [55, 54] allowing one to

relate the unknowns of linear systems resulting from both methods with each other if, as in our case, the problems expose identical properties on the interface between FEM and BEM domains. Figure 1.3 shows examples of a 3D cylinder volume mesh resulting from a FEM discretization and a 2D surface mesh resulting from a BEM discretization, used on the outer surface of the volume mesh as well.

(a) A real-life case. (b) A numerical model example.

Figure 1.3: Example of a FEM/BEM discretization. The red mesh corresponds to the BEM discretization of the surface of the aircraft as well as the outer surface of the green 3D cylinder volume mesh. The latter represents the FEM discretization of the jet of exhaust gas produced by the aircraft's engine.

1.3 Problem

The discretization of the continuous model (see Section 1.1) based on the FEM/BEM coupling discussed in Section 1.2 leads to the global linear system in (1.1) featuring three main categories of unknowns: x_1 associated with the formulation of the FEM discretization on the three-dimensional domain corresponding to the jet exhaust flow, x_2 associated with the coupling where unknowns are shared between the BEM-discretized domain and the boundaries of the FEM-discretized domain on the exterior surface of the jet exhaust flow, x_3 associated with the formulation of the BEM discretization on the two-dimensional domain corresponding to the surface of the aircraft (see Figure 1.3b). The zeros in the coefficient matrix A, namely at A_{31} and A_{13}, indicate that there is no interaction between the FEM-discretized and the BEM-discretized domains in the matching part of the model.

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

In practice, x_2 and x_3 can be grouped to one unique surface mesh s related to the BEM discretization while x_1 is associated to the volume mesh v resulting from the FEM discretization. Consequently, in the simplified formulation of the system, x_2 and x_3 from 1.1 become x_s and x_1 becomes x_v:

$$\begin{matrix} R_1 \\ R_2 \end{matrix} \begin{bmatrix} A_{vv} & A_{sv}^T \\ A_{sv} & A_{ss} \end{bmatrix} \times \begin{bmatrix} x_v \\ x_s \end{bmatrix} = \begin{bmatrix} b_v \\ b_s \end{bmatrix}. \tag{1.2}$$

A in (1.2) is a 2×2 block symmetric coefficient matrix, where A_{vv} represents the action of the volume part on itself, A_{ss} represents the action of the exterior surface on itself, A_{sv} represents the action of the volume part on the exterior surface and A_{sv}^T is the transpose of A_{sv}. R_1 and R_2 respectively denote the first and the second block rows of the linear system.

The solution of (1.2) leads to the result of the numerical simulation of the acoustic pressure in different parts of the model (see Section 1.2). Key properties of the system with respect to the design of dedicated numerical solution methods, i.e. the total number of unknowns and the ratio of FEM-related unknowns x_v to BEM-related unknowns x_s, depend on the simulated acoustic frequencies as well as the extent of the model. In a model considering a full aircraft (see Figure 1.3b) and the acoustic frequency up to 20 kHz inducing edge sizes below 1 cm, the number of unknowns x_v and the number of unknowns x_s can be extremely high. It grows like the cube and the square of the simulated acoustic frequency, respectively. Also, such a model has a larger surface mesh and consequently a higher ratio of x_s to x_v compared to a more reduced model, e.g. representing only a wing and a part of the aircraft's fuselage (see Figure 1.6), or to an academic model, e.g. considering only the boundary of the volume domain (see figures 1.4a and 1.4b).

From an industrial point of view, it is crucial to be able to reduce the cost of these simulations in terms of memory usage and time, in order to tackle the largest possible spectrum of audio-frequencies on existing workstations and high-performance computing clusters. The main focus of this thesis is therefore to propose efficient numerical solution methods for coupled sparse/dense FEM/BEM linear systems such as (1.2). For the evaluation of the proposed methods, we rely on multiple test models introduced in Section 1.4.

1.4 Test models

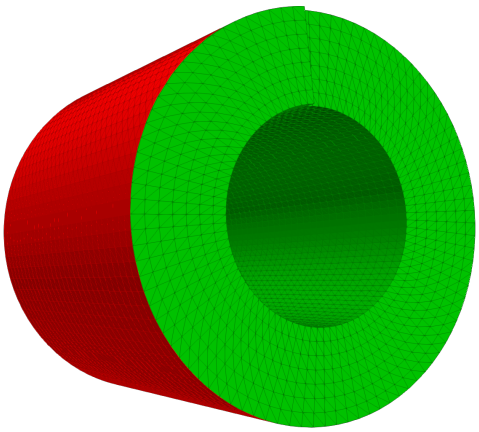
Most of the time, we assess the proposed numerical solvers for coupled FEM/BEM systems such as (1.2) on open-source academic test cases (see Section 1.4.1) available to the scientific community [22]. They are easy and fast to generate in arbitrarily large sizes and still yield linear systems which are close enough to those arising from real-life models (see Figure 1.3). In addition to that, we validate the solvers on an industrial test case (see Section 1.4.2).

1.4.1 Academic cases

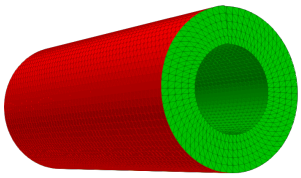
Academic test cases have a simplified pipe shape (see Figure 1.4). We can conveniently choose arbitrary pipe size, radius as well as the total number of unknowns in the corresponding coupled FEM/BEM linear system. In this thesis, we perform benchmarks on a *wide pipe* and a *narrow pipe* test cases. Both are 4 meters long. However, the *wide pipe* (see Figure 1.4a) has a radius of 2 meters and the *narrow pipe* (see Figure 1.4b) has a radius of 0.8 meters. Smaller radius reduces the volume domain and therefore leads to a higher proportion of BEM-related unknowns x_s in the *narrow pipe* compared to the *wide pipe*.

The vertices of the pipe mesh represent the unknowns in the linear system coming out of the problem discretization (see Section 1.3). In the volume part of the pipe (green portion in Figure 1.4), discretized using FEM, each vertex interacts solely with its immediate neighbors. In the outer surface of the pipe (red portion in Figure 1.4), discretized using BEM, each vertex interacts with all the others. All the unknowns rely on the same mesh. The unknown count depends on the wavelength λ of the physical problem being simulated. The more there are vertices in the mesh, the more there are unknowns and the more the model is accurate (see Figure 1.5). For the experiments, we consider the wavelength parameter λ to be set such that there are 10 vertices per wavelength.

The coefficient matrix A in the resulting coupled sparse/dense FEM/BEM linear system can be symmetric or non-symmetric. The elements of A can be either real or complex, using simple or double precision arithmetic and are defined depending on the associated category of unknowns (see Section 1.3). For two unknowns x and y associated with the BEM discretization, with

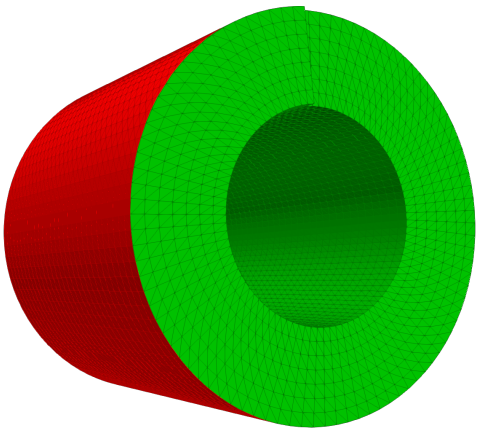


(a) *wide pipe*
(length: 4 m, radius: 2 m)

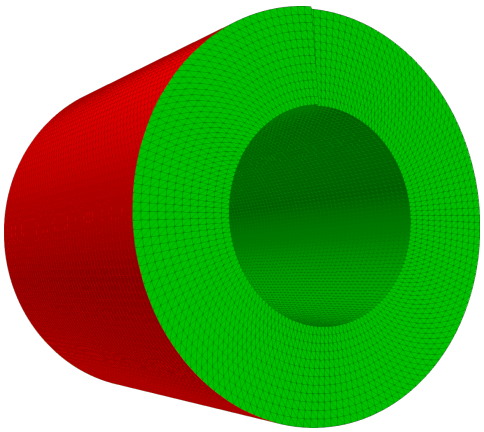


(b) *narrow pipe*
(length: 4 m, radius: 0.8 m)

Figure 1.4: Academic test cases in shape of pipe with BEM surface mesh in red and FEM volume mesh in green.



(a) *wide pipe* couting 20,000 vertices.



(b) *wide pipe* counting 100,000 vertices.

Figure 1.5: *wide pipe* with different number of vertices. Both images use the same scale.

$k = 2\pi/\lambda$ and r the distance between these unknowns, the interaction a_{xy} of the latter is defined as follows:

$$a_{xy} = \frac{e^{ikr}}{4\pi r}. \tag{1.3}$$

If r is 0, we set r to the size of an average mesh step divided by 2. Then, for two unknowns x and y associated with the FEM discretization (connected by one edge), the interaction a_{xy} of these two unknowns is defined as follows:

$$a_{xy} = \begin{cases} 0.05 \times (1 - \frac{i}{3}) & \text{if } x \neq y \\ 0.1 \times (1 - \frac{i}{3}) & \text{if } x = y. \end{cases} \tag{1.4}$$

Note that this is not the way the interactions are determined in real-life aeroacoustic models. Nevertheless, the definitions in (1.3) and (1.4) allow for a simple construction of coefficient matrices for testing purposes while providing values that are similar enough to real-life cases.

In the experimental evaluations, we consider symmetric coupled FEM/BEM linear systems of various sizes. N, n_{FEM} and n_{BEM} denote the total number of unknowns in the system, the count of FEM-related unknowns and the count of BEM-related unknowns, respectively. Regarding the *wide pipe* test case, the values of n_{FEM} and n_{BEM} for each value of N we consider in this thesis are detailed in Table 1.1.

Total unknowns N	# FEM-related unknowns n_{FEM}	# BEM-related unknowns n_{BEM}
50,000	44,960	5,040
100,000	91,891	8,109
250,000	235,165	14,835
500,000	476,423	23,577
1,000,000	962,831	37,169
1,300,000	1,255,732	44,268
1,500,000	1,451,250	48,750
2,000,000	1,941,090	58,910
2,500,000	2,431,476	68,524
3,000,000	2,922,756	77,244
4,000,000	3,906,407	93,593
5,000,000	4,891,562	108,438
6,000,000	5,878,057	121,943
7,000,000	6,864,384	135,616
8,000,000	7,852,006	147,994
9,000,000	8,839,766	160,234
12,000,000	11,805,780	194,220
14,000,000	13,784,898	215,102
15,000,000	14,774,880	225,120
28,000,000	27,658,780	341,220
34,000,000	33,611,392	388,608
40,000,000	39,567,785	432,215

Table 1.1: Counts of FEM-related and BEM-related unknowns for all the coupled FEM/BEM linear systems considered within the thesis and arising from the *wide pipe* test case.

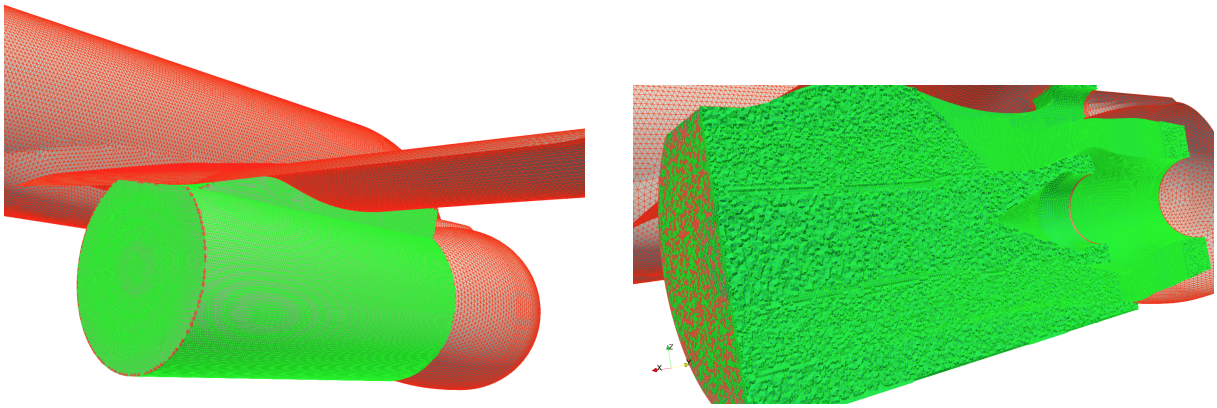
For the *narrow pipe* test case, the values of n_{FEM} and n_{BEM} are given in Table 1.2.

Total unknowns N	# FEM-related unknowns n_{FEM}	# BEM-related unknowns n_{BEM}
1,000,000	949,800	50,200
2,000,000	1,920,116	79,884
3,000,000	2,895,744	104,256
4,000,000	3,873,517	126,483
5,000,000	4,852,940	147,060
6,000,000	5,834,472	165,528
7,000,000	6,816,640	183,360

Table 1.2: Counts of FEM-related and BEM-related unknowns for all the coupled FEM/BEM linear systems considered within the thesis and arising from the *narrow pipe* test case.

1.4.2 Industrial case

The industrial test case we rely on is illustrated in Figure 1.6. It features 2,090,638 volume unknowns x_v and 168,830 surface unknowns x_s. The proportion of surface unknowns is significantly higher than in the considered academic cases (see Section 1.4.1). Indeed, in the pipe the surface mesh covers only the outer surface of the jet flow (i.e. the volume mesh), whereas in this industrial test case it also includes the wing and part of the fuselage of the aircraft. Due to the physical model used, the coefficient matrix A is complex and non-symmetric.



(a) A global view.

(b) A vertical cut-plane showing the inside of the reactor and the flow: the green mesh is made of tetrahedra, while the red mesh is hollow and made of triangles.

Figure 1.6: Industrial test case with BEM surface mesh in red (the right part of the plane, the wing and the engine) and FEM volume mesh in green (the jet flow).

1.5 Solution of linear systems

Prior to focusing on the solution methods shaped for the target coupled FEM/BEM linear system (1.2) in particular, we give an overview of numerical solving techniques for linear systems in general. We also discuss the accuracy of the numerical solution in Section 1.5.2.

We consider two classes of linear systems, sparse and dense. In sparse linear systems, the coefficient matrix is mostly composed of zeros and has only a few non-zero entries. In dense linear systems, it is the opposite. There is no universal condition for deciding whether a matrix

should be considered sparse or not. In practice, a matrix can be characterized as sparse when the computation involving the matrix can take advantage of the high count of zero entries and their locations [114].

Regardless its sparsity, a linear system may be solved using either an iterative or a direct approach. The goal of iterative methods is to find an approximation of the solution. A sequence of terms $x_1,...,x_n$ is calculated, in which a given term x_i is based on previous ones and approximates the solution x. The idea is to converge as much as possible to the exact solution while preserving the performance advantage over direct methods. For solving coupled FEM/BEM systems, numerous iterative approaches have been proposed in the last decades [124, 51, 106, 84, 92, 91, 94, 109, 83, 50, 116].

We remind the reader that, in this thesis, we exclusively focus on direct methods. In a sparse context, direct methods are known to possibly consume a large amount of memory due to fill-in [66] (earlier announced in Introduction and further discussed in Section 1.5.1.2). On the other hand, when they fit in memory, they are extremely robust from a numerical point of view and represent a must-have in an industrial context where they are commonly employed to solve moderate to relatively large problems.

1.5.1 Direct methods

Instead of computing the inverse of the coefficient matrix, direct methods generally rely on an initial decomposition or factorization of the latter into a product of matrices making the linear system easier to solve. For example, the LU factorization algorithm decomposes the coefficient matrix into a lower triangular matrix L and an upper triangular matrix U transforming an initial system of form $Ax = b$ to $LUx = b$ which can be split into a couple of equations such as $Ly = b$ and $Ux = y$. These triangular systems may be eventually solved using the so-called forward substitution for the first equation and the so-called backward substitution for the second one.

The LU factorization first appeared in the work of the Polish astronomer Tadeusz Banachiewicz from 1938 [45, 119]. It is also applicable on non-symmetric matrices unlike the earlier Cholesky decomposition, named after André-Louis Cholesky and published in 1924 after his death [58, 119], working only with symmetric matrices. These procedures may be considered as matrix forms of Gaussian elimination, described by Leonhard Euler as the most natural way of solving simultaneous linear equations [70] and the traces of which have been discovered in multiple ancient sources [78]. Other factorization methods for symmetric matrices appeared in the last decades [52]. For instance, the Cholesky procedure can be viewed as an $A = LL^T$ decomposition where L is a lower triangular matrix and L^T the transpose of L. Another example is LDL^T factorization where D is a diagonal or a block diagonal matrix (see also [52]).

In this thesis, we rely on the LL^T factorization for complex (symmetric but not positive definite) matrices. In case of real symmetric matrices, we use the LDL^T factorization instead. For non-symmetric matrices, we resort to the LU factorization.

1.5.1.1 Hierarchical low-rank approximation

As mentioned above, the main drawback of direct methods certainly is a high consumption of computing resources. Hierarchical low-rank or $\mathcal{H}$-matrices [81] represent a way to store matrices in a compressed form. It is an algebraic hierarchical structure consisting of either full-rank or compressed low-rank submatrices (see Figure 1.7). Being a major advantage of the format, the complexity of a matrix-matrix product of two dense matrices may be, under certain conditions, lowered from $\mathcal{O}(n^3)$ to $\mathcal{O}(nlog(n))$ [82] thanks to data compression. Despite

the compression, the implied loss of accuracy remains acceptable in our context as further discussed in Section 1.5.2.3.

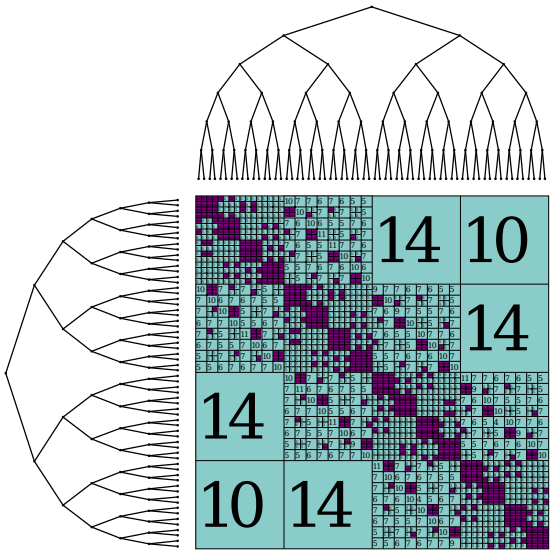


Figure 1.7: An $\mathcal{H}$-matrix where violet submatrices are stored as full-rank and light blue ones are in the compressed low-rank format with the inner values indicating the rank.

1.5.1.2 Sparse systems

Moreover, in the case of sparse systems, we can take advantage of the high number of zero entries in the coefficient matrix and reduce the time and the memory footprint of the computation by preventing the storage of null coefficients. However, the factorization of a sparse matrix is likely to cause some initially zero entries to become non-zero. This effect, called fill-in [76] (see Figure 1.8), is directly related to the process of Gaussian elimination. Depending on the order in which the unknowns in the linear system are eliminated, the fill-in may be more or less important. There are multiple reordering techniques to reduce the fill-in [37, 98, 105, 110, 75]. The problem is generally NP-complete [127]; so these techniques have to rely on heuristics. The aim is to search for an optimal rearrangement of the equations and the unknowns in the linear system so as to reduce the appearance of new non-zero entries during the factorization.

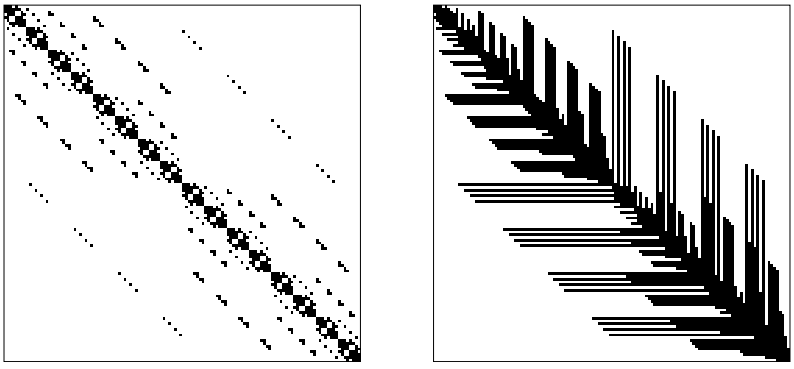


Figure 1.8: Example of a sparse matrix before factorization (left) and after an LU factorization (right) demonstrating the fill-in effect. Black squares represent non-zero entries.

The goal being to store only the non-zero elements of a sparse matrix, it is useful to estimate the memory that has to be allocated for the factorization phase. This is the role of symbolic factorization [97] which is a simulation of the actual factorization based on the zero/non-zero

pattern of the matrix and allows one to locate where the fill-in appears. The direct solution of a sparse system usually features more computation steps in addition to factorization and solve: reordering of unknowns and symbolic factorization. We shall not further describe these steps in this thesis. We refer the reader to [97, 66] for more details.

1.5.2 Solution accuracy

The quality of the numerical solution of a linear system depends on various factors. On the one hand, we have the errors due to the approximation of the concepts of continuous mathematics such as real and complex numbers on computers working in finite precision arithmetic. On the other hand, the design of the algorithms involved in the computation comes into play. For example, in many cases, we can opt for a solution with lower accuracy (while remaining on an acceptable level) in exchange for a significant decrease in computation time and memory consumption. Eventually, it is important to be able to evaluate the accuracy of the numerical solution, which is only an approximation of the exact solution to the system.

1.5.2.1 Forward error

The forward error is an approach to measure the accuracy of a numerical approximation of the exact solution to a given linear system [86]. In this section, let $Ax = b$ be a linear system, where A is the coefficient matrix, x the unknown solution vector and b the right-hand side vector. Following [86], we note $\hat{x}$ a numerically computed approximation of the exact solution x.

The absolute forward error E_{abs} of $\hat{x}$ is defined as the difference between the exact solution and its approximation:

$$E_{abs}(\hat{x}) = \|\hat{x} - x\|.$$

Then, the relative forward error of $\hat{x}$, referred to as E_{rel}, corresponds to:

$$E_{rel}(\hat{x}) = \|\hat{x} - x\|/\|x\|. \tag{1.5}$$

In practice, the exact solution x is not known and therefore, we have to rely on an estimation of $E_{rel}(\hat{x})$. However, in order to assess a numerical solver, we can make use of an artificial test case allowing us to actually compute the relative forward error. We give ourselves A and x as well as a right-hand side b such that:

$$b = Ax.$$

Then, assuming that the computation of $b = Ax$ was exact, we solve $A\hat{x} = b$ using a numerical method:

$$\hat{x} = A^{-1}b$$

which involves the factorization of A and the solve operation performing the so-called backward and forward substitutions on the resulting triangular systems (see Section 1.5). As both x and $\hat{x}$ are eventually known, we are able to determine $E_{rel}(\hat{x})$ (see Equation 1.5).

1.5.2.2 Machine precision

A machine working in finite precision arithmetic can be used to simulate an arbitrarily high precision, which means that theoretically, the accuracy of the solution is not limited by the machine precision [86]. However, within this study, we do not consider such a simulation. Therefore, we assume that the accuracy of a numerically computed solution using a direct method (see Section 1.5) is proportional to the precision of the target machine provided that the problem is not ill-conditioned. According to [86, 125], the problem is considered ill-conditioned when the condition number of the matrix A is high. This means that inaccuracies in A or b may have a great impact and significantly worsen the accuracy of the solution approximation. The machine precision, noted u, is defined as the difference between 1.0 and the next greater floating-point value [86]. When using a 64 bits IEEE floating-point representation, u is approximately $2^{-53} \approx 1.11 \times 10^{-16}$ [77, 39], assuming u rounded to the nearest representable number.

1.5.2.3 Trade-off on accuracy

Computing in high precision may be too consuming in terms of time and memory. With the aim to reduce the cost of the computation, solvers providing some kind of data compression can be instrumented to compute an approximate solution $\hat{x}$ of a system $Ax = b$ in a lower precision. Eventually, the relative forward error of $\hat{x}$ (see Section 1.5.2.1) is higher but the accuracy of the result remains acceptable in our context.

There is no absolute guarantee of achieving such a result, especially in the case of very ill-conditioned problems [86]. However, when the problem is not ill-conditioned, a recent work proved that it is possible to guarantee for a particular kind of compression mechanism to yield a solution approximation verifying the target accuracy [87].

To control the accuracy of solution approximations, compressed solvers usually expose a threshold parameter referred to as ϵ. During the compression, all the values that are smaller than the compression threshold times the maximum value are discarded, i.e. they are rounded to zero. At Airbus, in research [101, 71] and further in engineering work, setting ϵ to 10^{-3} is considered enough to obtain satisfying results.

1.6 Direct solution of FEM/BEM systems

In Section 1.3, we defined the coupled FEM/BEM linear system (1.2) we want to solve. In Section 1.5, we presented different direct methods for solving linear systems depending on their sparse or dense character. In this section, we address the particular partitioning and different sparsity levels, due to the FEM/BEM coupling, in the coefficient matrix A of the system. We then focus on the application of the sparse and dense direct methods on the numerical solution of the latter.

1.6.1 Properties

In our case, the linear systems resulting from FEM are termed sparse as each element of the corresponding mesh interacts only with its direct neighbors. The linear systems resulting from BEM are termed dense as each element of the associated mesh interacts with all the others.

Eventually, in the target system resulting from the coupling of both methods (1.2), A_{vv} is a large and sparse submatrix, A_{sv} is another sparse submatrix and A_{ss} is a smaller dense submatrix (see

Figure 1.9). Note that the relative dimensions of these submatrices within A are defined by the ratio of volume unknowns x_v to surface unknowns x_s in the system (see Section 1.3).

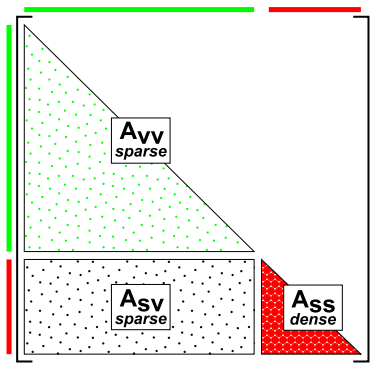


Figure 1.9: Internal dimensions, partitioning and sparsity of A in (1.2).

1.6.2 Formulation

The first step of a direct solution of (1.2) consists of reducing the problem on the boundaries and simplifying the system to solve. Based on its first block row R_1, we express x_v as:

$$x_v = A_{vv}^{-1}(b_v - A_{sv}^T x_s)\,. \tag{1.6}$$

Then, substituting x_v in R_2 by (1.6) yields a reduced system without x_v in R_2. This represents one step of Gaussian elimination, i.e. $R_2 \leftarrow R_2 - A_{sv}A_{vv}^{-1} \times R_1$:

$$\begin{matrix} R_1 \\ R_2 \end{matrix} \begin{bmatrix} A_{vv} & A_{vs} \\ 0 & A_{ss} - A_{sv}A_{vv}^{-1}A_{sv}^T \end{bmatrix} \times \begin{bmatrix} x_v \\ x_s \end{bmatrix} = \begin{bmatrix} b_v \\ b_s - A_{sv}A_{vv}^{-1}b_v \end{bmatrix}. \tag{1.7}$$

The expression $A_{ss} - A_{sv}A_{vv}^{-1}A_{sv}^T$, which appears in R_2 in (1.7), is often referred to as the Schur complement [128], denoted S, associated with the partitioning v and s of the variables in the system (see Section 1.3).

Basically, we have to compute S:

$$S = A_{ss} - A_{sv}A_{vv}^{-1}A_{sv}^T \tag{1.8}$$

and find x_s by solving the reduced Schur complement system matching R_2 in (1.7) after elimination of x_v:

$$x_s = S^{-1}(b_s - A_{sv}A_{vv}^{-1}b_v)\,. \tag{1.9}$$

Once we have computed x_s, we use its value to determine x_v according to (1.6).

The decomposition of A (see Section 1.3) and the choice to eliminate x_v from R_2 in (1.2) allows one to take advantage of the sparsity of the submatrix A_{vv} during the solution process. On the contrary, eliminating x_s from R_1 instead of the current choice would result in an important fill-in (see Section 1.5.1.2) of A_{vv}, where the Schur complement would have been computed.

When solving the problem numerically, rather than actually computing the inverses of A_{vv} and S, we factorize A_{vv} as well as S into products of matrices making the equations easier to solve (see Section 1.5.1).

1.6.3 Ideal symmetry and sparsity of the factorized coupled system

In this section, we describe the ideal approach for the numerical computation of the solution of (1.2) which would allow us to fully take advantage of the symmetry of the system by containing the computation in its lower symmetric part as well as to exploit the sparse pattern of A_{vv} and A_{sv} as much as possible.

In theory, we would begin by computing S which corresponds to the core step of the entire solution process. To do this, we would factorize A_{vv} into $L_{vv}L_{vv}^T$ and compute $A_{sv}(L_{vv}^T)^{-1}$ in order to express S as:

$$S = A_{ss} - [A_{sv}(L_{vv}^T)^{-1}][A_{sv}(L_{vv}^T)^{-1}]^T \,. \tag{1.10}$$

Then, we would factorize S into $L_S L_S^T$ and finally compute the solutions x_s and x_v using:

$$\begin{cases} x_s &= (L_S L_S^T)^{-1}(b_s - A_{sv}(L_{vv}L_{vv}^T)^{-1}b_v) \\ x_v &= (L_{vv}L_{vv}^T)^{-1}(b_v - A_{sv}^T x_s)\,. \end{cases} \tag{1.11}$$

In practice, exploiting the sparsity of A_{vv} and A_{sv} is a hard task. It requires to resort to advanced techniques, including symbolic factorization and management of complex data structures, e.g. to cope with arising dense submatrices due to fill-in (see Section 1.5.1.2) in both A_{vv} and A_{sv} (see Figure 1.10).

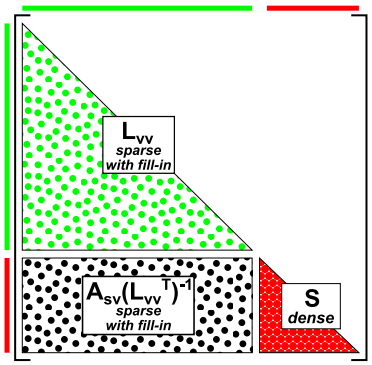


Figure 1.10: Internal dimensions, partitioning and sparsity of A in (1.2) when using the ideal approach for computing S according to (1.10).

This is what sparse direct solvers are meant for. They can take care of numerical, combinatorial and performance issues for us. In particular, thanks to them, we can perform the proper factorization of A_{vv} and benefit from reordering techniques (see Section 1.5.1.2) to limit fill-in as well as from the performance of BLAS-3 operations. If we wanted to implement the above approach by ourselves, it would require to, at least partially, implement a sparse direct solver. The necessary effort would not only be extremely time-consuming (e.g. the version 5.1.2 of MUMPS, introduced in Section 2.6.4.1, has 418,556 lines of code) but might also lead to an under-performing implementation.

In this thesis and from an industrial perspective, we instead decided to build our coupled solver on top of existing sparse and dense solvers. This also means that we have to deal with their API. Chapter 2 draws a state of the art of different capabilities of sparse and dense direct solvers and introduces some possible ways of exploiting them to design a coupled sparse/dense solver for (1.2).

Chapter 2

State of the art

In Chapter 1, we introduced the aeroacoustic context of this thesis and the related coupled sparse/dense FEM/BEM linear systems we seek to solve. Based on an overview of different general numerical techniques for solving linear systems, we explained a direct approach for solving the target coupled FEM/BEM system (1.2). This chapter deals with existing software, its characteristics and features as well as the possible ways of exploiting it for the design of a coupled sparse/dense FEM/BEM solver. In Section 2.1, we identify various building blocks available in the API of the sparse and dense direct solvers. Then, in Section 2.2, we describe how we can make use of these building blocks to implement a coupled solver and what are the limitations of such couplings. In Section 2.4, we go through the related work found in the literature. In Section 2.5, we position this thesis with respect to the state of the art. In Section 2.6, we discuss selected sparse and dense direct solvers from the point of view of their different characteristics and functionalities.

2.1 Building blocks

In order to implement a coupled sparse/dense direct solver for the target linear system (1.2) following the computation steps described in Section 1.6, we can make use of different building blocks of sparse and dense direct solvers. This section introduces the relevant building blocks.

2.1.1 Sparse direct solver building blocks

Most sparse direct solvers do not allow one to express that part of the unknowns is associated with a dense block. In this case, only the A_{vv} block (see (1.2) in Section 1.3) can be handled with the sparse direct solver and other operations must be handled on top of that, leading to a suboptimal scheme for the reasons discussed in Section 1.6.3. We call this first scenario the baseline usage of the sparse direct solver (see Section 2.1.1.1). Nonetheless, some fully-featured sparse direct solvers, such as MUMPS [38], PaStiX [85] or PARDISO [118], provide in their API a Schur complement functionality (for MUMPS see option `ICNTL(19)` in [56], for PaStiX see [89] and for PARDISO see Section 1.3 in [117]). It allows one to delegate the computation of S entirely to the sparse direct solver, which can (it is designed for that) fully exploit the symmetry and the sparsity of the system according to the ideal scenario described in Section 1.6.3. We call this scheme the advanced usage of the sparse direct solver (see Section 2.1.1.2).

2.1.1.1 Baseline usage

In the first scenario, we use the sparse direct solver only on the A_{vv} block, for performing the *sparse factorization* of the A_{vv} submatrix into $L_{vv}L_{vv}^T$ and for computing $(L_{vv}L_{vv}^T)^{-1}A_{sv}^T$ using a *sparse solve* step like in (1.8).

2.1.1.2 Advanced usage

In the second scenario, we rely on the aforementioned Schur complement functionality. This feature consists of the factorization of A_{vv} and the computation of the Schur complement $-A_{sv}A_{vv}^{-1}A_{sv}^T$ associated with the $\begin{bmatrix} A_{vv} & A_{sv}^T \\ A_{sv} & 0 \end{bmatrix}$ matrix. This functionality represents a building block on its own. In the following, we shall refer to the latter as to *sparse factorization+Schur* step. Due to the API of sparse direct solvers that support this functionality, the resulting Schur complement is returned as a non-compressed dense matrix stored entirely in RAM [56, 89, 117], which will still represent a limitation in our context as discussed further.

2.1.2 Dense direct solver building blocks

Once the Schur complement is obtained with either the baseline or the advanced usage of sparse direct solver, a dense direct solver may be used for some of the operations associated with (1.11), i.e. the *dense factorization* of S and the *dense solve* for computing x_s.

2.2 Vanilla couplings

In Section 2.1, we introduced various sparse and dense direct solver building blocks. Here, we discuss possible schemes for applying these building blocks directly on the submatrices A_{vv}, A_{sv} and A_{ss} of (1.2) in order to compose a coupled sparse/dense solver. As explained in Introduction, throughout the thesis, we refer to these straightforward couplings of state-of-the-art sparse and dense direct solver as to vanilla couplings.

2.2.1 Baseline sparse/dense solver coupling

A possible way of composing these building blocks is to rely on the baseline usage of the sparse direct solver (Section 2.1.1.1). This leads to Algorithm 1. The first step (line 2) of the solution process is thus a *sparse factorization* of A_{vv} into $L_{vv}L_{vv}^T$. The factorization is followed by a *sparse solve* step (line 3) to get $Y = (L_{vv}L_{vv}^T)^{-1}A_{sv}^T$ (1.8), which is, in this baseline usage, non optimally retrieved as a dense matrix entirely stored in RAM. From a combinatorial perspective, Y is not dense. However, taking advantage of its sparsity is far from trivial (see Section 1.6.3). It is possible to exploit the sparsity of the operands during the *sparse solve* [35] (for MUMPS see option `ICNTL(20)` in [56], which we always turn on in this thesis). Nevertheless, because the internal data structures are complex, the user still gets, as in all fully-featured direct solvers we are aware of, the output as dense.

A sparse-dense matrix multiplication (SpMM) then follows (line 4) to compute $Z = A_{sv}Y$, for which it is not evident to exploit the sparsity either. Indeed, Y is retrieved entirely in RAM as a dense matrix while A_{sv} is a 'raw' sparse matrix yielding a sub-optimal arithmetic intensity in addition to useless computation on the zeros stored in Y. The subtraction $A_{ss} - Z$ (line 5) finally yields S (see Figure 2.1).

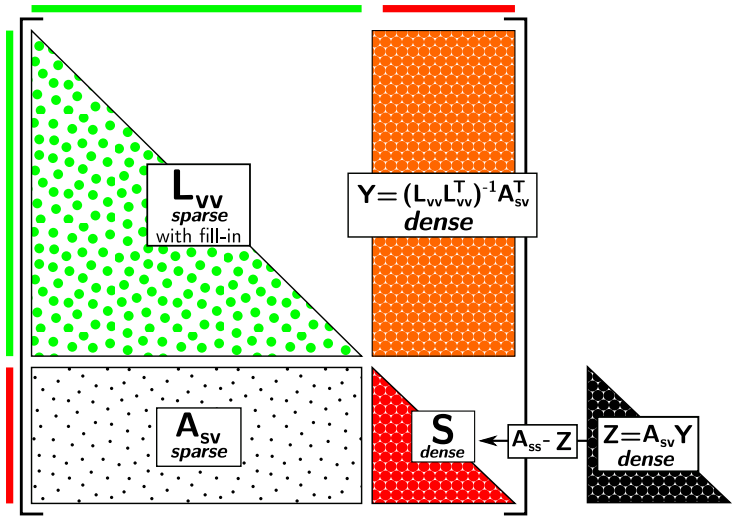


Figure 2.1: Computation of S in the baseline sparse/dense solver coupling (see Algorithm 1).

In the next stage, we compute the solutions x_s and x_v following (1.11). At first, we form the right-hand side $b_s - A_{sv}(L_{vv}L_{vv}^T)^{-1}b_v$ by performing the *sparse solve* $b_v = (L_{vv}L_{vv}^T)^{-1}b_v$ (line 6), the sparse matrix-vector product $A_{sv}b_v$ and a final vector subtraction (line 7). Then, we perform the *dense factorization* of S (line 8) and a *dense solve* to determine x_s (line 9). At the end, we compute the sparse matrix-vector product $A_{sv}^T x_s$ and after a vector subtraction, we compute the *sparse solve* of $(L_{vv}L_{vv}^T)^{-1}(b_v - A_{sv}^T x_s)$ to get x_v (line 10).

Algorithm 1: Vanilla baseline coupling algorithm for computing S based on (1.8) and solving (1.2).

```
1  Function VanillaBaselineCoupling(A, b):
2      A_vv ← SparseFactorization(A_vv)
3      Y ← SparseSolve(A_vv, A_sv^T)
4      Z ← A_sv × Y                                  ▷ SpMM
5      A_ss ← A_ss − Z                               ▷ AXPY
6      b_v ← SparseSolve(A_vv, b_v)
7      b_s ← b_s − A_sv b_v
8      A_ss ← DenseFactorization(A_ss)
9      x_s ← DenseSolve(A_ss, b_s)
10     x_v ← SparseSolve(A_vv, b_v − A_sv^T x_s)
```

2.2.2 Advanced sparse/dense solver coupling

Alternatively, as proposed in Algorithm 2, we can use a *sparse factorization+Schur* step (see Section 2.1.1.2). Thanks to this building block, the sparse direct solver yields (line 3) the Schur complement $-A_{sv}A_{vv}^{-1}A_{sv}^T$, denoted X, associated with the $\begin{bmatrix} A_{vv} & A_{sv}^T \\ A_{sv} & 0 \end{bmatrix}$ matrix, denoted W. Because the system is symmetric, we do not have to explicitly store either the upper triangular part of A_{vv} or A_{sv}^T in the W matrix (line 2). We sum the resulting matrix X with A_{ss} based on (1.10) to get S (see Figure 2.2). The advantage of doing so is that we benefit from the fine management of the sparsity and efficient usage of BLAS-3 on non-zero blocks, transparently offered by the sparse direct solver, up to the computation of the Schur. Internally, the solver follows the ideal approach for computing S described in Section 1.6.3, therefore benefiting from all the optimized operations. However, as discussed in Section 2.1.1.2 and recaped below, the

Schur complement matrix returned by the *sparse factorization+Schur* step is dense and entirely stored in RAM. The computation steps following the computation of S (lines 5 - 9) then remain identical to the ones of the baseline coupling (see Section 2.2.1).

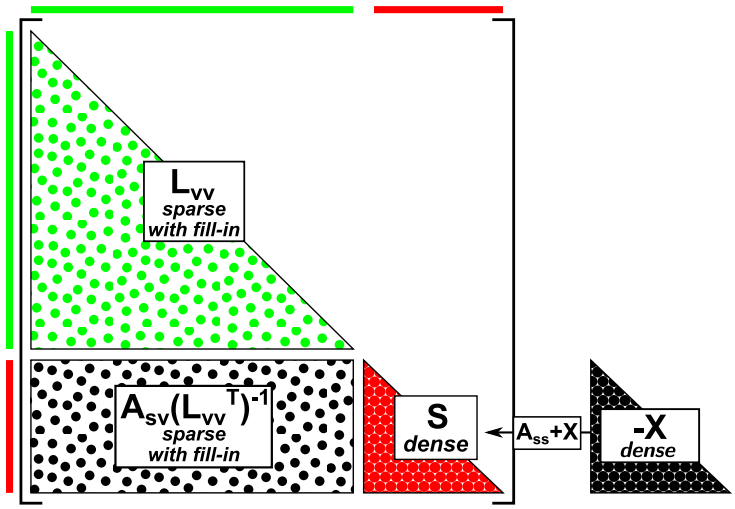


Figure 2.2: Computation of S in the advanced sparse/dense solver coupling (see Algorithm 2).

Algorithm 2: Vanilla advanced coupling algorithm for computing S based on (1.10) and solving (1.2).

1 **Function** `VanillaAdvancedCoupling`(A, b)**:**
  ▷ A_{sv}^T within W is implicitly known based on A_{sv}:
2 $W \leftarrow \begin{bmatrix} A_{vv} & 0 \\ A_{sv} & 0 \end{bmatrix}$
3 $X \leftarrow$ `SparseFactorization+Schur`(W)
4 $A_{ss} \leftarrow A_{ss} + X$ ▷ AXPY
  ▷ Reusing the factorized A_{vv} within W:
5 $b_v \leftarrow$ `SparseSolve`(A_{vv}, b_v)
6 $b_s \leftarrow b_s - A_{sv} b_v$
7 $A_{ss} \leftarrow$ `DenseFactorization`(A_{ss})
8 $x_s \leftarrow$ `DenseSolve`(A_{ss}, b_s)
9 $x_v \leftarrow$ `SparseSolve`$(A_{vv}, b_v - A_{sv}^T x_s)$

2.3 Limitations

We formulated the solution method of (1.2) in Section 1.6. The vanilla solver couplings introduced in Section 2.2 simply apply the sparse and the dense direct solver building blocks (see Section 2.1) to perform the necessary operations on the submatrices A_{vv}, A_{sv} and A_{ss} of the system. However, there are several major drawbacks in those vanilla couplings. In this section, we address them in details.

2.3.1 Baseline coupling

The baseline sparse/dense solver coupling (see Section 2.2.1) presents important limitations in solving larger FEM/BEM systems in terms of both performance and memory consumption.

In particular, we lose the ideal symmetry and sparsity condition of the factorized system (see Section 1.6.3) and the result Y of $(L_{vv}L_{vv}^T)^{-1}A_{sv}^T$, stored in an extra matrix, as well as the Schur complement itself are large dense matrices entirely stored in RAM.

2.3.2 Advanced coupling

Although the advanced solver coupling (see Section 2.2.2) is an optimal approach in terms of performance when it fits in memory, it implies storing the Schur complement matrix entirely in RAM in dense format, which restricts the range of problem sizes we can handle.

2.3.3 Discussion

The API of the sparse direct solver lead, in case of both baseline and advanced sparse/dense solver couplings, to the storage of potentially very large non compressed dense matrices. In terms of memory constraints, this quickly becomes a significant limiting factor in solving larger coupled systems. For example, considering the FEM/BEM system (counting 2,090,638 in the sparse part and 168,830 unknowns in the dense part) associated with the test case in Section 1.4.2, the sole storage of the Schur complement matrix would require around 212 GiB ($168,830^2 \times 16$ bytes per coefficient $\div\, 2 \div 1,024 \div 1,024 \div 1,024$) of RAM (considering complex matrices). In case of the baseline solver coupling, we would need 2.6 TiB ($2,090,638 \times 168,830 \times$ 16 bytes per coefficient $\div\, 2 \div 1,024 \div 1,024 \div 1,024$) of extra RAM to store the result Y of the *sparse solve* $(L_{vv}L_{vv}^T)^{-1}A_{sv}^T$ in a dense matrix.

Moreover, even if some direct solvers individually propose asynchronous execution of computations, it does not allow us to call the sparse and the dense solver an asynchronous fashion (see Section 2.6.2). Indeed, it is not possible to express the inter-solver dependencies on data and computations using the existing APIs of the sparse direct and dense direct solvers.

2.4 Related work

In the literature, we found similar approaches for solving coupled sparse and dense linear systems based on direct methods [61, 74, 129, 73, 116]. [61] addresses linear systems with both sparse and dense parts in the context of genomic prediction. According to the authors, such systems may have up to 100,000 unknowns associated with the dense part. However, they evaluate the proposed implementation on a smaller system with 1,279 unknowns in the dense part. [74] is related to soil-structure interaction problems. In this case, the author does not precise the target system size and presents performance evaluation results for systems with BEM-discretized part counting, to the best of our understanding, at most 1,536 boundary elements. In [129], the authors are interested in solution of linear systems arising from a coupled FEM/BEM formulation in the context of acoustic radiation problems. In the performance evaluation, they have processed systems with up to 3,017 unknowns in total. [73] is also set in the acoustic domain. In terms of problem size, the performance of the proposed implementation is evaluated on FEM systems with up to 982,912 degrees of freedom and up to 2,048 for BEM systems. [116] belongs to the domain of soil-structure interaction problems. The largest coupled test cases considered have 52,758 degrees of freedom.

To the best of our understanding, all of these approaches rely on a baseline usage (similar to the one discussed in Section 2.2.1) of the sparse direct solver, i.e., without using the Schur complement functionality. More importantly, the tackled problem size associated with the BEM part (yielding the dense block) is relatively small, which makes it possible to handle it with one

of the above schemes (see sections 2.2.1 and 2.2.2). In particular, no numerical compression nor out-of-core computation are employed. On the contrary, due to their dimension (especially the size of the dense block), the problems we tackle cannot be processed with the baseline and advanced solver coupling approaches.

2.5 Positioning of the thesis

Throughout the current chapter, we drew a state of the art of different capabilities of sparse and dense direct solvers, introduced the possible ways of using them to design a coupled sparse/dense solver and discussed their limitations in relation to the nature and the expected size of the system to solve (see Section 1.3). In this thesis, we therefore propose new classes of algorithms, namely two-stage and single-stage algorithms, to cope with the limitations exposed in Section 2.3 preventing us from solving large coupled sparse/dense FEM/BEM linear systems using a direct method (see Section 1.5.1).

In the first place, in chapters 3, 4, 5 and 6, we choose to rely on existing API of sparse and dense direct solvers. The idea is to work on carefully chosen submatrices of the coefficient matrix A in (1.2) so as to reduce the negative impact of the limitations in vanilla solver couplings. In this context, we propose two algorithms, multi-solve and multi-factorization, based on a block-wise computation of the Schur complement matrix S. This work builds upon an existing codebase from Airbus. In the thesis, the goal is to consolidate the arising algorithms and maximize their performance. The choice to use the existing API as-is gives us the possibility to rely on some well-optimized fully-featured solvers developed by the community implementing numerical compression, out-of-core computation and distributed-memory parallelism. The algorithms are then designed to take advantage of these advanced features in all the parts of the linear system, including S, unlike in the case of vanilla couplings (see Section 2.3). However, the same design choice prevents us from fully achieving the ideal implementation seen in Section 1.6.3. Indeed, the current API of the direct solvers was not designed with this application in mind in particular. Even if multi-solve and multi-factorization allow for bypassing the vanilla couplings limitations in terms of usage of advanced solver features, they do not eliminate them completely. As explained in Section 2.3.3, our design choice also implies synchronous calls to the sparse and the dense solver, i.e. in two separate stages. Therefore, we commonly refer to multi-solve and multi-factorization as to two-stage algorithms.

The objective of the more exploratory chapter 7 is to get on the track of the ideal implementation (see Section 1.6.3) through an alternative solver coupling relying on selected task-based direct solvers built on top of the same runtime. In this context, we explore the possibilities of adapting the API of the direct solvers. In the first place, the goal is to achieve the ideal symmetry and sparsity condition of the system during the solution process. In the second place, making the computation asynchronous (see Section 2.6.2) would allow for pipelining the calls to the sparse and the dense solver. This way, we do not have to wait for the Schur complement to be entirely assembled prior to beginning its factorization and the subsequent solve operations (see Section 1.6). Therefore, we refer to this design pattern as to single-stage algorithm. In terms of the ideal symmetry and sparsity condition of the system, it represents the best option with respect to the two-stage algorithms. However, implementing this approach into fully-featured direct solvers can only be a long-term work due to the complexity of the sparse solver's codebase (up to hundred thousands lines of code) in particular. In the short term, a prototype of such implementation can be realized within a non fully-featured framework, in our case, by sacrificing numerical compression and distributed memory parallelism.

Throughout this thesis, we put a strong emphasis on ensuring the reproducibility of our work. Chapter 8 explains how we make use of a functional transactional package manager allowing

for a complete reproducibility of software environments across different machines as well as the principles of literate programming. This paradigm combines formatted text with source code. In terms of experimental studies, it allows us to surround our algorithms and experiments with the related explanations and experimental analysis. In terms of experimental software environments, it allows us to associate the source code and the description of its purpose.

2.6 Selected sparse and dense direct solvers

In Section 2.1, we introduced the building blocks of the sparse and dense direct solvers we can make use of to design a coupled solver for (1.2). Then, in Section 2.2, we presented vanilla solver couplings based on these building blocks and analyzed their limitations preventing us from processing large coupled linear systems. Finally, in Section 2.5, we announced the research directions we explore within this thesis to cope with the limitations in the state-of-the-art approaches. Here, we discuss different features and types of API that sparse and dense direct solvers may provide as well as different programming models they can build on. Based on these properties, we then present the direct solvers we choose to rely on in this thesis.

2.6.1 Feature range

Fully-featured direct solvers may provide numerical compression, out-of-core computation or distributed-memory parallelism. Using numerical compression we can, for instance, instrument the solver to provide a solution approximation with lower accuracy (while remaining on an acceptable level) in exchange for an important performance improvement (see Section 1.5.2.3).

Thanks to out-of-core techniques, we can lower the memory footprint of the computation by moving the portions of data that are not being currently used from RAM (Random Access Memory) to disk. The RAM being also referred to as the *core memory*, the out-of-core qualifier refers to the temporary storage of data by the algorithms out of the core memory, i.e. on disk. On the contrary, we say of the algorithms that operate with data in RAM that they are *in-core* algorithms.

Once we reach the limits of a given workstation, we can distribute data and workload across multiple workstations, e.g. within a high-performance computing cluster.

2.6.2 API type

Solvers usually implement a synchronous API where all function calls are blocking. However, some solvers provide also an asynchronous API where the function calls only trigger the corresponding operation but return to the calling program as soon as possible. This allows for a concurrent execution of multiple operations working on different data or for pipelining of the operations sharing a data dependency.

2.6.3 Parallel programming model

Regardless the features they provide and their API type, solvers may follow different parallel programming models. In our context, we consider two categories of solvers based on this criterion. On the one hand, solvers may handle the aspects of parallel execution, such as partitioning and scheduling computations or transferring data to and from processing units, on

their own. On the other hand, it is possible to achieve a higher level of abstraction in parallel programming by delegating these functions to a runtime. In this case, the solver submits computation tasks and associated data dependencies to the runtime. The latter takes care of scheduling and executing the tasks on available (possibly heterogeneous) computing resources and ensures related data transfers. Runtimes may also implement features such as out-of-core computation the solver can implicitly benefit from as the data transfers are handled by the runtime and not by the solver itself. Using runtime-based solvers can be further beneficial for designing a coupled solver. Indeed, if two solvers resort to the same runtime, it favors the sharing of data, relevant dependencies as well as asynchronous execution of tasks between the two solvers. This composability is likely to be much more complex to achieve within the first category of solvers.

2.6.4 Selected direct solvers

Within this thesis, we choose different sparse and dense direct solvers to implement our algorithms depending on their feature range, type of API and programming model. For two-stage algorithms (introduced in Section 2.5), we rely on fully-featured direct solvers. In terms of functionalities, we are looking for parallel processing of multiple sparse right-hand sides and the *sparse factorization+Schur* building block (see Section 2.1.1.2) in the sparse solver and more generally for numerical compression, out-of-core computation and distributed-memory parallelism in both the sparse and the dense solvers. For instance, the sparse solver MUMPS [38, 36] meets the aforementioned criteria. Regarding the dense solvers, we can consider HMAT [101].

For single-stage algorithm (also introduced in Section 2.5), we rely on task-based sparse and dense direct solvers sharing the same runtime. These are the key criteria that should be met despite renouncing to some of the advanced features (see Section 2.6.1). Here, `qr_mumps` [20] and PaStiX [96] are suitable candidates for the sparse solver and HMAT for the dense solver.

2.6.4.1 Sparse solvers

Regarding sparse solvers, we rely on MUMPS [38] for the implementation of two-stage algorithms and we choose `qr_mumps` [20] for the implementation of single-stage algorithm. This choice is motivated by the fact that both MUMPS and `qr_mumps` implement the same approach for solving linear systems, the multifrontal method [67, 68, 99]. From the experimental point of view, this allows for a more insightful comparison between the implementations of two-stage and single-stage algorithms.

MUMPS relies on LU or LDL^T factorization and provides also Schur complement (see Section 2.1.1.2) computation routines. It features both distributed and thread-level parallelism relying on MPI (Message-Passing Interface) [104] and OpenMP (Open Multi-Processing) [13], respectively. Moreover, the solver implements out-of-core computation as well as numerical compression through a single-level $\mathcal{H}$-matrix scheme referred to as Block Low-Rank compression (BLR) [36].

`qr_mumps` implements QR and Cholesky factorization algorithms. It is a task-based solver relying on the StarPU runtime [44]. Contrary to MUMPS, `qr_mumps` does not aim at implementing numerical compression. Parallel execution of tasks and out-of-core computation [28] are delegated to the runtime. Note that `qr_mumps` and MUMPS are two independent software projects. The name of `qr_mumps` was inspired by the close relationship between the development teams of the two solvers.

2.6.4.2 Dense solvers

We consider two dense direct solvers, SPIDO [100, 34] and HMAT [11, 101]. Both solvers implement out-of-core computation and distributed-memory parallelism. However, SPIDO is a non-compressed solver and HMAT a compressed solver. Using also a non-compressed dense solver allows for the evaluation of the impact of numerical compression in the dense part of the target coupled linear system (1.2).

SPIDO is a proprietary dense direct solver from Airbus. To solve linear systems, it relies either on LU or LDL^T factorization. When the linear system is too large to fit in memory, SPIDO can perform computations out-of-core. It splits the coefficient matrix evenly into multiple submatrices called blocks. The currently unused blocks are temporarily stored on disk to reduce memory consumption. When an out-of-core block is loaded into memory for computation, it is further divided into smaller parts called processor blocks that may be distributed and processed in parallel using MPI (see Figure 2.3). Also, an additional thread-level parallelism relying on OpenMP can be enabled to potentially further speed-up the computation. Block sizes may be either set manually or determined automatically by the solver itself based on the size of the problem, the count of available computation nodes and threads as well as the amount of available memory.

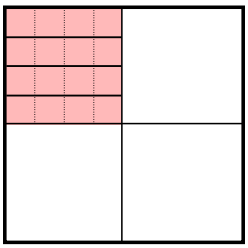


Figure 2.3: A coefficient matrix split into 4 out-of-core blocks. Currently processed block (in red) is divided into 4 processor blocks counting 4 thread blocks each.

HMAT is another dense direct solver developed and maintained at Airbus. Like SPIDO, it is a proprietary solver. However, an open-source sequential implementation of HMAT is available as HMAT-OSS [11]. The solver implements the hierarchical low-rank ($\mathcal{H}$-matrix) structure (see Section 1.5.1.1) and associated hierarchical variants of LU, LL^T and LDL^T factorization and solve algorithms. So as `qr_mumps`, it is based on the StarPU runtime. Out-of-core computations and parallel execution of tasks in HMAT are delegated to the runtime. Unlike other dense direct solvers, HMAT handles sparse matrices too [1]. Although, this is not one of the goals of this thesis, a preliminary experimental study in Section A.1 in Appendix evaluates HMAT for the processing of sparse matrices.

[1] To reduce fill-in (see Section 1.5.1.2) resulting from factorization, HMAT provides an implementation prototype of the Nested Dissection algorithm [75]. Improvements has been made in this direction [71], although further work is required to include these into the mainstream version of HMAT.

Chapter 3

Two-stage algorithms in shared memory

In Chapter 2, we discussed sparse and dense direct solvers, different features and building blocks they provide as well as how to possibly compose them to implement a sparse/dense solver for coupled FEM/BEM linear systems such as (1.2) defined in Section 1.3. We also exposed the limitations of vanilla couplings (see Section 2.2) of the state-of-the-art solvers preventing us from fully taking advantage of some of their advanced features such as numerical compression, out-of-core computation and asynchronous execution of the sparse and the dense solver. Finally, we presented the two main research directions we explore in this work, namely the two-stage and single-stage classes of algorithms. They are based on different strategies but both of them try to cope with the limitations related to vanilla solver couplings. The goal is to approach as much as possible the ideal implementation of a coupled sparse/dense FEM/BEM solution presented in Section 1.6.3.

In this chapter, we present two-stage algorithms (discussed in Section 2.5), namely multi-solve and multi-factorization, adopting the approach of exploiting the existing API of sparse and dense direct solvers on well chosen submatrices of the target linear system (1.2). The main strength of these algorithms is to build upon fully-featured direct solvers developed and optimized by the community. These solvers implement features we want to benefit from in order to process larger problems in a shared-memory environment, i.e. numerical compression and out-of-core computation. While it is possible to transparently activate numerical compression and out-of-core computation within individual building blocks, in all cases, the Schur complement S (see Section 1.6) is retrieved as non-compressed dense matrix entirely stored in RAM. The core idea of both multi-solve and multi-factorization is thus to allow for exploiting numerical compression and out-of-core computation also within the Schur complement part. This is achieved through a block-wise computation of S using a sparse direct solver which represents the first stage of the solution. Once S is entirely assembled, a dense direct solver takes over for the dense operations on S in the second stage. Multi-solve is a variation of the baseline coupling (see Section 2.2.1) while multi-factorization is a variation of the advanced coupling (see Section 2.2.2). In this chapter, we introduce only the usage of numerical compression. We present an alternative and complementary approach relying on out-of-core computation in Chapter 4.

For each one of the algorithms, we propose two variants: a baseline version (see sections 3.1.1 and 3.2.1) and an extension ensuring compression of the Schur complement matrix S (see sections 3.1.2 and 3.2.2). The baseline multi-solve and multi-factorization represent the starting

point for their compressed Schur counterparts and serve us in the experimental study (see Section 3.3) to assess the intrinsic overhead of the proposed multi-stage algorithms with respect to the vanilla baseline and advanced solver couplings from sections 2.2.1 and 2.2.2. The purpose of the compressed Schur variants (sections 3.1.2 and 3.2.2) is to build on top of the blocking scheme of their baseline counterparts (from sections 3.1.1 and 3.2.1, respectively) to compress the dense Schur blocks successively retrieved in order to limit the memory consumption so as to process larger problems. In Section 4.3, we rely on academic test cases to analyze the impact of numerical compression on the time to solution and memory consumption in shared memory. Experiments on an industrial application are discussed in Section 4.4. We conclude in Section 4.5.

3.1 Multi-solve algorithm

In this approach, we build on the baseline solver coupling presented in Section 2.2.1. However, instead of performing the *sparse solve* step $(L_{vv}L_{vv}^T)^{-1}A_{sv}^T$ using the entire submatrix A_{sv}^T, we split the latter in blocks of n_c columns $A_{sv_i}^T$ and perform successive parallel *sparse solve* operations. This leads to a block-wise assembly of the Schur complement matrix S by blocks of columns S_i. Given n_{BEM}, the number of unknowns associated with the formulation of BEM, and n_c, the number of columns of A_{sv}^T in one block $A_{sv_i}^T$, there are n_{BEM}/n_c blocks in total (see Figure 3.1). We denote $A_{sv_i}^T$ and A_{ss_i} the i$^{\text{th}}$ blocks of n_c columns of A_{sv}^T and A_{ss}, respectively. Then, based on the definition of S in (1.8), S_i is a block of n_c columns of S defined as:

$$S_i = A_{ss_i} - \underbrace{A_{sv}\overbrace{(L_{vv}L_{vv}^T)^{-1}A_{sv_i}^T}^{Y_i}}_{Z_i}. \tag{3.1}$$

3.1.1 Baseline algorithm

The *baseline multi-solve* algorithm (see Algorithm 3) begins by the *sparse factorization* of A_{vv} (line 3) into $L_{vv}L_{vv}^T$. Then, we loop (line 4) over the blocks $A_{sv_i}^T$ of A_{sv}^T and the blocks A_{ss_i} of A_{ss} to compute all the blocks Z_i such as defined in (3.1). The first step of this computation (line 4) is a *sparse solve* for determining the block $Y_i = (L_{vv}L_{vv}^T)^{-1}A_{sv_i}^T$. Fully-featured sparse direct solvers additionally allow us to benefit from the sparsity of the right-hand side matrix $A_{sv_i}^T$ during the solve operation [35]. However, independently from the sparsity of the input right-hand side, the resulting Y_i is a **dense matrix** [56]. Finally, we have to temporarily store both the $A_{sv_i}^T$ and Y_i blocks explicitly (see Figure 3.1).

3.1.2 Compressed Schur variant

Both the sparse and the dense direct solver implement numerical compression within their respective building blocks (see Section 2.6). In the *compressed Schur multi-solve* variant (see Algorithm 4), we can thus transparently apply it to the sparse submatrices A_{vv} and A_{sv} as well as to the dense submatrix A_{ss}. However, the Y_i block is still retrieved dense and non-compressed so as the Z_i block (see Figure 3.2). Therefore, we have to transform Z_i into a temporary compressed matrix (line 8) before performing the final operation of the computation of the associated Schur complement block S_i (line 9), i.e. $A_{ss_i} - Z_i$ (see Figure 3.2). Although A_{ss_i} is initially compressed, this operation implies a re-compression of the block at each iteration of the loop on i.

Algorithm 3: *baseline multi-solve* algorithm for computing S based on (3.1) and solving (1.2).

Function `BaselineMultiSolve`(A, b)**:**
  $A_{vv} \leftarrow$ `SparseFactorization`(A_{vv})
  for $i = 1$ **to** n_{BEM}/n_c **do**
    ▷ Using the i^{th} block of columns of A_{sv}^T as right-hand side:
    $Y_i \leftarrow$ `SparseSolve`$(A_{vv}, A_{sv_i}^T)$
    $Z_i \leftarrow A_{sv} \times Y_i$ ▷ SpMM
    $A_{ss_i} \leftarrow A_{ss_i} - Z_i$ ▷ AXPY
  $b_v \leftarrow$ `SparseSolve`(A_{vv}, b_v)
  $A_{ss} \leftarrow$ `DenseFactorization`(A_{ss})
  $x_s \leftarrow$ `DenseSolve`$(A_{ss}, b_s - A_{sv} b_v)$
  $x_v \leftarrow$ `SparseSolve`$(A_{vv}, b_v - A_{sv}^T x_s)$

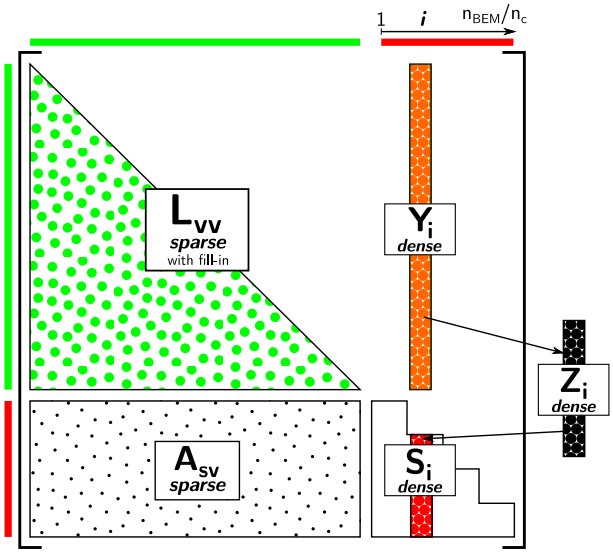


Figure 3.1: Computation loop of S in *baseline multi-solve* (see Algorithm 3). $A_{sv_i}^T$ is explicitly stored and Y_i is a **dense matrix**.

Algorithm 4: *compressed Schur multi-solve* variant for computing S based on (3.1) and solving (1.2).

```
1  Function CompressedSchurMultiSolve(A, b):
2      A_vv ← SparseFactorization(A_vv)
3      for i = 1 to n_BEM/n_S do
4          for j = 1 to n_S/n_c do
               ▷ Using the ij^th block of columns of A_sv^T as right-hand side:
5              Y_ij ← SparseSolve(A_vv, A_sv_ij^T)
6              Z_ij ← A_sv × Y_ij                                  ▷ SpMM
7              Z_i ← Concatenate(Z_i, Z_ij)
8          Compress(Z_i)
9          S_i ← A_ss_i − Z_i                                      ▷ Compressed AXPY
10     b_v ← SparseSolve(A_vv, b_v)
11     A_ss ← DenseFactorization(A_ss)                             ▷ Compressed factorization
12     x_s ← DenseSolve(A_ss, b_s − A_sv b_v)                      ▷ Compressed solve
13     x_v ← SparseSolve(A_vv, b_v − A_sv^T x_s)
```

Moreover, in the *compressed Schur multi-solve* algorithm, we dissociate the parameter n_c, handling the size of blocks $A^T_{sv_i}$ of A^T_{sv} and consequently the size of Y_i and Z_i, from the parameter n_S handling the size of the Schur complement blocks S_i. The reason for this separation is the overhead associated with the transformation of the blocks Z_i into compressed matrices as well as the computation of $A_{ss_i} - Z_i$ (line 9) which implies a re-compression of A_{ss_i}. With the separate parameter n_S, we can use larger blocks Z_i to minimize additional computational cost due to frequent matrix compressions and keep smaller blocks $A^T_{sv_i}$ and Y_i preventing an excessive rise of memory consumption. We discuss this further in Section 3.3.4.1.

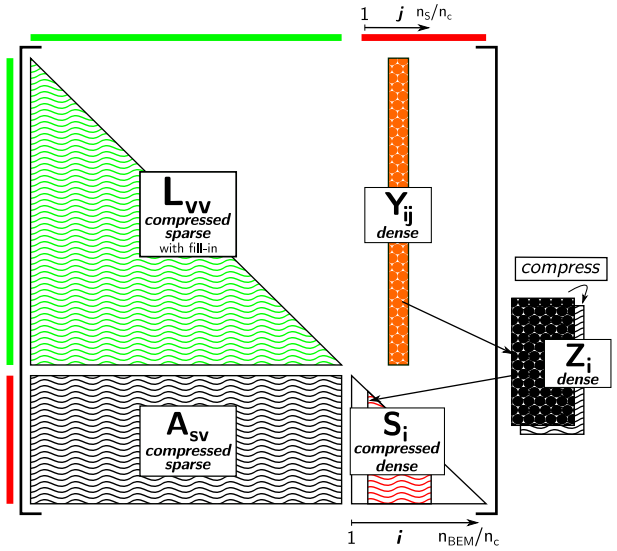


Figure 3.2: Computation loop of S in *compressed Schur multi-solve* (see Algorithm 4). $A^T_{sv_i}$ is explicitly stored and Y_{ij} is a **dense matrix**.

3.2 Multi-factorization algorithm

The multi-factorization method is based on the advanced sparse/dense solver coupling presented in Section 2.2.2. Nevertheless, instead of trying to compute the entire Schur complement using a single call to the *sparse factorization+Schur* step, we split A_{sv} and A_{sv}^T submatrices into n_b blocks A_{sv_i} and $A_{sv_j}^T$, respectively. The goal is to call the *sparse factorization+Schur* step on smaller submatrices composed of A_{vv}, A_{sv_i} and $A_{sv_j}^T$ and compute the Schur complement S by square blocks S_{ij} of equal size n_{BEM}/n_b (see Figure 3.3). We denote A_{sv_i} a block of n_{BEM}/n_b rows of A_{sv} and $A_{sv_j}^T$ a block of n_{BEM}/n_b columns of A_{sv}^T. Then, based on the definition of S in (1.8), S_{ij} is a block of n_{BEM}/n_b rows and columns of S such as:

$$S_{ij} = A_{ss_{ij}} \overbrace{-A_{sv_i} A_{vv}^{-1} A_{sv_j}^T}^{X_{ij}} . \tag{3.2}$$

3.2.1 Baseline algorithm

The computation of the Schur complement S in the *baseline multi-factorization* algorithm (see Algorithm 5) is performed within the main loop on line 2. In this loop, we construct a temporary submatrix W from A_{vv}, A_{sv_i} and $A_{sv_j}^T$:

$$W \leftarrow \begin{bmatrix} A_{vv} & A_{sv_j}^T \\ A_{sv_i} & 0 \end{bmatrix}.$$

However, when $i = j$, W is composed of:

$$W \leftarrow \begin{bmatrix} A_{vv} & 0 \\ A_{sv_i} & 0 \end{bmatrix}.$$

Then, we call the *sparse factorization+Schur* step on W (line 8) relying on the Schur complement feature provided by the sparse direct solver (see Section 2.1.1). When $i \neq j$, we can not rely on a symmetric mode of the direct solver and have to enter both the lower and upper parts of A_{vv} (see Figure 3.3) as well as $A_{sv_j}^T$ into W. The *sparse factorization+Schur* step then returns the Schur complement block $X_{ij} = -A_{sv_i}(L_{vv}U_{vv})^{-1}A_{sv_j}^T$ associated with the submatrix W. When $i = j$, W do not have to contain the upper part of A_{vv} nor $A_{sv_j}^T$ which is implicitly known from A_{sv_i}. In this case, we can rely on a symmetric mode of the direct solver and the *sparse factorization+Schur* step returns $X_{ij} = -A_{sv_i}(L_{vv}L_{vv}^T)^{-1}A_{sv_j}^T$. To determine the block S_{ij} of the Schur complement S, we have to compute $A_{ss_{ij}} + X_{ij}$ (line 9) following (3.2).

The need for the temporary submatrix W leads to **duplicated storage**. This comes from two sources. One, a block of rows A_{sv_i} of A_{sv} is copied into W. When $i \neq j$, we also have to count a block of columns $A_{sv_j}^T$ of A_{sv}^T. Two, W explicitly stores the copy of the lower-triangular part of A_{vv} in its upper-triangular part when $i \neq j$. To prevent the storage of two copies of A_{vv}, the original is stored on disk.

The *sparse factorization+Schur* step involving W implies a re-factorization of A_{vv} in W at each iteration although it does not change during the computation, hence the name of the method - multi-factorization. The API of the sparse direct solver does not allow us to reuse the result of the first factorization in the subsequent calls to *sparse factorization+Schur*. As a result, the more blocks A_{ss} is split into, the more superfluous factorizations of A_{vv} are performed. We discuss this further in Section 3.3.4.2.

Algorithm 5: *baseline multi-factorization* algorithm for computing S based on (3.2) and solving (1.2).

Function `BaselineMultiFactorization`(A, b)**:**
  for $i = 1$ **to** n_b **do**
    for $j = 1$ **to** i **do**
      if $i = j$ **then**
        $W \leftarrow \begin{bmatrix} A_{vv} & 0 \\ A_{sv_i} & 0 \end{bmatrix}$
      else
        $W \leftarrow \begin{bmatrix} A_{vv} & A_{sv_j}^T \\ A_{sv_i} & 0 \end{bmatrix}$
      $X_{ij} \leftarrow$ `SparseFactorization+Schur`(W)
      $A_{ss_{ij}} \leftarrow A_{ss_{ij}} + X_{ij}$ ▷ AXPY
  $A_{vv} \leftarrow$ `SparseFactorization`(A_{vv})
  $b_v \leftarrow$ `SparseSolve`(A_{vv}, b_v)
  $A_{ss} \leftarrow$ `DenseFactorization`(A_{ss})
  $x_s \leftarrow$ `DenseSolve`$(A_{ss}, b_s - A_{sv} b_v)$
  $x_v \leftarrow$ `SparseSolve`$(A_{vv}, b_v - A_{sv}^T x_s)$

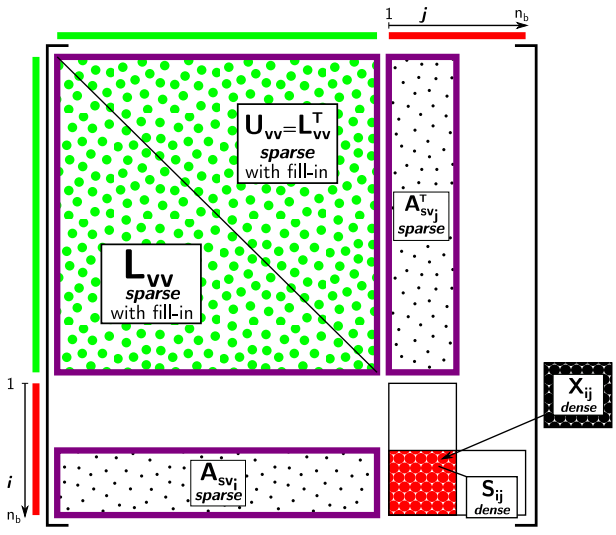


Figure 3.3: Computation loop of S in *baseline multi-factorization* (see Algorithm 5). Constructing W requires a temporary **duplicated storage** of A_{vv} (upper part), A_{sv_i} and optionally $A_{sv_j}^T$ (when $i \neq j$).

3.2.2 Compressed Schur variant

As in the multi-solve algorithm, we rely on the sparse direct solver to process A_{vv}, A_{sv} and on the dense direct solver to process A_{ss} in a compressed fashion out-of-the-box for us. Nevertheless, the Schur complement itself remains returned as a non-compressed dense matrix (see Section 2.3). In the *compressed Schur multi-factorization* variant (see Algorithm 6), we thus compress the X_{ij} Schur block into a temporary compressed matrix as soon as the sparse solver returns it (line 9).

Algorithm 6: *compressed Schur multi-factorization* algorithm for computing S based on (3.2) and solving (1.2).

```
1  Function CompressedSchurMultiFactorization(A, b):
2      for i = 1 to n_b do
3          for j = 1 to i do
4              if i = j then
5                  W ← [ A_vv      0 ]
                       [ A_sv_i    0 ]
6              else
7                  W ← [ A_vv      A_sv_j^T ]
                       [ A_sv_i    0        ]
8              X_ij ← SparseFactorization+Schur(W)
9              Compress(X_ij)
10             A_ss_ij ← A_ss_ij + X_ij                    ▷ Compressed AXPY
11     A_vv ← SparseFactorization(A_vv)
12     b_v ← SparseSolve(A_vv, b_v)
13     A_ss ← DenseFactorization(A_ss)                     ▷ Compressed factorization
14     x_s ← DenseSolve(A_ss, b_s − A_sv b_v)              ▷ Compressed solve
15     x_v ← SparseSolve(A_vv, b_v − A_sv^T x_s)
```

The corresponding fully assembled S_{ij} block can then be computed using both the compressed X_{ij} and $A_{ss_{ij}}$ (line 10). Like in the case of *compressed Schur multi-solve* (see Section 3.1.2), this operation implies a re-compression of the initially compressed $A_{ss_{ij}}$.

3.3 Experimental results

We have evaluated the previously discussed algorithms allowing for efficient low-rank compression schemes for solving coupled sparse/dense FEM/BEM linear systems such as defined in (1.2). For the purpose of this evaluation, we used the *wide pipe* test case (see Section 1.4.1). The test case is designed so that we can compute the relative error of the numerical solution (see Section 1.5.2.1).

We have conducted our experiments on a single `miriel` node on the PlaFRIM platform [19]. A `miriel` node has a total of 24 processor cores running each at 2.5 GHz and 128 GiB of RAM. It is the policy of the platform to deactivate Hyper-Threading and Turbo-Boost in order to improve the reproducibility of the experiments. The solver test suite is compiled with GNU C Compiler (gcc) 9.3.0, Intel(R) MKL library 2019.1.144, and MUMPS 5.2.1. Each run presented below uses one node, with one process and 24 threads.

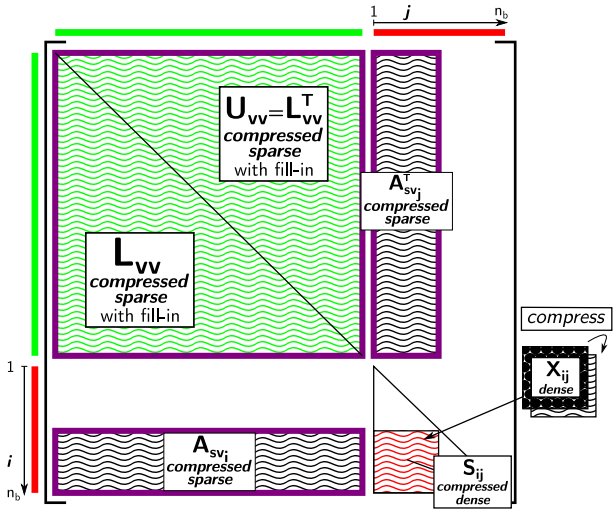


Figure 3.4: Computation loop of S in *compressed Schur multi-factorization* (see Algorithm 6). Constructing W requires a temporary **duplicated storage** of A_{vv} (upper part), A_{sv_i} and optionally $A^T_{sv_j}$ (when $i \neq j$).

3.3.1 Implementation and configuration

We resort to the selected sparse and dense direct solvers introduced in Section 2.6.4 to implement the proposed two-stage algorithms. For comparison with the state of the art, we consider also the advanced vanilla solver coupling defined in Section 2.2.2.

3.3.1.1 Two-stage algorithms

We have implemented the multi-solve and multi-factorization algorithms (see sections 3.1 and 3.2) on top of the coupling of the sparse direct solver MUMPS (see Section 2.6.4.1) with either the proprietary ScaLAPACK-like [48] dense direct solver SPIDO or the hierarchical low-rank $\mathcal{H}$-matrix compressed solver HMAT (see Section 2.6.4.2). SPIDO is used for the *baseline* variants from sections 3.1.1 and 3.2.1 and HMAT for the *compressed* variants from sections 3.1.2 and 3.2.2. In the rest of the experimental study, we thus refer to these *baseline* and *compressed* couplings as to MUMPS/SPIDO and MUMPS/HMAT, respectively. We use double precision accuracy. MUMPS and HMAT both provide low-rank compression and expose a precision parameter ϵ set to 10^{-3} (see Section 1.5.2.3). Because the activation of the compression within the sparse direct solver, MUMPS in this evaluation, is independent from the two-stage algorithms and is completely transparent from the coupling point of view, we systematically turn it on for both MUMPS/SPIDO and MUMPS/HMAT. For the tests in sections 3.3.3 and 3.3.4, all the matrices are stored in-core (see Section 2.6.1), i.e. the out-of-core feature of the sparse and dense solvers (see Section 2.6.4), when available, was not used.

3.3.1.2 Reference vanilla *advanced* coupling

From the algorithmic point of view, the advanced vanilla coupling (see Algorithm 2) is almost equivalent to the *baseline multi-factorization* (see Algorithm 5 in Section 3.2.1) with n_b set to 1, i.e. when the Schur complement matrix is not split into multiple blocks and processed at once. The only difference is the additional *sparse factorization* of A_{vv} on line 10 of the *baseline multi-factorization* algorithm. While it extends the overall time to solution, the peak in terms of RAM usage remains in the *sparse factorization+Schur* step (see experimental confirmation in

Section 6.2.2 of Chapter 6) being the core step of both of the algorithms.

To evaluate the performance of the advanced vanilla coupling, we thus rely on the implementation of the *baseline multi-factorization* algorithm on top of MUMPS and SPIDO as described in Section 3.3.1.1 and set the n_b parameter to 1. We use double precision accuracy, numerical compression in MUMPS is systematically turned on and ϵ set to 10^{-3} like in the case of two-stage algorithms.

3.3.1.3 Known limitations

There are two limitations in the implementation of the multi-factorization algorithm we rely on in this particular experimental study. One, it uses a non-symmetric mode of the sparse direct solver within the *sparse factorization+Schur* step even for diagonal Schur complement blocks, i.e. when $i = j$ in the loop on line 8 in algorithms 5 and 6. Note that this leads to a longer computation time but has no impact on the RAM usage peak. Two, in the version 5.2.1 of MUMPS used in this study, an integer overflow prevents from setting the Schur complement block size above 46,340 rows or columns [56]. In a particular case (discussed below), this prevents us from lowering the value of the n_b parameter as much as we could with respect to the available amount of RAM.

Note that these limitations appear only in the implementation used within this experimental study, i.e. within Section 3.3, and in Section 6.2 of Chapter 6. They do **not appear** within all the other experimental studies, i.e. in Section 3.4 of the present chapter as well as in chapters 4, 5 and 7.

3.3.2 Monitoring tools

Computation times appearing in this study are provided by our application which measures the execution time of all the routines. In figures, we report the 'Factorization time' which corresponds to the execution time of all the steps of the multi-solve and multi-factorization algorithms except for the computation of the solution vectors x_s and x_v (see the last two lines of algorithms 3, 4, 5 and 6).

To monitor the RAM usage we resort to an external Python script relying on the `/proc/[PID]/statm` system file (the resident size field). The acquisition frequency is set to 1 Hz, i.e. one measure per second.

3.3.3 Solving larger systems

In this section, the goal is to determine what are the largest systems that the multi-solve and multi-factorization algorithms allow us to process on the target compute node, and the associated computation times. We consider coupled FEM/BEM linear systems with the total number of unknowns N ranging from 1,000,000 up to exceeding the memory limit. Table 1.1 details the counts of FEM and BEM unknowns, namely n_{FEM} and n_{BEM}, for each value of N. In addition, we evaluate multiple configurations for each algorithm. Regarding the *baseline multi-solve* algorithm (see Section 3.1.1) relying on the MUMPS/SPIDO coupling, we vary the size n_c of blocks $A_{sv_i}^T$ of columns of the A_{sv}^T submatrix (see Figure 3.1) between 32 and 256. For the *compressed Schur multi-solve* (see Section 3.1.2), relying on the MUMPS/HMAT coupling, the size of blocks of columns of S and A_{sv}^T is handled by two different parameters, n_S and n_c respectively (see Figure 3.2). In this case, we set n_c to a constant value of 256 columns (motivated by the results of the study in Section 3.3.4) and vary n_S in the range from 512 to 4,096. In the case of the multi-factorization algorithm, both the *baseline multi-factorization* (see Section

3.2.1) and the *compressed Schur multi-factorization* variants (see Section 3.2.2) expose the n_b parameter handling the count of square blocks S_{ij} per block row and block column of the Schur complement submatrix S (see figures 3.3 and 3.4). The tested values of n_b are between 1 and 10.

In Figure 3.5, for each solver coupling, we show the best computation times of both variants of the multi-solve and the multi-factorization algorithms among all of the evaluated configurations and problem sizes. The algorithm allowing us to process the largest coupled sparse/dense FEM/BEM system is the *compressed Schur multi-solve* variant for N as high as 9,000,000 unknowns in total. In the case of the MUMPS/SPIDO coupling, when S and A_{ss} are not compressed, we could reach 7,000,000 unknowns. In the multi-factorization case, the compression of S and A_{ss} did not allow us to lower the memory footprint enough for processing larger systems than what we could achieve without. Indeed, in both cases we could process systems with up to 2,500,000 unknowns which is a considerably smaller size compared to multi-solve. This is due, in particular, to the duplicated storage induced by the loss of symmetry in the multi-factorization method (see Section 3.2) combined with the relatively large ratio of FEM-related to BEM-related unknowns of the *wide pipe* test case (see Section 1.4.1). In Section 3.4, we will see that this differs in the industrial test case. Eventually, both the multi-solve and the multi-factorization methods make it possible to process significantly larger systems than the advanced vanilla coupling (see Section 2.2.2) employed in the state of the art. According to our experiments, the latter allowed us to process at most 1,300,000 unknowns in 455 seconds. However, in this case, we were not able to process larger systems due to the limitation on the Schur complement block size (see Section 3.3.1.3). A recent experiment using a newer version of MUMPS (5.5.1) which does not suffer from this limitation showed that the advanced vanilla coupling can process a coupled system with up to 1,700,000 unknowns in 957 seconds before exceeding the memory limit. In this experiment, we also relied on a symmetric mode of MUMPS within the *sparse factorization+Schur* step (see Section 3.3.1.3).

One may expect that the multi-solve method should always present better computation time than the multi-factorization method due to the superfluous re-factorizations of the A_{vv} submatrix in the latter. However, in Figure 3.5, we can see that multi-factorization may outperform multi-solve on smaller systems, here for N as high as 2,000,000. Indeed, unlike multi-solve, which relies on a baseline usage of the sparse direct solver (see Section 2.2.1), multi-factorization takes advantage of the efficiency of the Schur complement functionality of the sparse solver. On the other hand, multi-factorization implies duplicated storage leading to increased memory consumption and a lot of re-factorizations of A_{vv} when there is not enough memory with respect to the size of the problem. Here, with a fixed amount of available memory, when the problem is small enough, we can use large blocks S_{ij} of the Schur complement S and need only few re-factorizations, in which case the multi-factorization performs better than multi-solve. For larger problems, multi-factorization is more and more penalized and the multi-solve algorithm becomes the best performing one. We further study these trade-offs in Section 3.3.4. We can also observe that in the case of multi-solve, the computation time is better for the *baseline multi-solve* variant. However, this is not true for all the runs nor does it mean that the compression of A_{ss} and S has no effect or a negative impact on the efficiency of the algorithm. The computation time of the factorization of the Schur complement is lower for the MUMPS/HMAT coupling but the time spent by MUMPS to perform the *sparse solve* step $A_{vv}^{-1}A_{sv}^{T}$ is higher for MUMPS/HMAT than for MUMPS/SPIDO. Indeed, there is a non-negligible computation time variability from one benchmark to another, especially if the benchmarks are not performed on the very same `miriel` node. We further address this phenomenon in the multi-metric study of multi-solve and multi-factorization presented in Chapter 6 (see Section 6.2.4 in particular).

Eventually, Figure 3.6 shows the relative error for the test cases featured in Figure 3.5. The

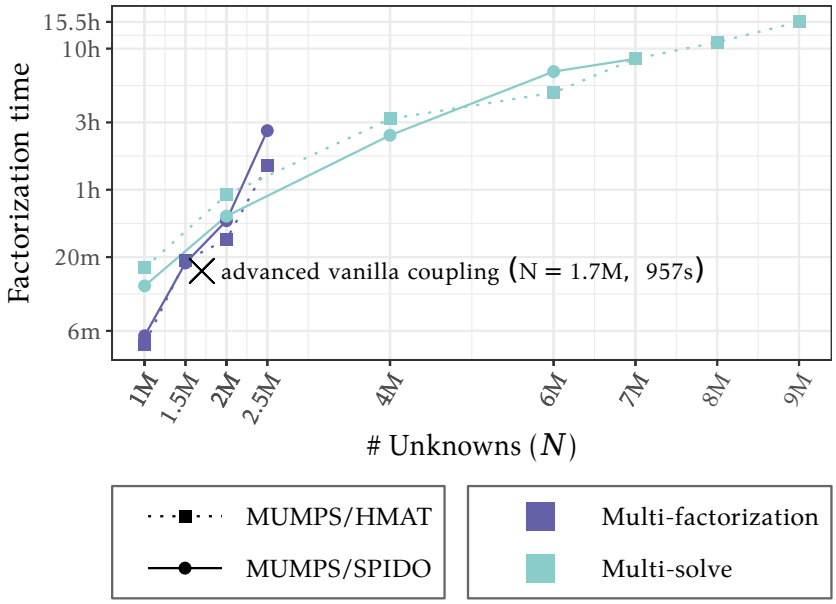


Figure 3.5: Best computation times of **multi-solve** and **multi-factorization** for both of the solver couplings MUMPS/HMAT and MUMPS/SPIDO. Parallel runs using 24 threads on single `miriel` node.

precision parameter ϵ was set to 10^{-3} for both MUMPS and HMAT solvers providing low-rank compression. Unlike for the fully-compressed test cases relying on the MUMPS/HMAT coupling, the relative error is smaller in the case of MUMPS/SPIDO when the dense part of the linear system is not compressed at all and thus the final result of the computation suffers less from the loss of accuracy due to the compression. It is to note that the low-rank compression in MUMPS was activated for both the MUMPS/SPIDO and the MUMPS/HMAT couplings. In all cases, the relative error is below the selected threshold 10^{-3} which confirms that the algorithm allows us to reach the expected accuracy.

3.3.4 Study of the performance-memory trade-off

We now further detail the trade-off between performance and memory consumption of the algorithms. We want to know how to run the algorithms efficiently with a given amount of memory. To a certain extent, this method may be related to the so-called memory-aware approaches [111, 26, 28, 102].

3.3.4.1 Multi-solve algorithm

We consider a coupled FEM/BEM linear system with N, the total unknown count, fixed to 2,000,000. Regarding the *baseline multi-solve* (see Section 3.1.1) relying on the MUMPS/SPIDO coupling, we vary the size n_c of block $A_{sv_i}^T$ of columns of the A_{sv}^T submatrix. For the *compressed Schur multi-solve* (see Section 3.1.2), using the MUMPS/HMAT solver coupling, the size of blocks of columns of S and A_{sv}^T is handled by the n_S and n_c parameters, respectively. In this case, we first set n_c equal to n_S varying from 32 to 256, then we maintain n_c at a constant value of 256 columns (motivated by the results presented further in this section) and vary n_S between 512 and 4,096. Note that the n_c parameter also handles the number of right-hand sides treated simultaneously by MUMPS during the *sparse solve* step $A_{vv}^{-1}A_{sv}^T$ within the Schur complement computation (see Section 1.6.2).

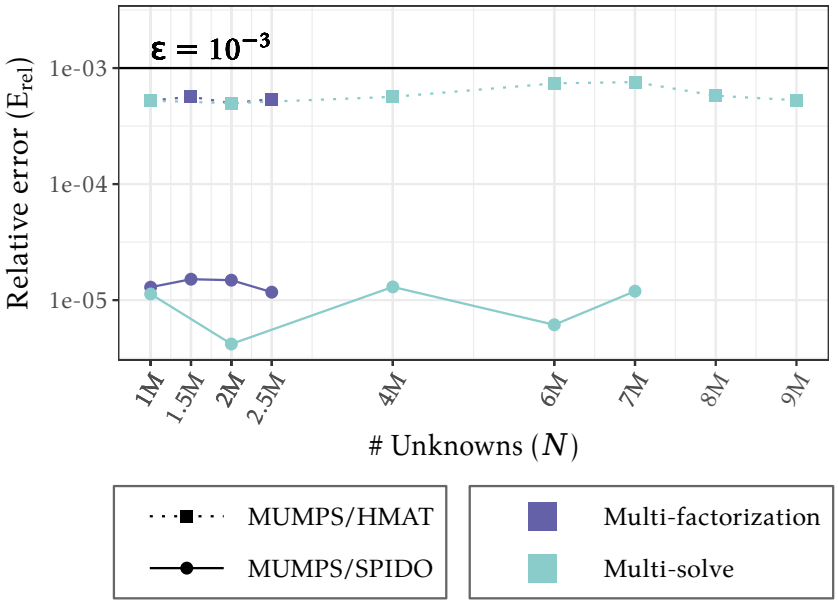


Figure 3.6: Relative error E_{rel} for the runs of **multi-solve** and **multi-factorization** having the best execution times and for both of the solver couplings MUMPS/HMAT and MUMPS/SPIDO. Parallel runs using 24 threads on single `miriel` node.

In the first place, we focus on the n_c parameter and its impact on the performance of MUMPS within the *baseline multi-solve* algorithm. According to Figure 3.7, setting n_c to a sufficiently high value, i.e. 256 in this case, can considerably improve the computation time. For higher values of n_c than 256, the performance improvement begins to be less significant compared to the rapidly increasing RAM usage due to the fact that the result of the *sparse solve* step $A_{vv}^{-1}A_{sv}^{T}$ is a dense matrix. Based on this result, we choose to set n_c to 256 in case of the *compressed Schur multi-solve* tests. In this compressed variant, if the Schur complement block is too small, it leads to too frequent matrix compressions and increases the computation time, hence the introduction of the separate parameter n_S for the size of Schur complement blocks. We can observe this phenomenon when n_S is too small, i.e. between 32 and 256 in this case. Just like for n_c, there is no need to increase n_S as much as possible. From a sufficiently high value, $n_S = 512$ in this case, n_S has only a little impact on the computation time of the *compressed Schur multi-solve* variant. Eventually, when we compare the *baseline multi-solve* to the *compressed Schur multi-solve*, we can observe that compressing the dense submatrices S and A_{ss} allows us to significantly decrease the memory consumption of the multi-solve algorithm.

3.3.4.2 Multi-factorization algorithm

We consider a coupled FEM/BEM linear systems with N, the total unknown count fixed to 1,000,000. Both the *baseline multi-factorization* (see Section 3.2.1) and the *compressed Schur multi-factorization* variants (see Section 3.2.2) expose the n_b parameter handling the count of square blocks S_{ij} per block row and block column of the Schur complement submatrix S. The tested values of n_b are between 1 and 4.

In Figure 3.8, we can observe the negative impact of the increasing number of superfluous re-factorizations of A_{vv} on the performance of the multi-factorization algorithm with the increasing number of Schur complement blocks S_{ij}. On the other hand, smaller Schur complement blocks allow one to reduce the memory footprint of the multi-factorization algorithm. While in multi-solve the main impact of the block size parameter n_c is on the parallel efficiency of the

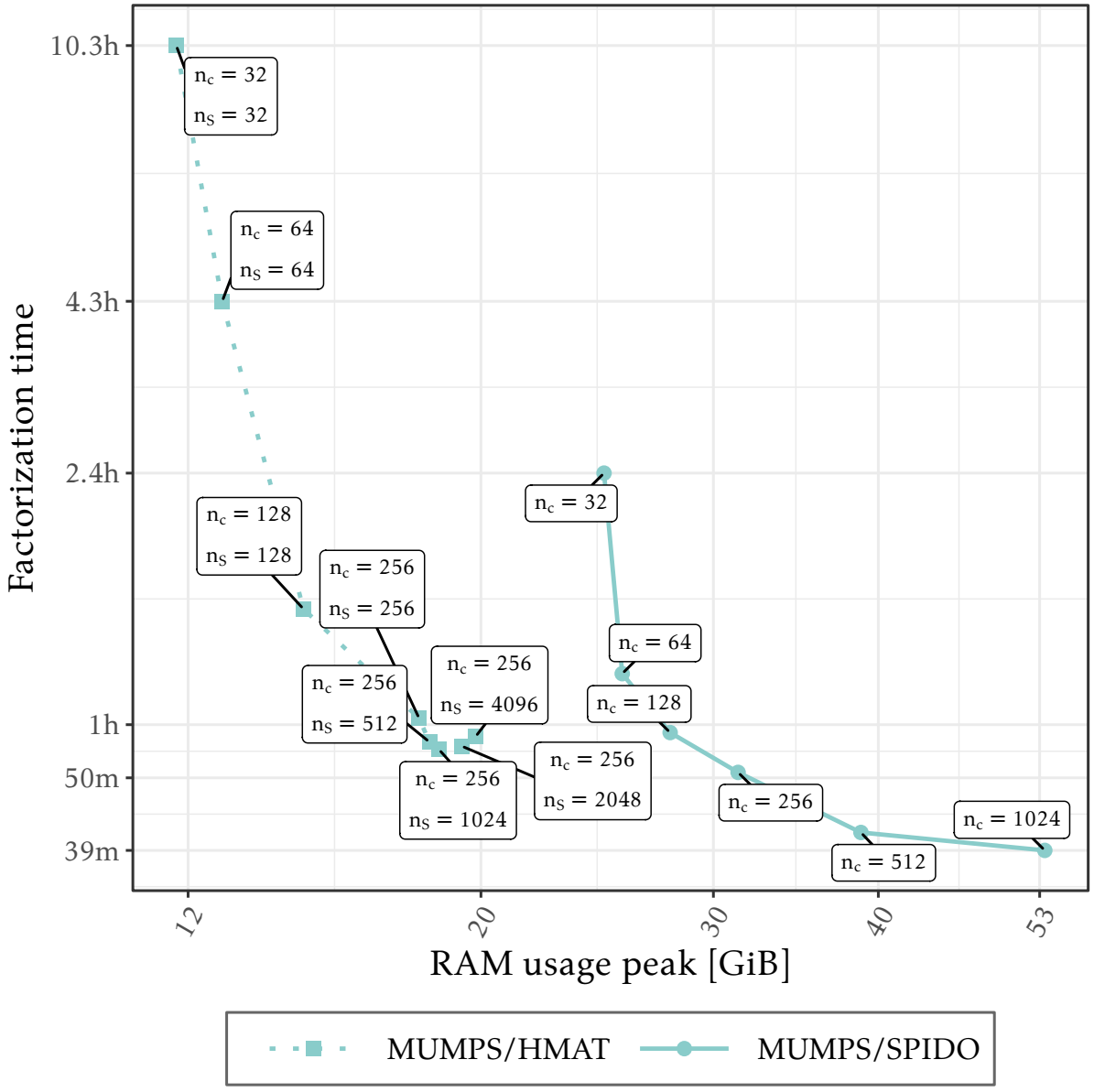


Figure 3.7: Comparison (in terms of RAM usage and performance) between the **multi-solve** implementations for the MUMPS/SPIDO and MUMPS/HMAT couplings on a coupled FEM/BEM system counting 2,000,000 unknowns for varying values of n_c and n_S.

successive *sparse solve* operations, in multi-factorization, using lower values of n_b we can reduce the number of factorizations to perform which is much more crucial and efficient. Application of low-rank compression techniques to the dense submatrix A_{ss} and the Schur complement submatrix S, further reduces the memory consumption of the algorithm. However, the gain is not as noticeable as for the multi-solve method. The duplicated storage due to the loss of symmetry in multi-factorization plus the dense storage of the Schur complement blocks by the sparse solver significantly increase the RAM usage of the application and counterbalances the positive effect of the low-rank compression in the dense part of the system.

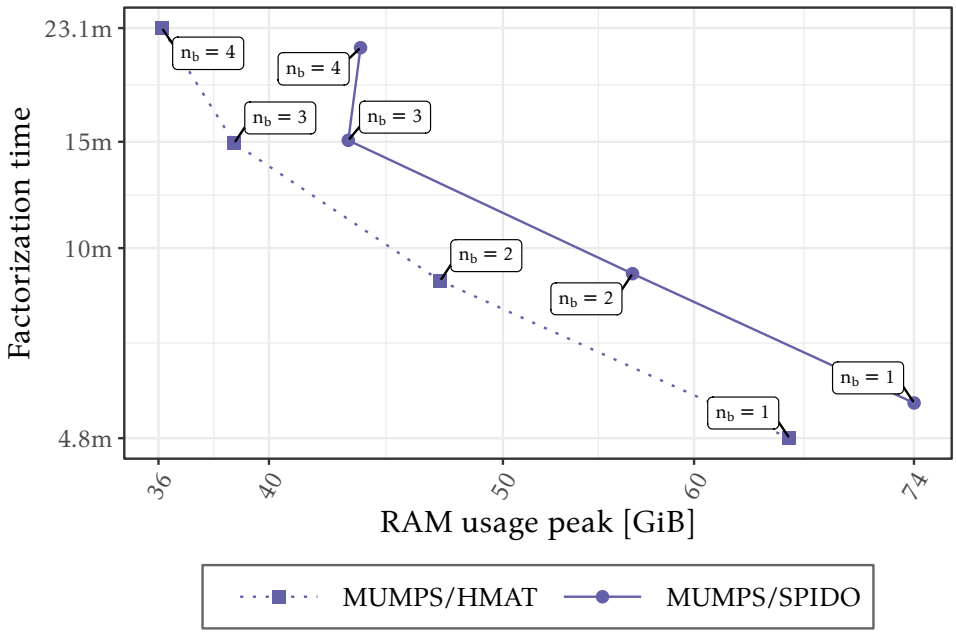


Figure 3.8: Comparison (in terms of RAM usage and performance) between the **multi-factorization** implementations for the MUMPS/SPIDO and MUMPS/HMAT couplings on a coupled FEM/BEM system counting 1,000,000 unknowns for varying values of n_b.

3.4 Industrial application

We present in this section an example of industrial application processed at Airbus R&T using the multi-solve and multi-factorization algorithms. In this case, we rely on the industrial test case from Section 1.4.2. Compared to the academic test cases from Section 1.4.1, it covers a larger surface domain leading to a higher proportion of BEM-related unknowns in the associated coupled linear system. Hence the relative cost of the (dense) BEM part will be more important and its compression has a bigger impact.

To run these tests, we use Airbus HPC5 computing facility. Each computing node has two Intel(R) Xeon(R) Gold 6142 CPU at 2.60GHz, for a total of 32 cores (Hyper-Threading and Turbo-Boost are deactivated) and 384 GiB of RAM. The acoustic application `Actipole` is compiled with Intel(R) 2016.4 compilers and libraries, and MUMPS 5.4.1. Each run presented below uses one node, with one process and 32 threads. For these tests, all the matrices are stored in memory, i.e. the out-of-core feature of the sparse and dense solvers (see Section 2.6.1), when available, was not used. We use simple precision accuracy and, for compressed solvers, the precision parameter ϵ is set to 10^{-4}. Measurements of computation time and RAM usage are done as described in Section 3.3.2.

Table 3.1 presents the results obtained on this test case using different approaches. For ref-

	Algorithm	Dense compressed	Sparse compressed	n_b	RAM (GiB)	Time (s)
1	state-of-the-art (§2.2.2)			N/A	OOM	-
2	multi-solve (§3.1.1)			N/A	249	18h
3	multi-facto. (§3.2.1)			12	OOM	-
4	multi-solve (§3.1.1)		x	N/A	224	**15.6h**
5	multi-facto. (§3.2.1)		x	**12**	275	**8.1h**
6	multi-solve (§3.1.2)	x	x	N/A	35	9.5h
7	multi-facto. (§3.2.2)	x	x	12	82	2.3h
8	multi-facto. (§3.2.2)	x	x	6	92	1.2h
9	multi-facto. (§3.2.2)	x	x	**3**	137	**52m**

Table 3.1: Performance of various algorithms on the industrial test case with numerical compression optionally enabled. OOM stands for 'out of memory'.

erence, we have performed preliminary experiments with compression turned off both in the sparse (unlike in the rest of the chapter) and dense solvers (rows 1 - 3 in the table). In this case, the state-of-the-art advanced vanilla coupling (see Section 2.2.2) and the multi-factorization algorithm can simply not run on this machine by lack of memory, multi-solve is the only uncompressed solver that can run here. In the first time (rows 4 - 5), adding compression in the sparse solver reduces CPU time and memory consumption for the multi-solve, and allows multi-factorization to complete successfully (using more memory but less time than the multi-solve). In a second time (rows 6 - 7), using compression in the dense solver yields an even larger improvement in CPU time and RAM usage. Finally (rows 8 - 9), multi-factorization can be further accelerated by lowering the number of tiles n_b per block row and block column of the Schur complement matrix. This reduces the number of factorizations at the cost of an increase in the memory usage. Hence, the benefit of the memory gain coming from our algorithms is twofold: one, it allows us to run cases that were inaccessible otherwise, and second, the memory spared can be used to increase the Schur complement block size and reduce even further the CPU-time in the multi-factorization approach. In the view of these results, multi-factorization is the privileged approach in production for this type of test case on this type of machines (but this conclusion strongly depends on the number of unknowns and the amount of memory available). An example of physical result is presented on Figure 3.9.

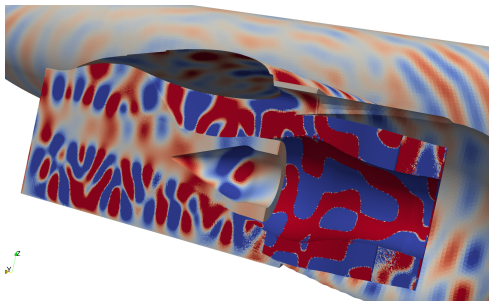


Figure 3.9: Industrial test case result: the acoustic pressure is visualized in the flow (at the front) and on the surface of the plane (at the back). The color scale is saturated, so as to see the acoustic pressure on the fuselage, which is much smaller than the pressure in the flow (as one might expect, the noise is much higher *inside* the engine). The blurry pale part of the flow on the left is the hot part of the jet flow, coming out of the combustion chamber (not represented). It underlines the strong heterogeneity of the flow.

3.5 Conclusion

In this chapter, we proposed the multi-solve and multi-factorization two-stage algorithms adopting the strategy of exploiting the existing API of sparse and dense direct solvers for the solution of coupled sparse/dense systems such as (1.2). The design of the algorithms makes it possible for them to benefit from the most advanced features and building blocks of the fully-featured direct solvers they are built on (such as the internal management of the Schur complement, compression techniques and sparse right-hand sides). Thanks to multi-solve and multi-factorization, and especially their further compressed variants, we were able to process academic aeroacoustic problems more than 5× larger than standard coupling approaches (see Section 2.2) allow for on a given shared-memory multi-core machine. We furthermore showed that the algorithms can take advantage of the whole available memory to increase their performance, in a memory-aware fashion. Regarding the multi-factorization algorithm, it proved to be particularly interesting in the context of the industrial test case when it was up to 11× faster than the multi-solve approach. In chapters 4 and 5, we extend this work to the out-of-core and distributed-memory cases, respectively.

Chapter 4

Out-of-core two-stage algorithms in shared memory

In Chapter 3, we proposed two-stage algorithms, namely multi-solve and multi-factorization, for solving coupled sparse/dense FEM/BEM systems such as (1.2) defined in Section 1.3. The main strength of these algorithms is to rely on well-optimized and fully-featured sparse and dense direct solvers. Fully-featured solvers implement some advanced functionalities we want to take advantage of in order to reduce the time to solution and memory requirements of our algorithms with the aim to process ever larger sparse/dense FEM/BEM systems. In particular, we are interested in numerical compression and out-of-core computation which amounts to moving currently unused data from RAM to disk, out of the core memory (see Section 2.6.1). Both of the selected sparse and dense direct solvers implement these functionalities within their different building blocks (see Section 2.1). Nevertheless, the current API of the solvers prevents us from efficiently using the advanced features on the articulation between the sparse and the dense building blocks, i.e. on the Schur complement part S of the system (see Section 1.3). The multi-solve and multi-factorization algorithms were designed to cope with these limitations. In Chapter 3, we focused on the application of numerical compression on S. We showed that, on a shared-memory workstation, the two-stage algorithms allow us to process considerably larger coupled systems compared to the state-of-the-art vanilla solver couplings (see Section 2.2). This could be achieved even without compressing S. However, its compression gives us the possibility to process even larger coupled systems. The purpose of this chapter is to explore the possibility of taking advantage of out-of-core computation within the solution process, including on the Schur complement matrix. The goal is to further reduce the memory consumption of the two-stage algorithms on a single shared-memory workstation and potentially allow for processing even larger coupled systems.

In sections 4.1 and 4.2, we thus propose an extension of the multi-solve and multi-factorization two-stage algorithms, respectively, to out-of-core computation. In Section 4.3, we rely on academic test cases to analyze the impact of out-of-core on the time to solution and memory consumption in shared memory. Experiments on an industrial application are discussed in Section 4.4. We conclude in Section 4.5.

4.1 Out-of-core multi-solve algorithm

This section introduces the extensions of the *baseline multi-solve* (see Algorithm 3 in Section 3.1.1) and the *compressed Schur multi-solve* (see Algorithm 4 in Section 3.1.2) two-stage algorithms to out-of-core computation.

As of the multi-solve algorithm, the result Y_i of the *sparse solve* step $(L_{vv}L_{vv}^T)^{-1}A_{sv_i}^T$ (line 4 in Algorithm 3 and line 5 in Algorithm 4), can only be retrieved in RAM, i.e. in-core (see Section 2.6.1). The block Z_i is also stored in-core. In *baseline multi-solve* (see Algorithm 3), A_{ss} is split into square out-of-core blocks (see Figure 4.1a). Then, when assembling S_i, we load data into memory from only one out-of-core block at a time. Furthermore, as the dimension of out-of-core blocks usually does not match the dimension of S_i blocks, we load into memory only the portion of the out-of-core block of A_{ss} which is overlayed by the current S_i block. Regarding *compressed Schur multi-solve* (see Algorithm 4, Figure 4.1b), the out-of-core feature of the dense direct solver is dynamically and transparently handled by the runtime on top of which the solver is implemented (see Section 2.6). The runtime starts to eject data of compressed Z_i and A_{ss} to disk when an arbitrary chosen memory limit is reached during the computation. In the experiments below, the dense solver is instrumented to use not more than 4% of available RAM. The remaining portion is left for the sparse direct solver and other data.

4.2 Out-of-core multi-factorization algorithm

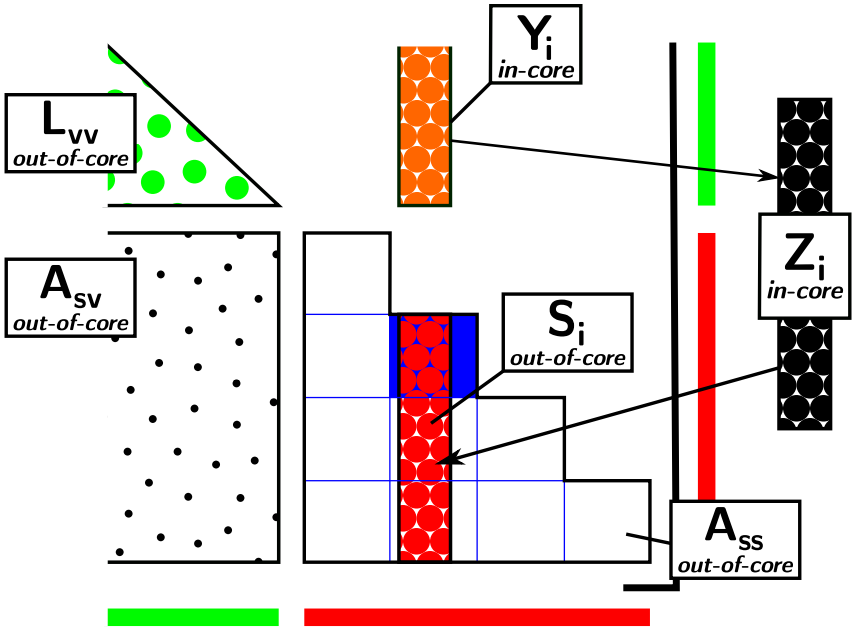
In this section, we propose the extensions of the *baseline multi-factorization* (see Algorithm 5 in Section 3.2.1) and the *compressed Schur multi-factorization* (see Algorithm 6 in Section 3.2.2) two-stage algorithms to out-of-core computation.

In the case of the multi-factorization algorithm, the Schur complement block X_{ij} (see Figure 4.2) associated with the temporary submatrix W is returned entirely assembled in RAM (in-core). Just like for *baseline multi-solve*, in the *baseline multi-factorization* case (see Algorithm 5), A_{ss} is split into square out-of-core blocks (see Figure 4.2a). However, here the dimension of the Schur blocks X_{ij} and S_{ij} is a multiple of the out-of-core block dimension. Then, when assembling S_{ij}, we load only one out-of-core block at a time into RAM. Regarding the *compressed Schur multi-factorization* (see Algorithm 6, Figure 4.2b), the out-of-core feature of the dense direct solver, concerning the compressed X_{ij} and A_{ss}, is dynamically and transparently handled by the underlying runtime as it is for *compressed Schur multi-solve* (see Section 4.1).

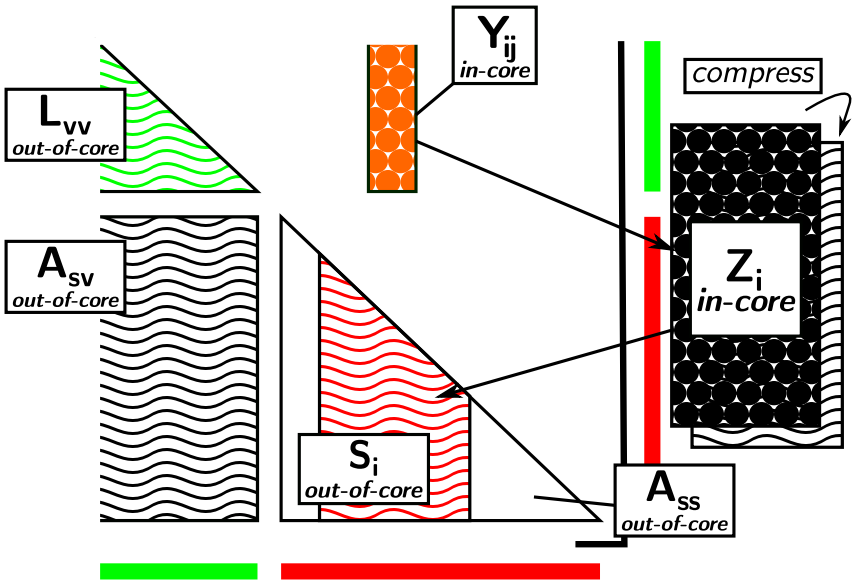
4.3 Experimental results

We conducted an experimental study of the multi-solve and multi-factorization algorithms for solving larger coupled sparse/dense FEM/BEM linear systems such as defined in (1.2) in order to evaluate the impact of out-of-core computation performance. For the purpose of this evaluation, we used both the *wide pipe* and the *narrow pipe* test cases (see Section 1.4.1).

We rely on the implementation of the multi-solve and multi-factorization algorithms from Section 3.3.1.1 in Chapter 3 on top of the MUMPS/SPIDO and MUMPS/HMAT solver couplings respectively for the *baseline* variants and the *compressed* variants of the algorithms. We use double precision accuracy. The precision parameter ϵ is set to 10^{-3} for both MUMPS and HMAT (see Section 1.5.2.3). The low-rank compression in MUMPS is systematically turned on for both MUMPS/SPIDO and MUMPS/HMAT. Measurements of computation time and RAM usage are done as described in Section 3.3.2 of Chapter 3. Like RAM consumption, disk usage is monitored by the `rss.py` Python script but through the `du` command [3].

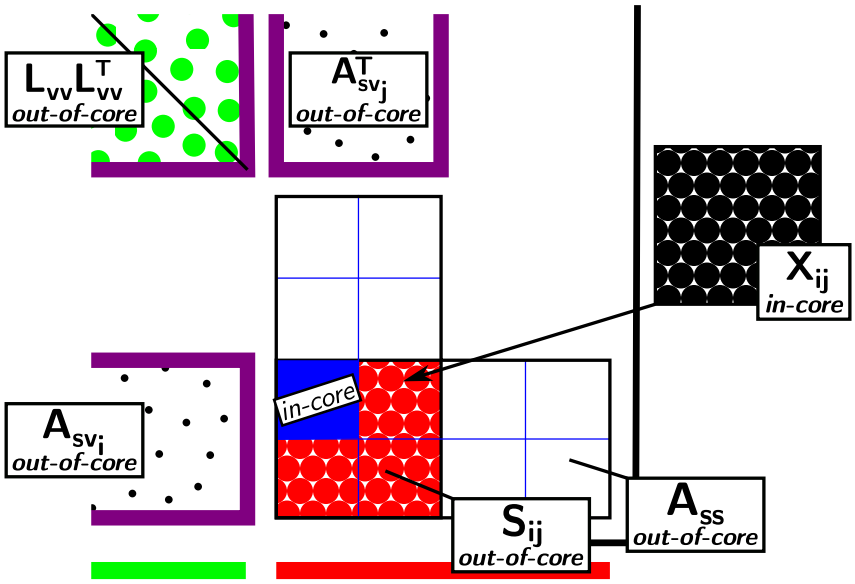


(a) *baseline multi-solve*, blue lines depict the partitioning of A_{ss} into out-of-core blocks.

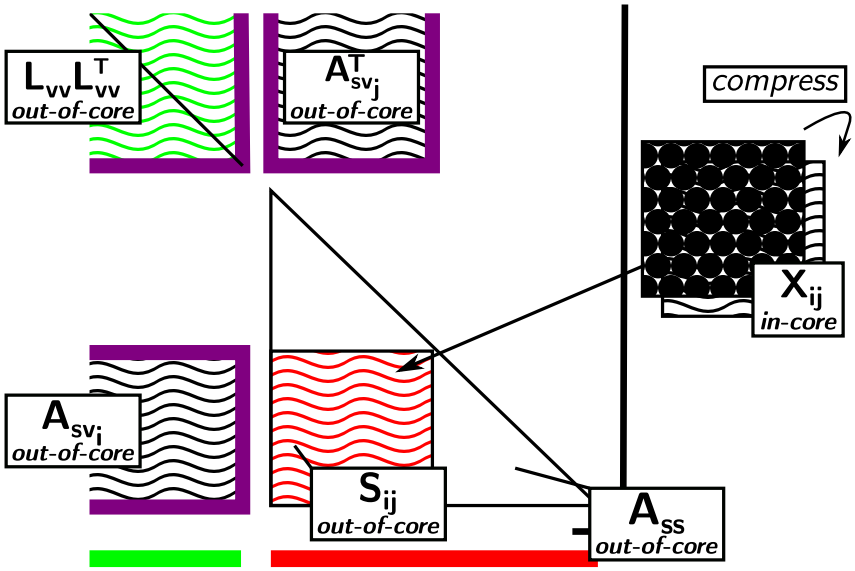


(b) *compressed Schur multi-solve*.

Figure 4.1: Out-of-core block-wise computation of S using the multi-solve algorithm (zoom on the Schur complement dense part of the system).



(a) *baseline multi-factorization*, blue lines depict the partitioning of A_{ss} into out-of-core blocks.



(b) *compressed Schur multi-factorization*.

Figure 4.2: Out-of-core block-wise computation of S using the multi-factorization algorithm (zoom on the Schur complement dense part of the system).

We have conducted our experiments on `bora` nodes on the PlaFRIM platform [19]. A `bora` node has a total of 36 processor cores running each at 2.5 GHz, 192 GiB of RAM and a local hard disk SATA Seagate ST1000NX0443 turning at 7200 rpm. It is the policy of the platform to deactivate Hyper-Threading and Turbo-Boost in order to improve the reproducibility of the experiments. The solver test suite is compiled with GNU C Compiler (gcc) 12.1.0, Intel(R) MKL library 2019.1.144, MUMPS 5.5.1 and StarPU 1.3.9. Each run presented below uses one node running one process with 36 threads. Regarding the out-of-core feature, we consider three scenarios. The *All in-core* label is used for benchmarks where out-of-core has been completely disabled in both sparse and dense solvers, i.e. all the data has been stored in the core memory (in-core). Then, the *Out-of-core except Schur complement* label is used for benchmarks where out-of-core has been enabled only in the sparse direct solver, i.e. the dense part (A_{ss} and consequently S) has been stored in-core. Finally, the *All out-of-core* label has been used for benchmarks where out-of-core has been enabled for both the sparse and the dense direct solver, including the Schur complement part S.

The goal of this study is to determine whether out-of-core techniques may help us to lower the memory footprint of the two-stage algorithms on a shared-memory workstation so as to allow for processing even larger coupled systems and if so, to which extent. We consider coupled FEM/BEM linear systems with N, the total unknown count, starting at 1,000,000 and increasing until the memory limit is reached. As of the algorithm parameters, we evaluate multiple configurations. For *baseline multi-solve* (see Section 3.1.1) relying on the MUMPS/SPIDO coupling, we vary the size n_c of blocks $A_{sv_i}^T$ of columns of the A_{sv}^T submatrix between 128 and 512. For the *compressed Schur multi-solve* variant (see Section 3.1.2), relying on the MUMPS/HMAT coupling, the size of blocks of columns of S and A_{sv}^T is handled by the n_S and n_c parameters, respectively. In this case, we set n_c to a constant value of 256 columns (motivated by the results of the study in Section 3.3.4 and the RAM capacity of the `bora` nodes) and vary n_S in the range from 256 to 1,024. In the case of the multi-factorization algorithm, both the *baseline multi-factorization* (see Section 3.2.1) and the *compressed Schur multi-factorization* variants (see Section 3.2.2) expose the n_b parameter handling the count of blocks S_{ij} per block row and block column of the Schur complement submatrix S. The tested values of n_b are 1, 3, 7 and 11. As a reference from state of the art, we consider the advanced vanilla solver coupling (see Section 2.2.2). From the implementation point of view, we rely on the *baseline multi-factorization* algorithm with the MUMPS/SPIDO coupling and n_b set to 1 (see further details in Section 3.3.1.2). The corresponding results are represented by blue curves with round points in the figures below.

4.3.1 *wide pipe*

In the first place, we consider the *wide pipe* academic test case (see Section 1.4.1). Note that compared to the *narrow pipe*, it has a larger volume mesh and therefore a higher ratio of FEM-related unknowns to BEM-related unknowns (see sections 1.3 and 1.4).

4.3.1.1 Multi-solve

Figures 4.3 and 4.4 show respectively the computation times as well as the peak RAM and hard drive usage of the multi-solve algorithm for both of the solver couplings, i.e. MUMPS/SPIDO and MUMPS/HMAT.

When both the sparse and the dense solver keep all the data in RAM (see the *All in-core* column in the figures), MUMPS/SPIDO is able to process up to 8,000,000 unknowns with n_c set up to 256 columns before reaching the memory limit. When the Schur complement part is compressed, i.e. in the case of MUMPS/HMAT, we can reach up to 12,000,000 unknowns.

Now, using out-of-core exclusively in the sparse solver (see the *Out-of-core except Schur complement* column in the figures) enables MUMPS/SPIDO to solve systems with N up to 9,000,000 before running out of RAM. With MUMPS/HMAT, we can go up to 14,000,000. However, in this case, we were not bound by the RAM, used at 62%, but by the time limit of the PlaFRIM platform for one execution, i.e. 3 days. Putting the RAM consumption aside, running the sparse solver out-of-core slows down the computation in the case of both couplings. In average, we can observe 67% slow-down for MUMPS/SPIDO and 69% for MUMPS/HMAT with respect to the *All in-core* benchmarks. Such an important overhead can be explained by the numerous calls to the *sparse solve* step in MUMPS during the Schur complement assembly. Indeed, in order to perform each of the *sparse solve* steps of MUMPS in multi-solve, the solver reads the factors of the previously factorized A_{vv} from disk. However, thanks to out-of-core, we can rise n_c to 512 which reduces the number of calls to *sparse solve* and thus allows us to counterbalance the slow-down. Understanding the different levels of overhead between MUMPS/SPIDO and MUMPS/HMAT would require a deeper investigation which is beyond the scope of this study.

Running all the solvers out-of-core (see the *All out-of-core* column in the figures) further degrades the time to solution leading to an overhead of 70% for MUMPS/SPIDO and 80% for MUMPS/HMAT. In this case, MUMPS/SPIDO and MUMPS/HMAT could process coupled systems with up to 14,000,000 unknowns while taking respectively 72% and 62% of available RAM. Past this problem size, the execution time exceeded the limit of 3 days on the PlaFRIM platform. This is also the explanation of why MUMPS/SPIDO with $n_c < 512$ could not reach further than 9,000,000 unknowns.

In summary, while it induces an overhead in terms of computation time, out-of-core allowed us to significantly lower the memory footprint of the multi-solve algorithm thanks to its activation in the Schur complement part in particular. As a result, multi-solve could process coupled systems which are up to 1,75× larger than without out-of-core, i.e. 14,000,000 against 8,000,000, and up to 7× larger than the state-of-the-art advanced vanilla coupling, i.e. 14,000,000 against 2,000,000 (see the blue round points in the *All in-core* column of Figure 4.5 representing an execution of MUMPS/SPIDO using the *baseline multi-factorization* algorithm with the entire Schur complement processed at once, i.e. with n_b set to 1). When out-of-core is used concurrently with numerical compression of S, the impact of out-of-core is only slightly noticeable due to the efficiency of the compression. This trend could be different on larger problems. Nevertheless, such experiments would require a high-performance computing platform with less restrictive execution time limits.

4.3.1.2 Multi-factorization

Figures 4.5 and 4.6 respectively show the computation times as well as the peak RAM and hard drive usage of the multi-factorization algorithm for both of the solver couplings, i.e. MUMPS/SPIDO and MUMPS/HMAT. Unlike in the case of multi-solve (see Section 4.3.1.1), we report only a very low computation time overhead associated to the usage of out-of-core. Without out-of-core and with out-of-core activated only in the sparse solver (see the *All in-core* and the *Out-of-core except Schur complement* columns in the figures), the MUMPS/SPIDO coupling can process up to 2,000,000 unknowns before reaching the memory limit. When running both the sparse and the dense solvers out-of-core (see the *All out-of-core* column in the figures), MUMPS/SPIDO allowed us to reach 3,000,000 unknowns with n_b set to 11. Thanks to the compression of the Schur complement part, i.e. with MUMPS/HMAT, we can reach up to 3,000,000 unknowns. Activating out-of-core in the sparse solver then allows for lowering the value of n_b from 11 to 3. This reduces the number of calls to the *sparse factorization+Schur* building block (see Section 2.1) in the core phase of multi-factorization (see Section 3.2) and thus significantly accelerates the computation of the 3,000,000-unknown test case.

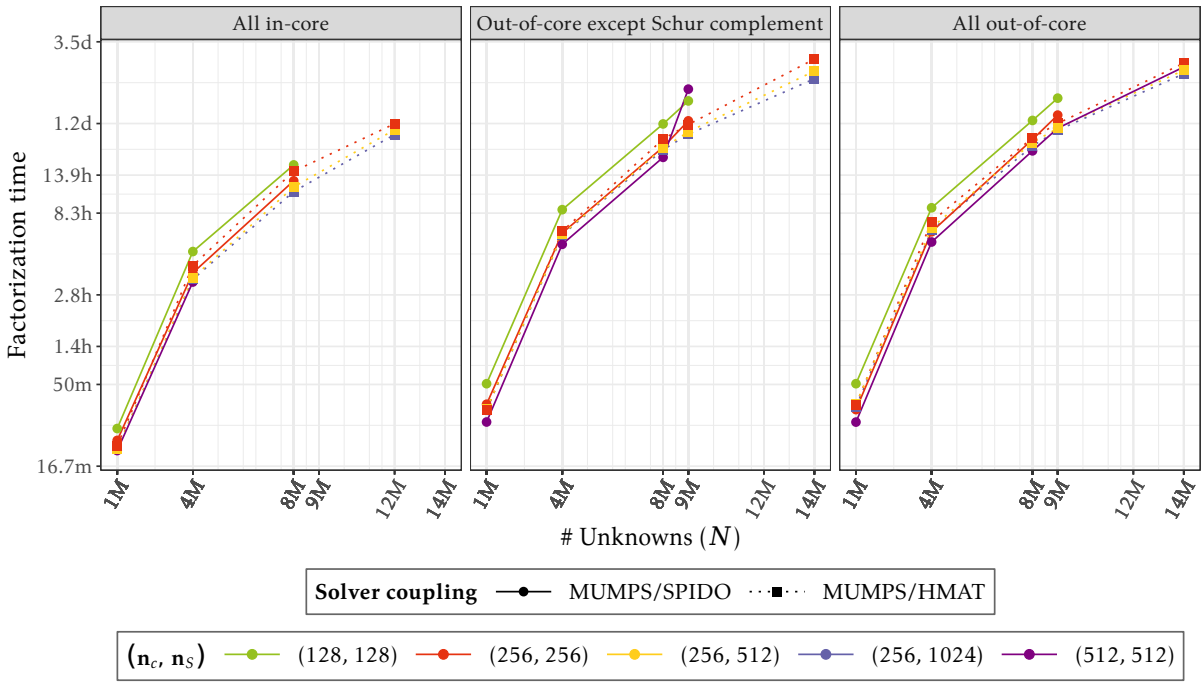


Figure 4.3: Computation times of **multi-solve** on coupled FEM/BEM linear systems of varying number of unknowns for both of the solver couplings MUMPS/HMAT and MUMPS/SPIDO and for varying values of n_c and n_S. We test 3 different configurations of the *out-of-core* feature: completely disabled (all in-core), enabled **except** for the Schur complement matrix S or enabled **including** for S (all out-of-core). Parallel runs on single `bora` node using the *wide pipe* test case.

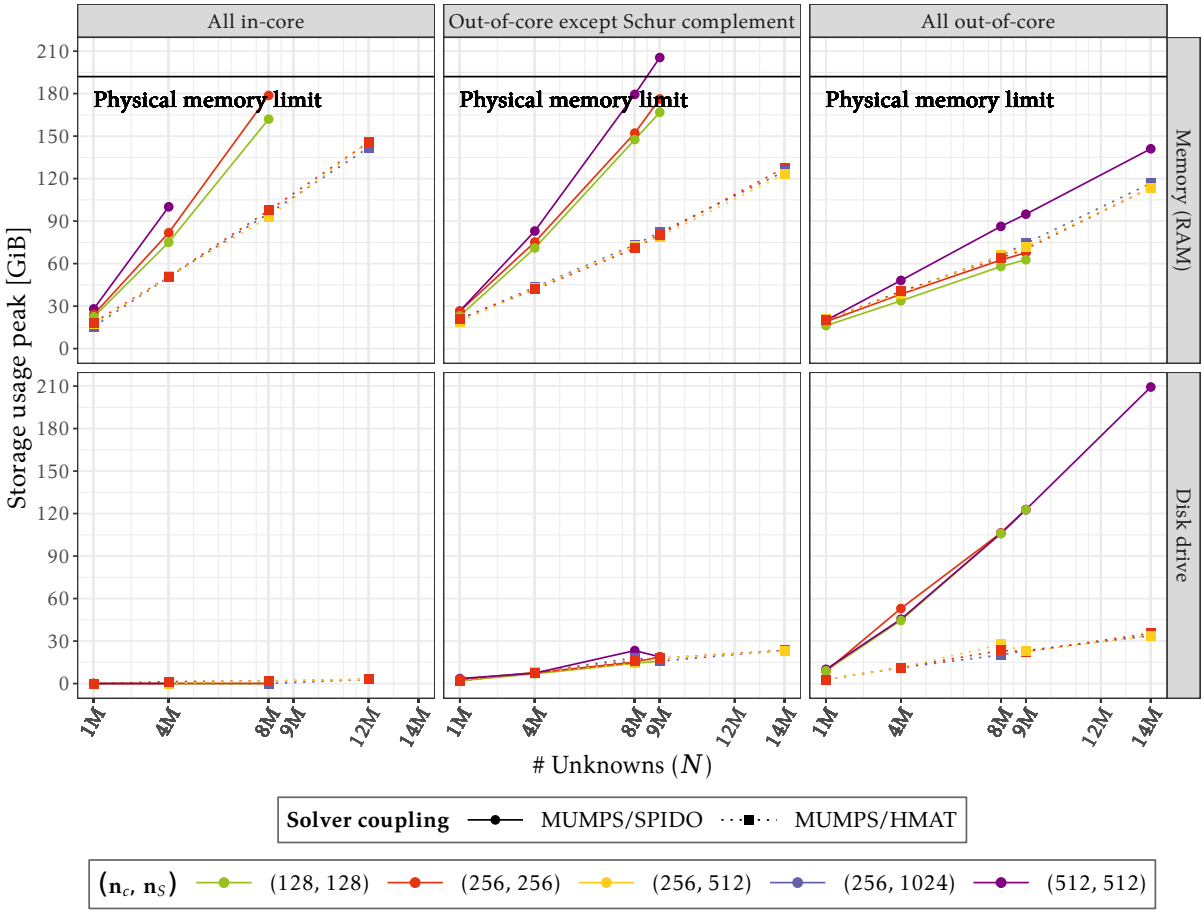


Figure 4.4: RAM and hard drive usage peaks of **multi-solve** on coupled FEM/BEM linear systems of varying number of unknowns for both of the solver couplings MUMPS/HMAT and MUMPS/SPIDO and for varying values of n_c and n_S. We test 3 different configurations of the *out-of-core* feature: completely disabled (all in-core), enabled **except** for the Schur complement matrix S or enabled **including** for S (all out-of-core). Parallel runs on single `bora` node using the *wide pipe* test case.

However, in this situation, the dense Schur complement part is not large enough with respect to the sparse submatrices of A, for it to represent the most important limitation in terms of RAM consumption. Actually, the bottleneck is the amount of RAM required by the sparse solver during the Schur complement assembly. The multi-factorization algorithm implies data duplication related to the loss of symmetry in the temporary submatrix W on which the *sparse factorization+Schur* building block (see Section 2.1) is applied (see Section 3.2). The loss of symmetry then amplifies the negative impact of fill-in in W on RAM usage. In addition to that, the out-of-core feature of MUMPS has only a limited influence on the RAM consumption of *sparse factorization+Schur* because the Schur complement is kept in the core memory as discussed in Section 2.1.1.2. This also explains the low overhead of out-of-core in this case. Nevertheless, we expect this trend to vary depending on the ratio of the number of unknowns in the dense part of the system to the number of unknowns in its sparse part. We therefore evaluate out-of-core in the multi-factorization algorithm on a second academic test case with higher proportion of unknowns in the dense part and further on an industrial test case in Section 4.4.

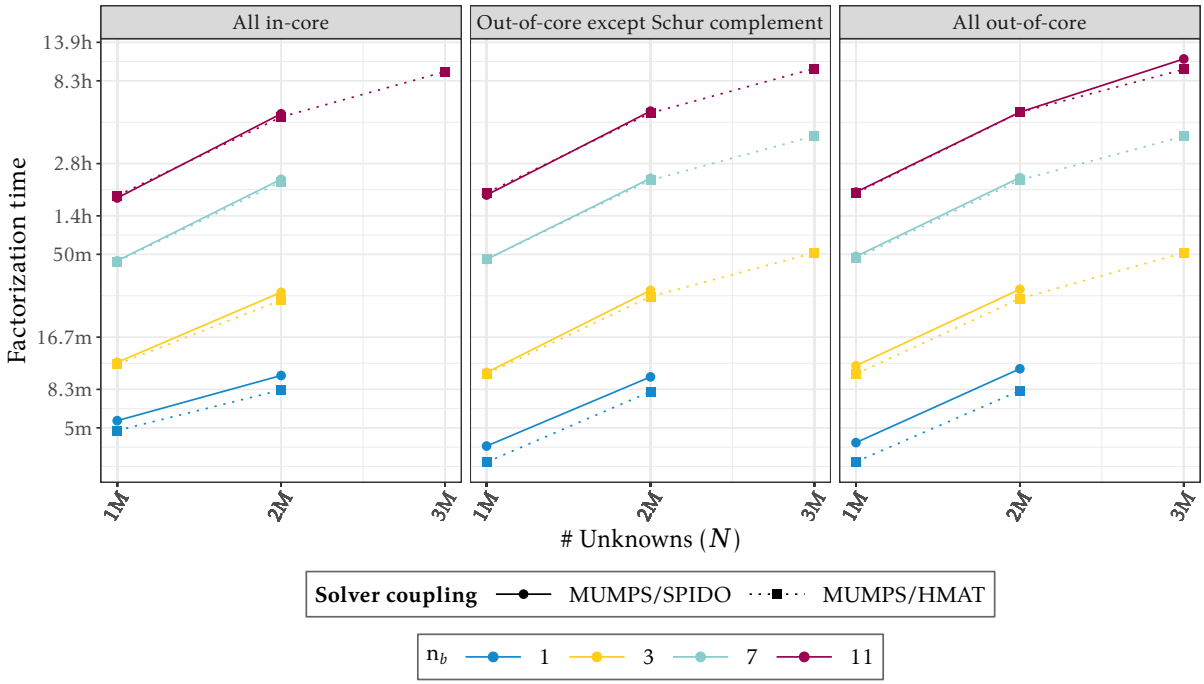


Figure 4.5: Computation times of **multi-factorization** on coupled FEM/BEM linear systems of varying number of unknowns for both of the solver couplings MUMPS/HMAT and MUMPS/SPIDO and for varying values of n_b. We test 3 different configurations of the *out-of-core* feature: completely disabled (all in-core), enabled **except** for the Schur complement matrix S or enabled **including** for S (all out-of-core). Parallel runs on single `bora` node using the *wide pipe* test case.

4.3.2 *narrow pipe*

In the second part of this study, we consider the *narrow pipe* academic test case (see Section 1.4.1). Compared to the *wide pipe*, it has a smaller volume mesh and therefore a higher ratio of BEM-related unknowns in the dense part of the resulting system to FEM-related unknowns in the sparse part of the resulting system (see sections 1.3 and 1.4).

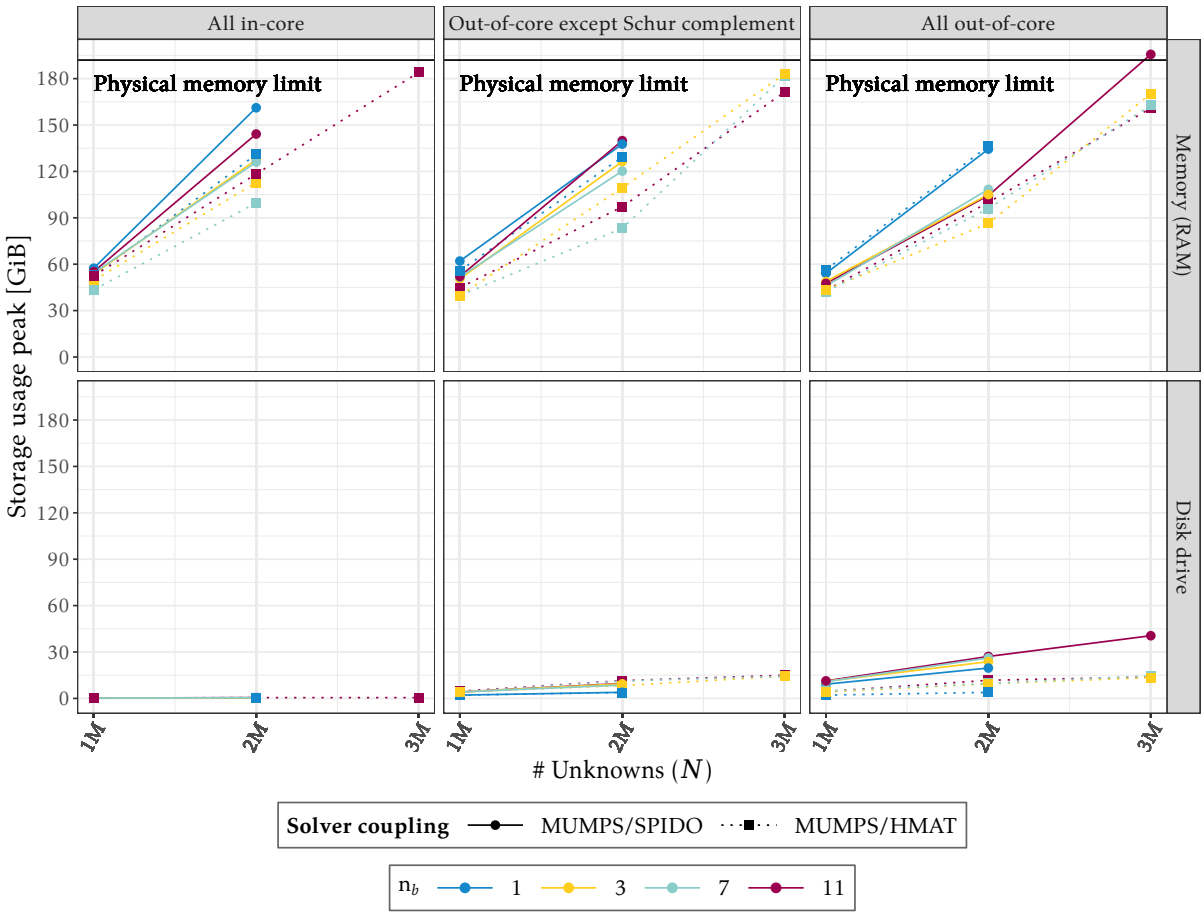


Figure 4.6: RAM and hard drive usage peaks of **multi-factorization** on coupled FEM/BEM linear systems of varying number of unknowns for both of the solver couplings MUMPS/HMAT and MUMPS/SPIDO and for varying values of n_b. We test 3 different configurations of the *out-of-core* feature: completely disabled (all in-core), enabled **except** for the Schur complement matrix S or enabled **including** for S (all out-of-core). Parallel runs on single `bora` node using the *wide pipe* test case.

4.3.2.1 Multi-factorization

Considering the *narrow pipe* test case, figures 4.7 and 4.8 show respectively the computation times as well as the peak RAM and hard drive usage of the multi-factorization algorithm for both of the solver couplings, i.e. MUMPS/SPIDO and MUMPS/HMAT.

When out-of-core is disabled for both the sparse and the dense solver (see the *All in-core* column of the figure), MUMPS/SPIDO can process up to 3,000,000 unknowns with n_b set to 11 blocks and up to 2,000,000 with n_b in between 1 and 7 blocks before reaching the memory limit. When the Schur complement part is compressed, i.e. in the case of MUMPS/HMAT, we can reach up to 4,000,000 unknowns in total. Moreover, we can process the 3,000,000 problem using only 7 blocks per block row and column of S instead of 11 like in the case of MUMPS/SPIDO.

Activating out-of-core only in the sparse solver (see the *Out-of-core except Schur complement* column in the figure) has again a limited effect. The overhead with respect to the *All in-core* benchmarks was of 3% for MUMPS/SPIDO and 4% for MUMPS/HMAT. However, MUMPS/HMAT could reach 4,000,000 of unknowns with n_b set to 7 instead of 11 and 3,000,000 unknowns with n_b set to 3 instead of 7.

Unlike in the *wide pipe* test case (see Section 4.3.1.2), using out-of-core also on the Schur complement part (see the *All out-of-core* column in the figure) brings more advantages. On the one hand, MUMPS/SPIDO is capable of processing systems with up to 4,000,000 of unknowns using less Schur complement blocks, i.e. with lower n_b. On the other hand, the 3,000,000-unknown case can be processed by both couplings with n_b set to 3 blocks, i.e. resorting to 6 calls to *sparse factorization+Schur* instead of 66 ($n_b = 11$) and 28 ($n_b = 7$) respectively for MUMPS/SPIDO and MUMPS/HMAT without using out-of-core on the Schur complement part. In the *All out-of-core* configuration, the reported slow-down with respect to the *All in-core* benchmarks was of 9% for MUMPS/SPIDO and 3% for MUMPS/HMAT.

Compared to the case of multi-solve (see Section 4.3.1.1), the impact of out-of-core, especially in the dense part of the system, when combined with numerical compression is more present both in terms of better computation time and lower RAM consumption.

4.4 Industrial application

This section presents an example of industrial application processed at Airbus Central R&T using the multi-solve and multi-factorization algorithms. Here, we use the same industrial test case and experimental setup as in Section 3.4. We remind the reader that compared to both of the *wide pipe* and the *narrow pipe* academic test cases also employed in this study (see Section 1.4.1), the industrial test case covers a larger surface domain leading to a higher proportion of BEM-related unknowns in the associated coupled linear system. Hence the relative cost of the dense BEM part will be more important and numerical compression as well as out-of-core techniques will have a bigger impact on time to solution and RAM consumption.

4.4.1 No numerical compression

For reference, we have performed preliminary benchmarks without compression neither in the sparse nor in the dense solver. The corresponding results are provided in Table 4.1. In the case of multi-solve, when we activate out-of-core in the dense solver (row 2), it allows us to considerably decrease the RAM usage from 255 to 52 GiB for an overhead of only 4% in computation time. When the sparse solver is out-of-core as well (row 3), we can further reduce the memory consumption but the associated slow-down becomes excessively high, i.e. 380%.

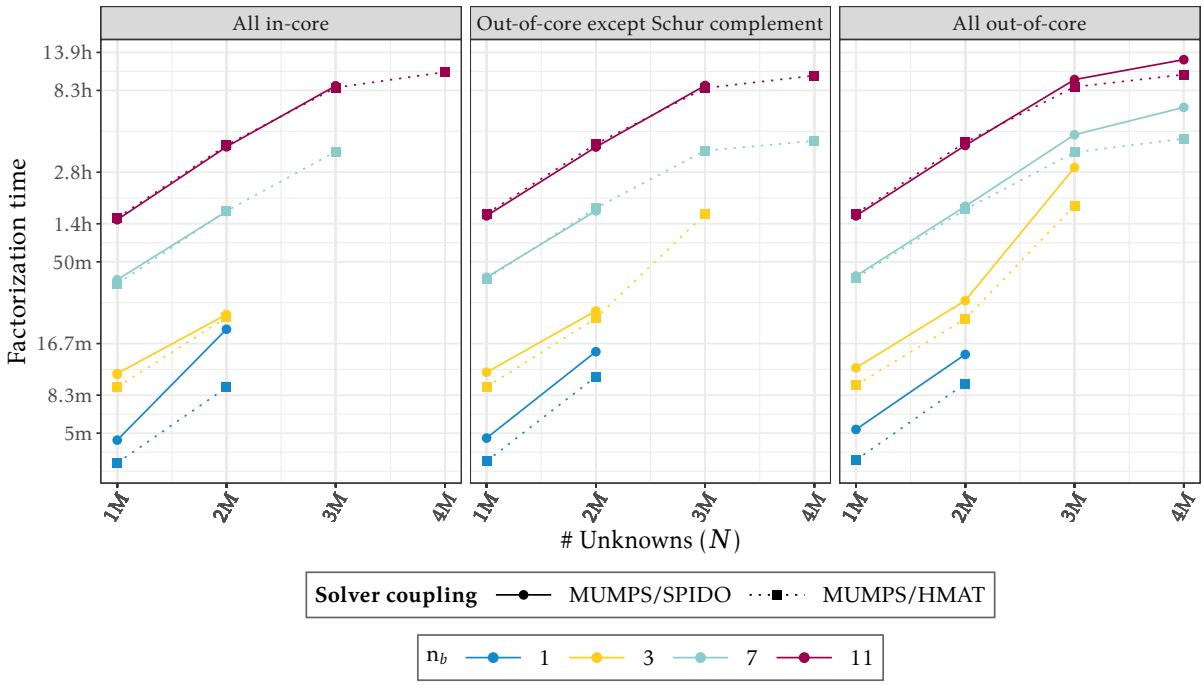


Figure 4.7: Computation times of **multi-factorization** on coupled FEM/BEM linear systems of varying number of unknowns for both of the solver couplings MUMPS/HMAT and MUMPS/SPIDO and for varying values of n_b. We test 3 different configurations of the *out-of-core* feature: completely disabled (all in-core), enabled **except** for the Schur complement matrix S or enabled **including** for S (all out-of-core). Parallel runs on single `bora` node using the *narrow pipe* test case.

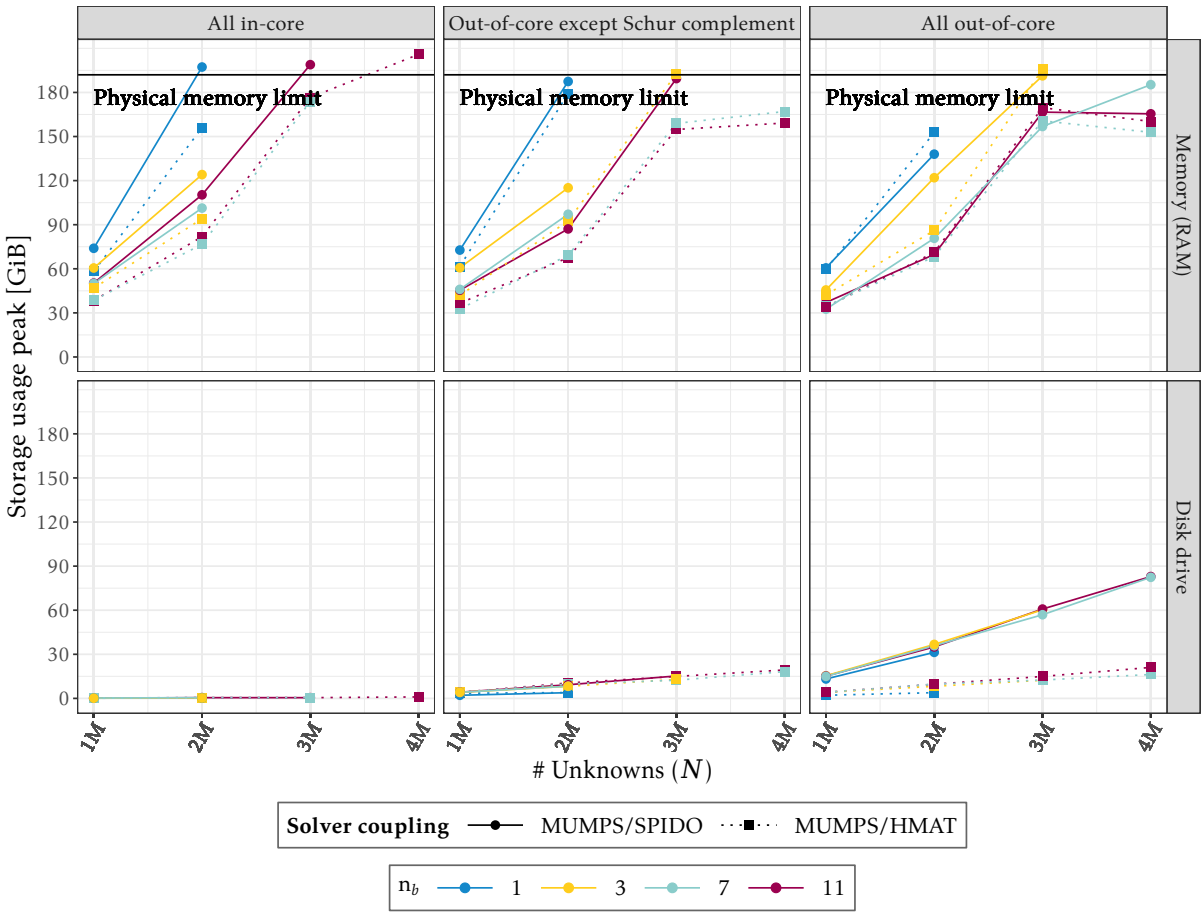


Figure 4.8: RAM and hard drive usage peaks of **multi-factorization** on coupled FEM/BEM linear systems of varying number of unknowns for both of the solver couplings MUMPS/HMAT and MUMPS/SPIDO and for varying values of n_b. We test 3 different configurations of the *out-of-core* feature: completely disabled (all in-core), enabled **except** for the Schur complement matrix S or enabled **including** for S (all out-of-core). Parallel runs on single `bora` node using the *narrow pipe* test case.

This is due to the fact that the sparse solver reads the factors of A_{vv} from disk on each call to the *sparse solve* building block during the Schur complement assembly. As the compression is deactivated, this represent large amounts of data. Without out-of-core, multi-factorization can only run with n_b set to 12 blocks (row 4) which leads to 1 day of execution time compared to 18.1 hours for multi-solve. Activating out-of-core in the dense and then also in the sparse solver allows us to lower n_b to 3 blocks, reduce the number of re-factorizations from 78 (with n_b set to 12) to 6 and process the linear system in 12.8 hours.

	Algorithm	Dense out-of-core	Sparse out-of-core	n_b	RAM (GiB)	Disk (GiB)	Time
1	multi-solve (§3.1.1)			N/A	255	25	18.1h
2	multi-solve (§4.1)	x		N/A	52	229	18.8h
3	multi-solve (§4.1)	x	x	N/A	12	265	3.6d
4	multi-facto. (§3.2.1)			**12**	377	20	**1d**
5	multi-facto. (§4.2)	x		12	207	223	1.4d
6	multi-facto. (§4.2)	x		6	227	223	20h
7	multi-facto. (§4.2)	x	x	12	55	398	1.6d
8	multi-facto. (§4.2)	x	x	6	68	408	22.4h
9	multi-facto. (§4.2)	x	x	**3**	115	423	**12.8h**

Table 4.1: Performance of the two-stage algorithms on the industrial test case without compression and with out-of-core optionally enabled.

4.4.2 Numerical compression in sparse solver

This time, we activate the numerical compression in the sparse solver but not in the dense solver. The results are presented in Table 4.2. In multi-solve, out-of-core in the dense solver (row 2) brings again an important decrease in RAM usage without degrading the computation time. Using out-of-core also in the sparse solver (row 3) leads to a lower overhead compared to the reference benchmarks (see Section 4.4.1), i.e. 54% instead of 380%, but does not considerably reduce the RAM consumption either which is already low thanks to the out-of-core in the dense solver. Here, multi-factorization performs better in all its configurations. However, thanks to out-of-core in the dense solver (rows 5 and 6) and further in the sparse parts of the system, we can lower n_b to 3 and accelerate the computation 3.25× (row 9) compared to the fastest multi-solve executions (rows 1 and 2).

	Algorithm	Dense out-of-core	Sparse out-of-core	n_b	RAM (GiB)	Disk (GiB)	Time
1	multi-solve (§3.1.1)			N/A	229	25	15.6h
2	multi-solve (§4.1)	x		N/A	24	229	15.6h
3	multi-solve (§4.1)	x	x	N/A	14	235	1d
4	multi-facto. (§3.2.1)			**12**	279	20	**8.3h**
5	multi-facto. (§4.2)	x		12	80	223	8.5h
6	multi-facto. (§4.2)	x		6	82	223	6.2h
7	multi-facto. (§4.2)	x	x	12	57	235	8.9h
8	multi-facto. (§4.2)	x	x	6	64	235	6.6h
9	multi-facto. (§4.2)	x	x	**3**	120	235	**4.8h**

Table 4.2: Performance of the two-stage algorithms on the industrial test case with compression in the sparse solver, without compression in the dense solver and with out-of-core optionally enabled.

4.4.3 Numerical compression in both sparse and dense solvers

Eventually, we activate the numerical compression in both the sparse and the dense solvers and observe the impact of out-of-core. The results are in Table 4.3. Regarding multi-solve, the compression in all the parts of the system brings an important gain in time to solution and RAM usage already (row 1). Moreover, activating out-of-core leads to only a limited decrease in memory consumption but to an important slow-down which may go as high as 213% (row 3). Similar observations come out of the analysis of *compressed Schur multi-solve*. In this case, the runs without out-of-core seem to be the best-performing ones in terms of the computation time. With n_b set to 3 (row 6), we were able to perform the computation in only 50 minutes. For the benchmarks resorting to out-of-core in rows 9 and 12 the memory consumption is even slightly higher than without it (row 6). So far, we have not found the reason for this increase. A deeper investigation is required to explain the phenomenon.

	Algorithm	Dense out-of-core	Sparse out-of-core	n_b	RAM (GiB)	Disk (GiB)	Time
1	multi-solve (§3.1.2)			N/A	35	4	9.3h
2	multi-solve (§4.1)	x		N/A	34	11	10.4h
3	multi-solve (§4.1)	x	x	N/A	22	17	29.1h
4	multi-facto. (§3.2.2)			12	81	5	2.1h
5	multi-facto. (§3.2.2)			6	122	5	1.1h
6	multi-facto. (§3.2.2)			3	142	5	50m
7	multi-facto. (§4.2)	x		12	89	11	3h
8	multi-facto. (§4.2)	x		6	113	11	1.2h
9	multi-facto. (§4.2)	x		3	162	11	53m
10	multi-facto. (§4.2)	x	x	12	75	17	2.5h
11	multi-facto. (§4.2)	x	x	6	103	17	1.2h
12	multi-facto. (§4.2)	x	x	3	150	15	52m

Table 4.3: Performance of the two-stage algorithms on the industrial test case with compression in both the sparse and the dense solvers and with out-of-core optionally enabled.

4.5 Conclusion

In this chapter, we proposed an extension of the multi-solve and multi-factorization two-stage algorithms to out-of-core computation. We validated the implementation on an experimental study including both academic and industrial test cases. On the one hand, when both the sparse and the dense solvers resort to numerical compression, the effect of out-of-core on the computation time and the RAM consumption of the two-stage algorithms is limited thanks to the efficiency of numerical compression. On the other hand, in a framework where numerical compression is not available, out-of-core computation allows us to significantly lower the memory footprint of the algorithms, especially when applied to the dense Schur complement part. On a given shared-memory multi-core machine, we could thus process academic aeroacoustic problems larger than both state-of-the-art advanced vanilla coupling (see Section 2.2.2) and the two-stage algorithms themselves without resorting to out-of-core techniques. Regarding the computation time, the overhead was more present in the case of multi-solve than in the case of multi-factorization. For multi-solve, the overhead can be reduced using higher values of n_c. However, instead of RAM the final bottleneck became the time to solution due to the limit on execution time of the considered computing platform. The boundaries for multi-solve on a single shared-memory workstation were therefore not reached yet. Out-of-core in the multi-

factorization algorithm had stronger positive impact in the context of the test cases leading to coupled systems with higher proportion of unknowns in the dense part with respect to the sparse part, i.e. in the *narrow pipe* academic test case and further in the industrial test case. Thanks to out-of-core, multi-factorization allowed us to to process larger problems as well as to process problems significantly faster thanks to lower values of n_b leading to fewer factorizations. This effect was especially strong in the context of the industrial test case. Without numerical compression, out-of-core allowed multi-factorization to perform considerably less calls ($n_b = 3$) to the *sparse factorization+Schur* step compared to in-core computation ($n_b = 12$). Interestingly, in such cases, the out-of-core multi-factorization is faster then its in-core counterpart. Moreover, multi-factorization remains faster then multi-solve like in in-core computation (see Section 3.4 in Chapter 3). In Chapter 5, we extend this work to distributed-memory cases.

Chapter 5

Two-stage algorithms in distributed memory

In this thesis, we are interested in the design of coupled solvers for large sparse/dense FEM/BEM linear systems such as (1.2) defined in Section 1.3). In Chapter 2, concretely in Section 2.3, we saw that vanilla couplings of the state-of-the-art sparse and dense direct solvers do not allow us to efficiently exploit advanced features such as numerical compression and out-of-core computation on the articulation between sparse and dense operations, i.e. in the Schur complement part (see Section 1.3). We thus dedicated chapters 3 and 4 to the design of two-stage algorithms, namely multi-solve and multi-factorization, allowing us to cope with this limitation in a shared-memory environment. In this chapter, we pursue our effort to process ever larger coupled FEM/BEM systems by resorting to distributed-memory computation.

However, taking advantage of distributed-memory parallelism in addition to the other advanced features within the solution process is not trivial either. The sparse solver implement the possibility to compute and store the Schur complement in a distributed fashion. This would allow vanilla solver couplings to process larger coupled systems compared to what they could achieve in a shared-memory environment. However, without the possibility to efficiently apply numerical compression nor out-of-core computation locally on different computational nodes, they would quickly run into the same limitations as in shared-memory. Indeed, the distribution of a compressed matrix is not compatible with the distribution of a dense non-compressed matrix returned by the sparse solver. A naive application of the compression routine would thus require for the distributed Schur complement matrix to be copied in its entirety over all of the computational nodes. Moreover, even if the dense A_{ss} submatrix may be stored out-of-core by the dense direct solver, the portions of the Schur complement matrix on each node would be entirely stored in RAM in a non-compressed dense form.

To deal with these limitations and enable a more efficient usage of numerical compression and out-of-core computation in a distributed-memory environment, the goal of this chapter is to present parallel distributed extensions of the multi-solve and the multi-factorization algorithms.

In sections 5.1 and 5.2, we introduce the extended multi-solve and multi-factorization algorithms. In Section 5.3, we then analyze their performance in distributed memory with and without compression of the Schur complement part S. Further experiments then attempt to solve the largest possible coupled systems using the two-stage algorithms on a given amount of computers and using both numerical compression and out-of-core techniques. We conclude

in Section 5.4.

5.1 Parallel distributed multi-solve algorithm

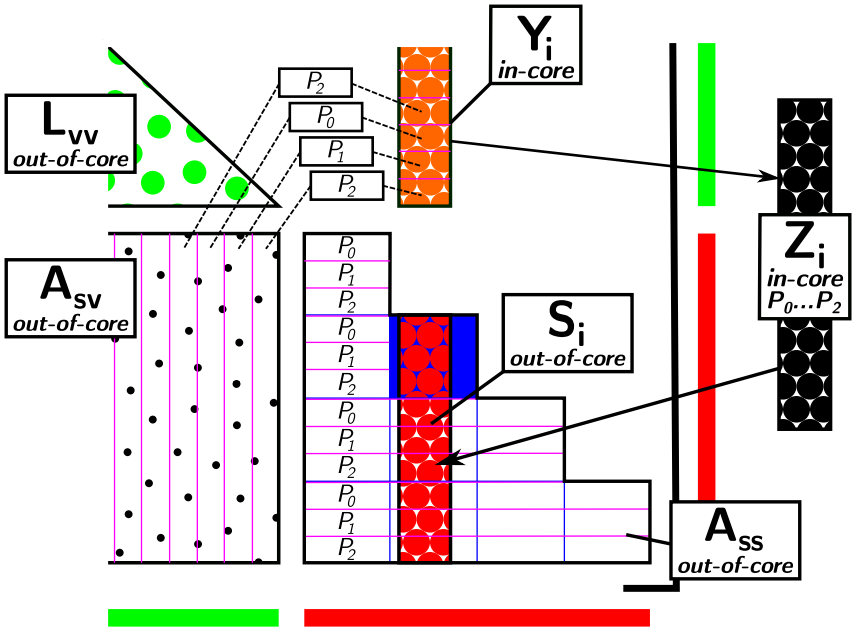
The original *baseline multi-solve* (see Algorithm 3) and *compressed Schur multi-solve* (see Algorithm 4) two-stage algorithms respectively proposed in Sections 3.1.1 and 3.1.2 of Chapter 3 were extended to out-of-core computation in Section 4.1 of Chapter 4. Their extension to distributed-memory parallelism introduced in this section is based on the out-of-core multi-solve algorithms.

In multi-solve, the *sparse factorization* of A_{vv} as well as the *dense factorization* of the Schur complement S, followed by the solve operations to determine x_s and x_v, can be transparently done in parallel in distributed memory by the sparse and the dense direct solvers. Indeed, the corresponding building blocks take care of distributing data and computations. However, in order to be able to perform the multiplication $A_{sv}Y_i$ leading to Z_i (line 5 in Algorithm 3 and line 6 in Algorithm 4), the columns of A_{sv} must be sorted so that on a given process P_k, the indices of local columns of A_{sv} match those of local rows of Y_i (see Figure 5.1). After the multiplication, values from all processes are combined to compute the resulting Z_i which is then sent back to all processes. In the case of *baseline multi-solve*, each out-of-core block of A_{ss} (see Section 4.1) is distributed among parallel processes by packs of rows in a cyclic manner. This way, during the final assembly of S_i, we process one parallel out-of-core block at a time. Note that when the out-of-core feature is turned off, all the blocks are kept in memory but the partitioning and the distribution remain the same. As of *compressed Schur multi-solve*, the distribution of the compressed Z_i and A_{ss} is handled transparently by the dense direct solver through the underlying runtime.

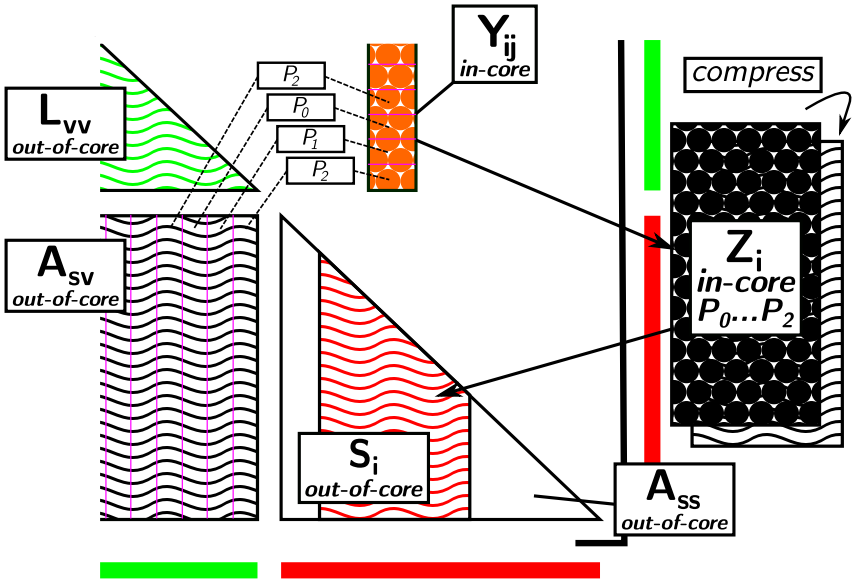
5.2 Parallel distributed multi-factorization algorithm

The original *baseline multi-factorization* (see Algorithm 5) and *compressed Schur multi-factorization* (see Algorithm 6) two-stage algorithms respectively proposed in Sections 3.2.1 and 3.2.2 of Chapter 3 were extended to out-of-core computation in Section 4.2 of Chapter 4. Their extension to distributed-memory parallelism introduced in this section is based on the out-of-core multi-factorization algorithms.

As of multi-factorization, the *sparse factorization+Schur* steps on the temporary W matrices as well as the *dense factorization* of the Schur complement S, followed by the solve operations to determine x_s and x_v, can be transparently done in parallel in distributed memory by the sparse and the dense direct solvers. The corresponding building blocks take care of distributing data and computations. Nevertheless in the *baseline multi-solve* variant, X_{ij} is obtained distributed across parallel processes in a cyclic manner by packs of rows. We choose the size of the latter so as to match the distribution of the out-of-core blocks (see Section 4.2) of A_{ss} (see Figure 5.2a). As in the case of *baseline multi-solve* (see Section 5.1), during the final assembly of S_{ij}, we process one parallel out-of-core block at a time. The algorithm behaves the same way when out-of-core is turned off. Then, all the blocks are simply kept in memory. As of *compressed Schur multi-factorization* (see Figure 5.2b), the distribution of A_{ss} is handled transparently by the dense direct solver through the underlying runtime. Moreover, due to the specificities of the hierarchical low-rank compression technique [101], data distribution in the compressed submatrix A_{ss} is more complex and incompatible with data distribution in the dense non-compressed submatrix X_{ij} following a standard 2D-block cyclic scheme [48]. Consequently, during the assembly of the block S_{ij}, one process may require a portion of the non-compressed



(a) *baseline multi-solve*, blue lines depict the partitioning of A_{ss} into out-of-core blocks and pink lines represent the distribution of A_{sv}, A_{ss} and Y_i among parallel processes.



(b) *compressed Schur multi-solve*, pink lines represent the distribution of A_{sv} and Y_{ij} among parallel processes.

Figure 5.1: Out-of-core parallel distributed (assuming three parallel processes P_0, P_1 and P_2) block-wise computation of S using the multi-solve algorithm (zoom on the Schur complement dense part of the system). Only parts of L_{vv}, A_{sv}, Y_i and Y_{ij} are represented.

X_{ij} located on another process. To accommodate with this constraint, we propose a communication scheme based on a virtual ring topology for the processes to share their local portions of X_{ij} so as to ensure that each process receives each of the local portions of X_{ij} one at a time without having to store a copy of the entire X_{ij} on each process.

Figure 5.3 illustrates this communication scheme on an example assuming three parallel processes P_0, P_1 and P_2. This leads to a three-step assembly of a tile S_{ij} based on a non-compressed tile X_{ij} split into three packs of rows $X_{ij\alpha}$, $X_{ij\beta}$ and $X_{ij\gamma}$ distributed among all processes. At first, P_0 assembles the local part of S_{ij} for the locally available pack of rows $X_{ij\alpha}$. Then, it sends the latter to its immediate neighbor P_1. Meanwhile, it receives $X_{ij\gamma}$ from P_2 and continues the assembly of the local part of S_{ij}. In the last step, P_0 receives $X_{ij\beta}$ from P_2 (which received it from P_1) and completes the assembly of its local part of S_{ij}. The remaining parts of S_{ij} are assembled analogically by P_1 and P_2.

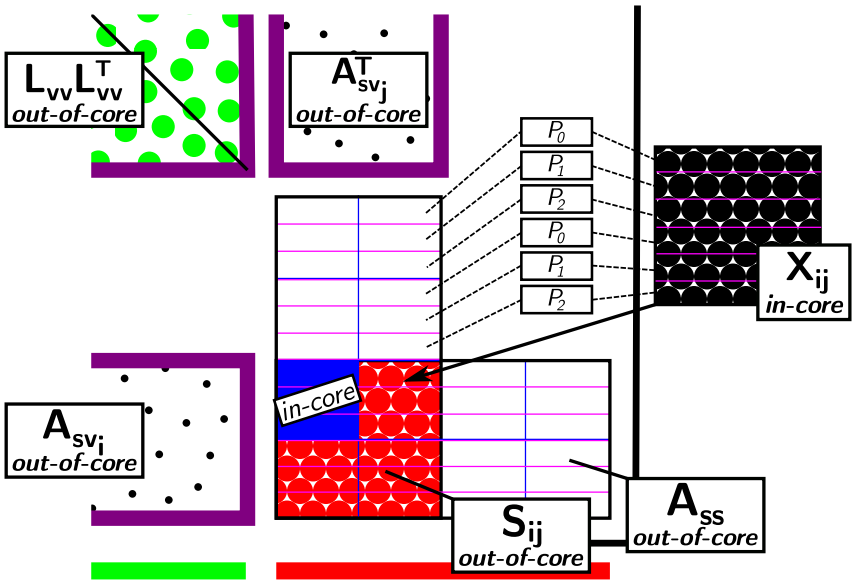
5.3 Experimental results

We conducted an experimental study of the extended multi-solve (see Section 5.1) and multi-factorization (see Section 5.2) algorithms for solving larger coupled sparse/dense FEM/BEM linear systems such as defined in (1.2) in order to evaluate their performance in a distributed-memory environment. For the purpose of this evaluation and based on previous results from chapters 3 and 4, we used the *wide pipe* test case (see Section 1.4.1) for the evaluation of multi-solve and the *narrow pipe* test case (see Section 1.4.1) for the evaluation of multi-factorization. We remind the reader that the test cases are designed so that we can compute the relative error of the numerical solution (see Section 1.5.2.1). Computation time and RAM usage are measured as described in Section 3.3.2, p. 39. For the RAM consumption, we collect the usage peaks from all the computation nodes and report the highest one.

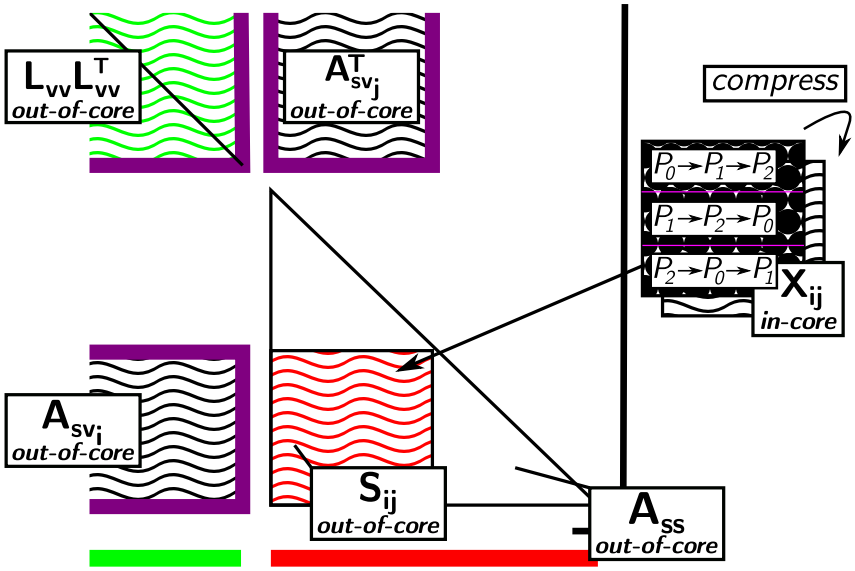
Regarding the implementation of the distributed multi-solve and multi-factorization algorithms, we have extended the shared-memory MUMPS/SPIDO and MUMPS/HMAT solver couplings defined in Section 3.3.1.1. The distributed-memory parallelism is implemented using the OpenMPI communication library [14] based on the MPI standard [79]. We use double precision accuracy and in the case of MUMPS and HMAT, the precision parameter ϵ set to 10^{-3} (see Section 1.5.2.3). Furthermore, we systematically turn on numerical compression in MUMPS for both MUMPS/SPIDO and MUMPS/HMAT (see Section 3.3.1.1).

We have conducted our experiments on `skylake` nodes on the TGCC Irene platform [23]. A `skylake` node has a total of 48 processor cores running each at 2.7 GHz and 180 GiB of RAM. This platform activates Hyper-Threading and Turbo-Boost by default. However, Hyper-Threading is not taken into account by the scheduler unless the user explicitly specifies it, which was not the case for the experiments below. The solver test suite is compiled with GNU C Compiler (gcc) 8.3.0, OpenMPI 4.1.4, StarPU 1.3.8, Intel(R) MKL library 19.0.5.281 and MUMPS 5.5.1. Each run presented below uses one MPI process and 48 threads per node. This choice of parallel configuration is motivated by earlier scalability tests of the solvers involved in the computation. The best-performing setup for the combined use of these solvers was to use one MPI process per computational node and delegate the intra-node parallelism to threads (see Figure 8.11 in the Appendix for MUMPS and Figure 8.17 in the Appendix for SPIDO and HMAT applied on dense matrices).

In the first part of the study (Section 5.3.1), the goal is to evaluate the distributed two-stage algorithms both from the point of view of scalability and parallel efficiency but also to address them in a memory-aware fashion (see Section 3.3.4). We indeed first consider problems that may be processed on a single node to see whether the distributed multi-solve and multi-factorization schemes can take advantage of the available memory and computational



(a) *baseline multi-factorization*, blue lines depict the partitioning of A_{ss} into out-of-core blocks and pink lines represent the distribution of A_{sv}, A_{ss} and X_{ij} among parallel processes.



(b) *compressed Schur multi-factorization*, pink lines represent the distribution of the non-compressed X_{ij} among parallel processes.

Figure 5.2: Out-of-core parallel distributed (assuming three parallel processes P_0, P_1 and P_2) block-wise computation of S using the multi-factorization algorithm (zoom on the Schur complement dense part of the system). Only parts of $L_{vv}L_{vv}^T$, A_{sv_i} and $A_{sv_j}^T$ are represented.

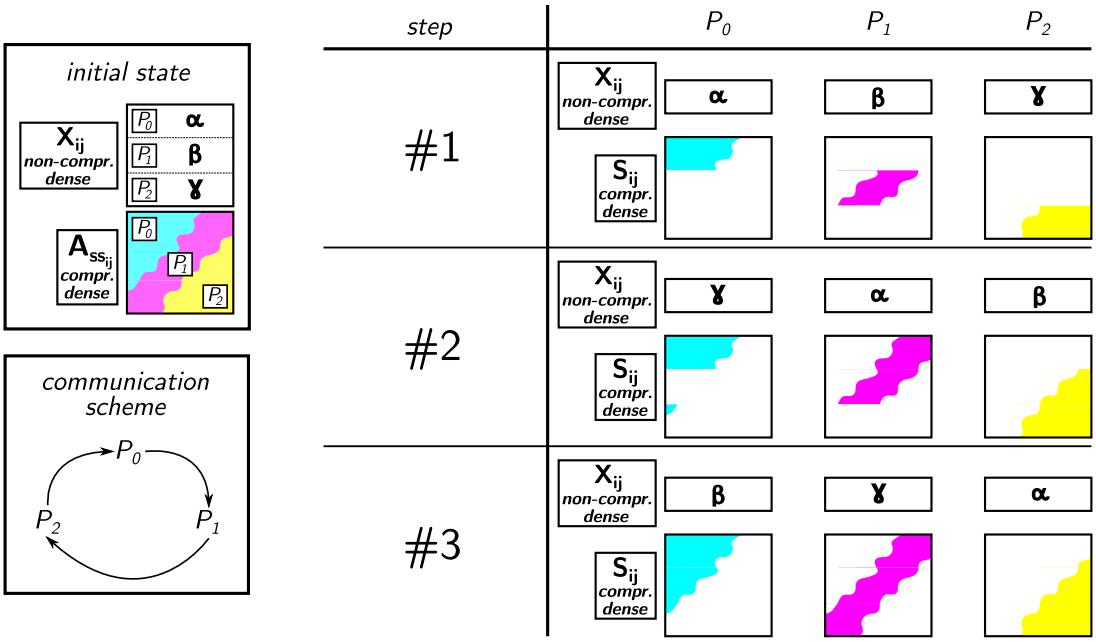


Figure 5.3: Parallel distributed assembly of a Schur complement tile S_{ij} by the *compressed Schur multi-factorization* algorithm (assuming three parallel processes P_0, P_1 and P_2).

resources to increase their performance. In the second part of the study (Section 5.3.2), we consider the two-stage algorithms with both the extension to out-of-core and the extension to parallel distributed computation and try to reach their limits on a given number of computational nodes.

5.3.1 In-core scalability

5.3.1.1 Multi-solve

We consider coupled FEM/BEM linear systems with the total unknown count N equal to 1,000,000, 3,000,000 and 7,000,000. We run benchmarks on 1 to 16 nodes, i.e. using 48 to 768 cores.

baseline multi-solve In the first place, we focus on the *baseline multi-solve* (see Section 3.1.1) relying on the MUMPS/SPIDO coupling. We vary the size n_c of block $A_{sv_i}^T$ of columns of the A_{sv}^T submatrix in between 128 and 4,096.

Figure 5.4 shows the computation time of the MUMPS/SPIDO coupling in function of per-node RAM usage peaks. Then, Figure 5.5 shows the scalability and the parallel efficiency.

Regarding the n_c parameter, we observe analogous behavior like in the case of the experiments in shared memory (see Section 3.3.4.1). Sufficiently high n_c (see Figure 3.1) gives the sparse solver enough right-hand sides to process in parallel during the *sparse solve* step $A_{vv}^{-1}A_{sv}^T$ (line 4 in Algorithm 3, p. 41) and leads to an important improvement in computation time and parallel efficiency. However, as the outcome of the operation is a dense non-compressed matrix, too high values for n_c can result in a dramatic increase in RAM consumption. Moreover, ever higher n_c does not translate into a proportional performance improvement. It may even lead to a degradation. For example, with $N = 7{,}000{,}000$ and 8 computational nodes, going from $n_c = 1{,}024$ to $n_c = 4{,}096$ shortens the computation time by 14% but more than doubles the memory footprint. There is no large gap between the parallel efficiency of these two cases

either. Then, with N = 3,000,000 and 4 to 8 nodes, increasing n_c from 2,048 to 4,096 even worsens the computation time in addition to a higher RAM consumption. In the end, the optimal value of n_c then depends on the size of the problem and the number of parallel processing units used for the computation.

From a more general point of view, with increasing number of nodes, we can reduce both the memory footprint as well as the computation time of the multi-solve algorithm. This is crucial for our effort to process ever larger coupled FEM/BEM systems. In this part of the study, we limited ourselves to at most 7,000,000 of unknowns. For systems of this size, 4 to 8 computational nodes seem to deliver satisfying efficiency both in terms of time and RAM consumption. Pushing up to 16 nodes leads to a relatively small improvement. However, in Section 5.3.2, we then demonstrate the ability of the algorithm to take advantage of the available memory and computational power of the 16 nodes to process considerably larger coupled systems.

compressed Schur multi-solve For the *compressed Schur multi-solve* (see Section 3.1.2), using the MUMPS/HMAT solver coupling, the size of blocks of columns of S and A_{sv}^T is handled by two different parameters, n_S and n_c, respectively (see Figure 3.2). For the experiments in shared memory (see Section 3.3.4.1), the optimal value of n_c for the MUMPS/HMAT coupling seemed to be 256 for a single problem with 2,000,000 unknowns. Here, we consider systems of varying size and on varying number of nodes. The best n_c setting will therefore not be the same for every case. For the evaluation of multi-solve using the MUMPS/HMAT coupling, we therefore set n_S to multiples (1×, 2× or 4×) of n_c instead of considering a fixed value. For example, according to the results in Figure 5.4, the optimal n_c for the problem with 3,000,000 of unknowns running on 8 nodes seems to be 2,048. For the equivalent MUMPS/HMAT benchmark, we thus set n_c to 2,048 and n_S to 2,048 ($1\times n_c$), 4,096 ($2\times n_c$) or 8,192 ($4\times n_c$).

Figure 5.6 then compares the computation times and maximum RAM usage peaks of multi-solve using MUMPS/HMAT with varying n_S to the corresponding best-performing runs relying on MUMPS/SPIDO. Figure 5.7 completes the analysis with parallel efficiency measures.

At first, we focus on the impact of the varying n_S parameter. The computation times between different values of n_S present only minor differences. Like in shared memory (see Section 3.3.4.1), high enough n_S reduces the overhead of compressing the dense Z_i blocks as well as of the corresponding compressed AXPY $S_i = A_{ss_i} - Z_i$ (lines 8 and 9 in Algorithm 4, p. 42) during the Schur complement assembly. The scalability and parallel efficiency results corroborate this observation. There is no visible impact of varying n_S on RAM consumption either.

In the second time, we compare the MUMPS/HMAT runs to their MUMPS/SPIDO references. As of the computation time, the compression of S does not bring significant improvements. In some cases MUMPS/HMAT is slightly faster than MUMPS/SPIDO, in some other cases it is the opposite. These differences have two main causes. First, our application presents a non-negligible execution time variability, even in shared memory (see Section 6.2.4). Second, the way MUMPS distributes computation load and data varies and impacts the time to solution differently from one execution to another, even if we consider the exact same linear system. As of the RAM usage of MUMPS/HMAT compared to MUMPS/SPIDO, the compression of S brings an important reduction of the application's memory footprint. On smaller problems, i.e. for N up to 3,000,000, this phenomenon is noticeable especially for low numbers of computational nodes, i.e. 1 or 2. However, for larger problems, i.e. starting with N at 7,000,000, the positive impact of compressing S is present even for higher numbers of nodes, i.e. up to 16.

Eventually, figures 5.8 and 5.9 show the relative error for the test cases featured in figure 5.4 and 5.6. The precision parameter ϵ was set to 10^{-3} for both MUMPS and HMAT solvers pro-

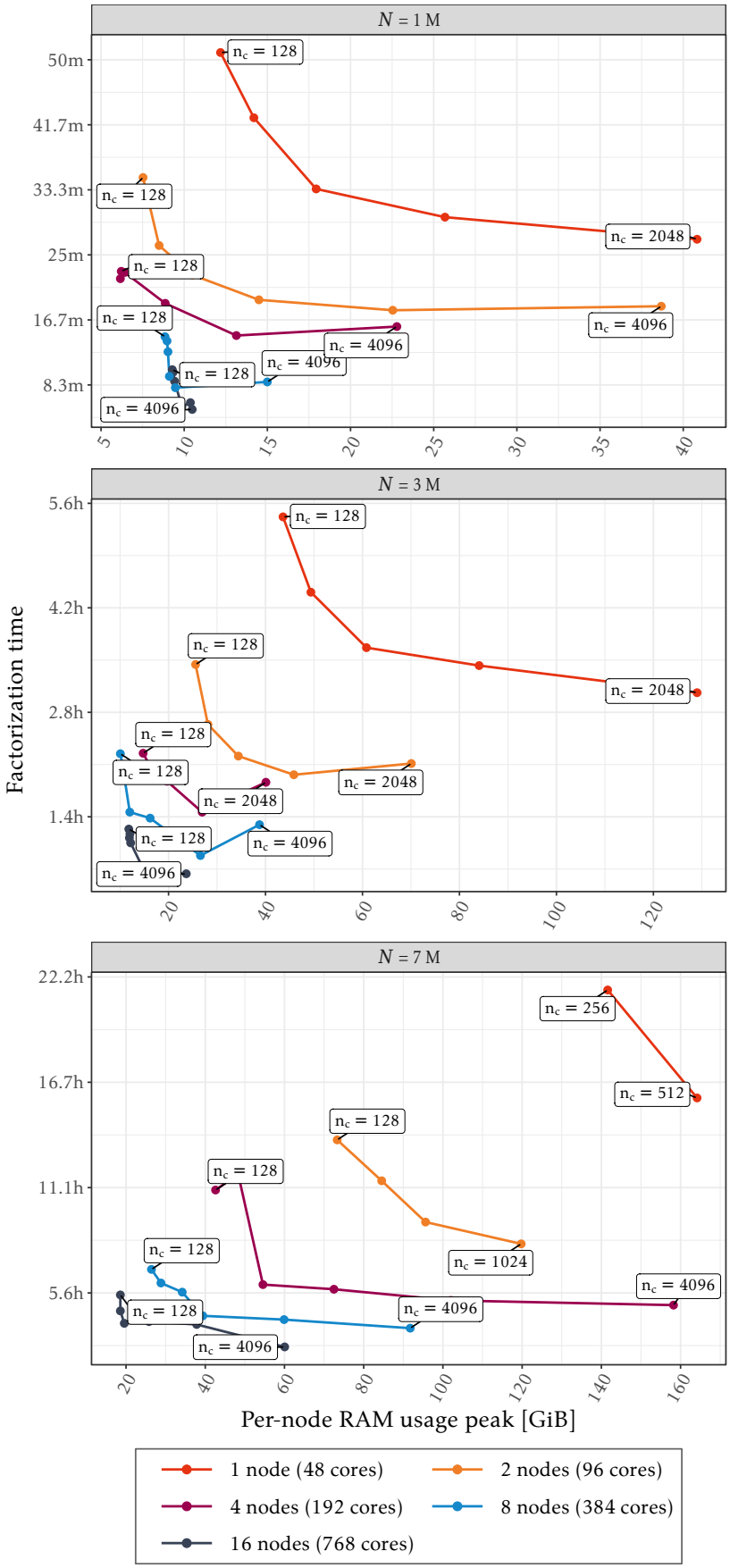


Figure 5.4: Computation times of **multi-solve** with MUMPS/SPIDO on FEM/BEM linear systems of varying size N and for varying values of n_c. Parallel runs on 1 to 16 `skylake` nodes using 48 threads per node.

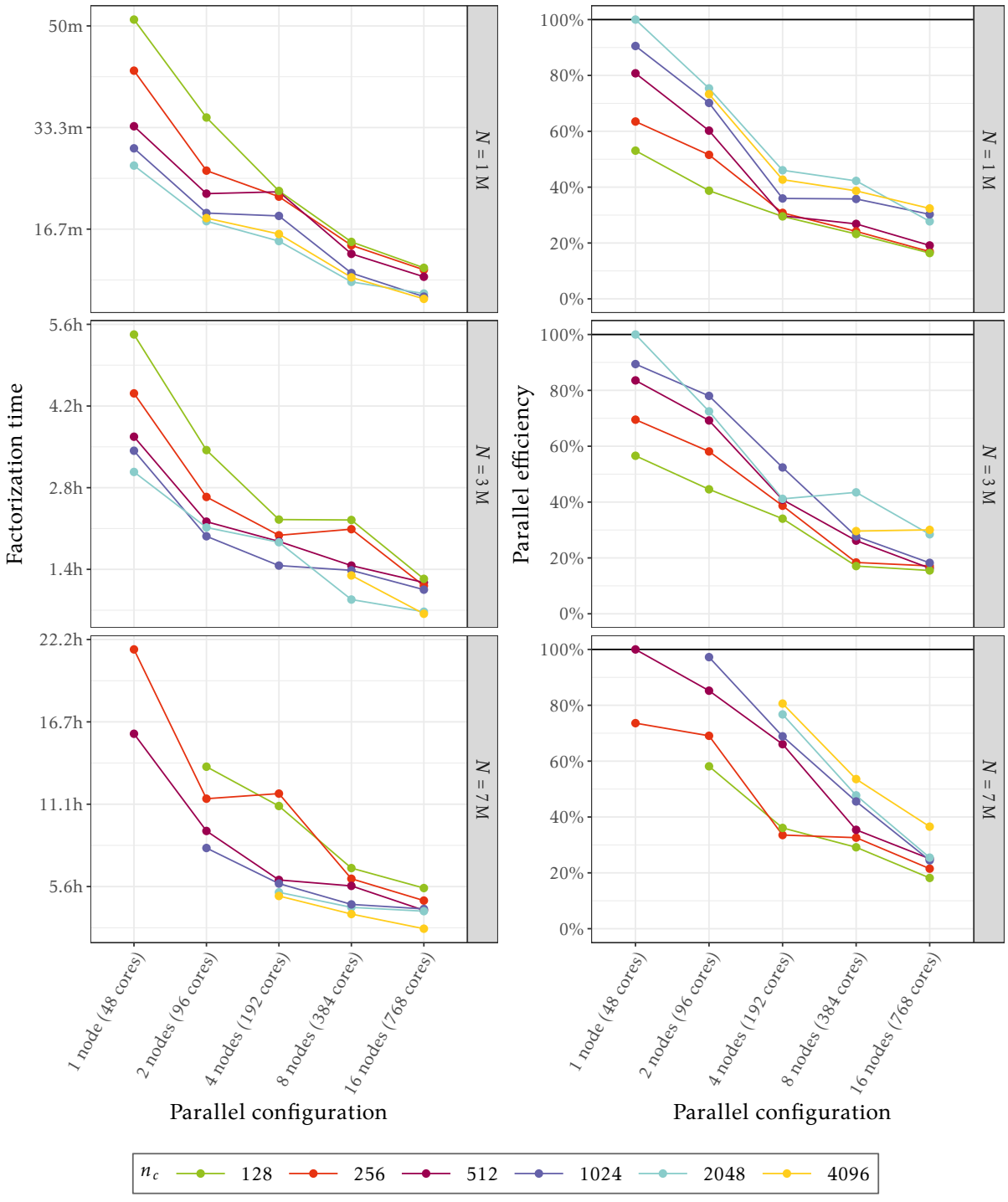


Figure 5.5: Scalability and parallel efficiency of **multi-solve** with MUMPS/SPIDO on FEM/BEM linear systems of varying size N and for varying values of n_c. Parallel runs on 1 to 16 `skylake` nodes using 48 threads per node.

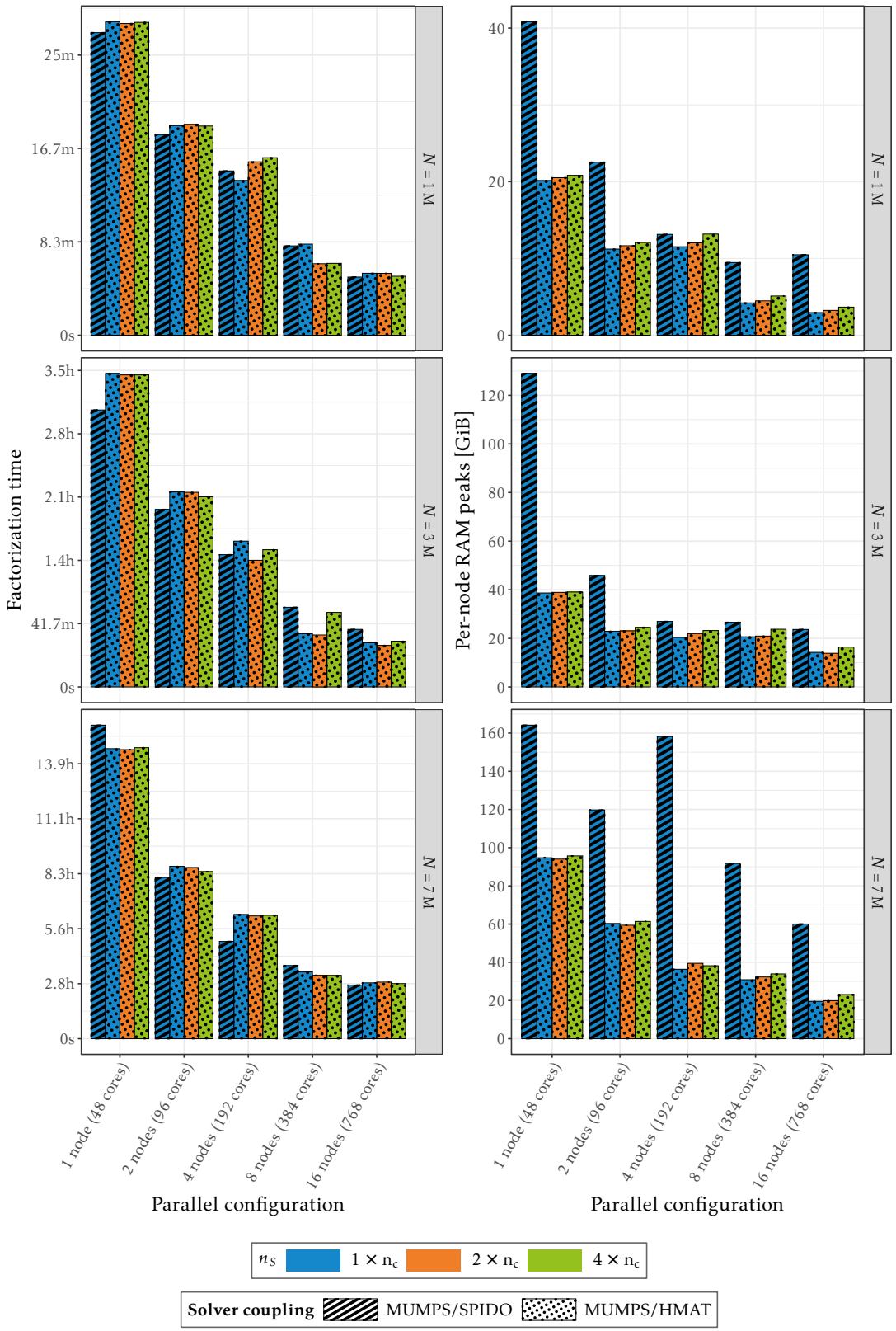


Figure 5.6: Computation times and per-node RAM usage peaks of **multi-solve** with MUMPS/HMAT on FEM/BEM linear systems of varying size N and for varying values of n_S. For each parallel configuration (x-axis) and problem size N, the starting value of n_c is based on the best-performing MUMPS/SPIDO execution (striped bars) with the same N and the same parallel configuration. The value of n_S for MUMPS/HMAT (dotted bars) represent multiples (1×, 2× and 4×) of this base n_c value. Parallel runs on up to 16 `skylake` nodes.

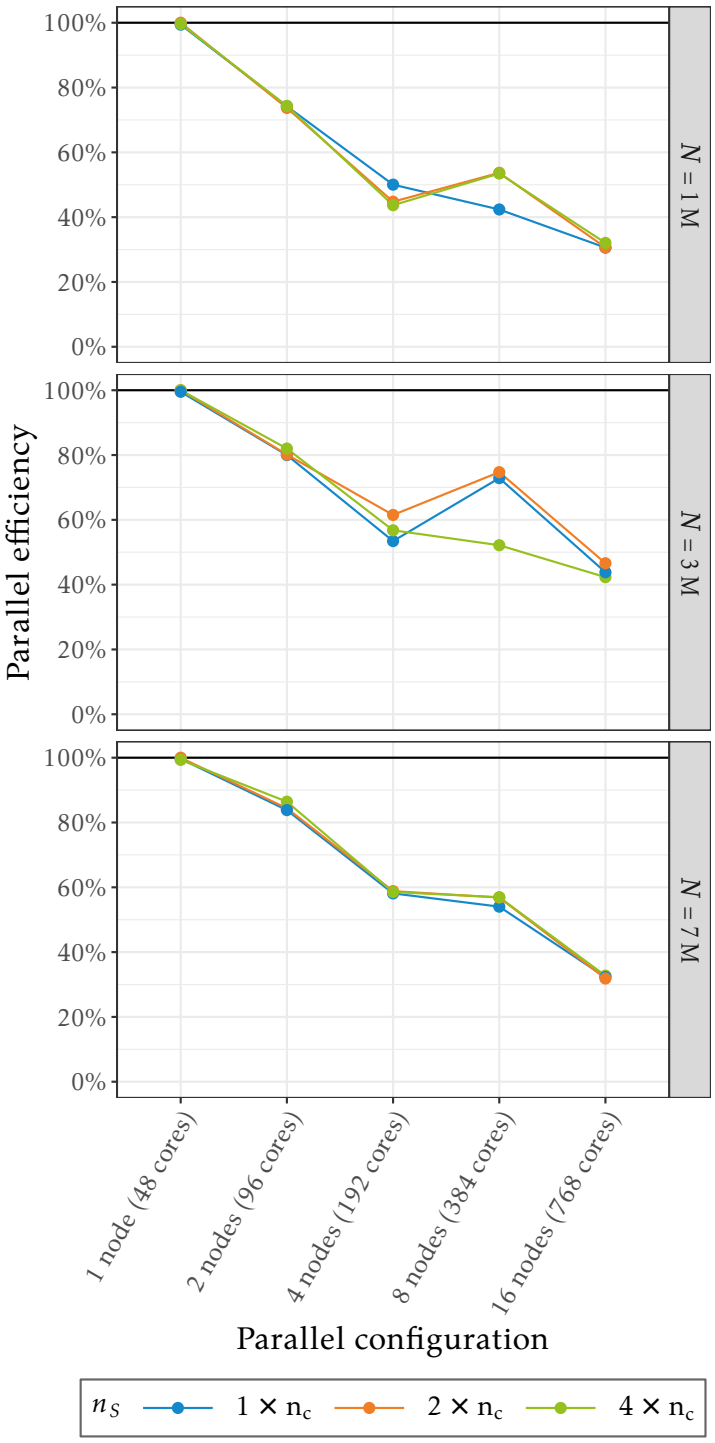


Figure 5.7: Parallel efficiency of **multi-solve** with MUMPS/HMAT on FEM/BEM linear systems of varying size N and for varying values of n_S. Parallel runs on 1 to 16 `skylake` nodes using 48 threads per node.

viding low-rank compression. Compared to the fully compressed test cases relying on the MUMPS/HMAT coupling, the relative error is smaller in the case of MUMPS/SPIDO when the dense part of the linear system is not compressed at all. Consequently, the final result of the computation suffers less from the loss of accuracy due to the compression. In all cases, the relative error is below the selected threshold 10^{-3} which confirms that the algorithm allows us to reach the expected accuracy.

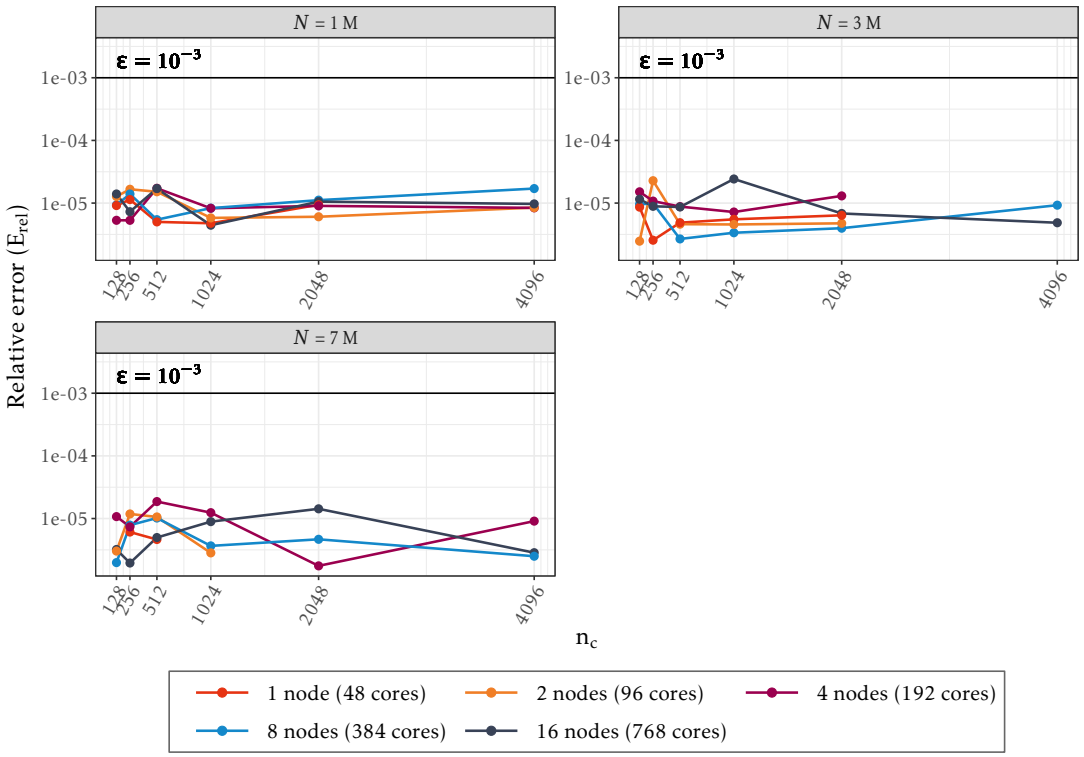


Figure 5.8: Relative error (E_{rel}) of **multi-solve** with MUMPS/SPIDO on FEM/BEM linear systems of varying size N and for varying values of n_c. Parallel runs on 1 to 16 `skylake` nodes using 48 threads per node.

5.3.1.2 Multi-factorization

In this section, we involve coupled FEM/BEM linear systems (arising from the *narrow pipe* test case) with N, the total unknown count, of 1,000,000, 2,000,000 and 3,000,000. We run benchmarks on 1 to 16 nodes, i.e. using 48 to 768 threads. Both the *baseline multi-factorization* (see Section 3.2.1) and the *compressed Schur multi-factorization* variants (see Section 3.2.2) expose the n_b parameter handling the count of square blocks S_{ij} per block row and block column of the Schur complement submatrix S. The tested values of n_b are 1, 3, 7 and 11. Note that, from the implementation point of view, running *baseline multi-factorization* (MUMPS/SPIDO) with n_b set to 1 (blue curves with round points in the figures below) corresponds to the advanced vanilla coupling from the state of the art (see definition in Section 2.2.2 and implementation details in Section 3.3.1.2). These results serve as reference points in the present study.

Figure 5.10 shows the computation time as a function of per-node RAM usage peaks for both the MUMPS/SPIDO and the MUMPS/HMAT couplings. Again, the behavior of the algorithm with respect to the n_b parameter is analogous to the one observed in the case of the experiments in shared memory (see Section 3.3.4.2). The smaller the value of n_b and the lower the number of calls to the *sparse factorization+Schur* building block, causing the superfluous and costly re-factorization of the large sparse A_{vv} submatrix within W. To optimize the perfor-

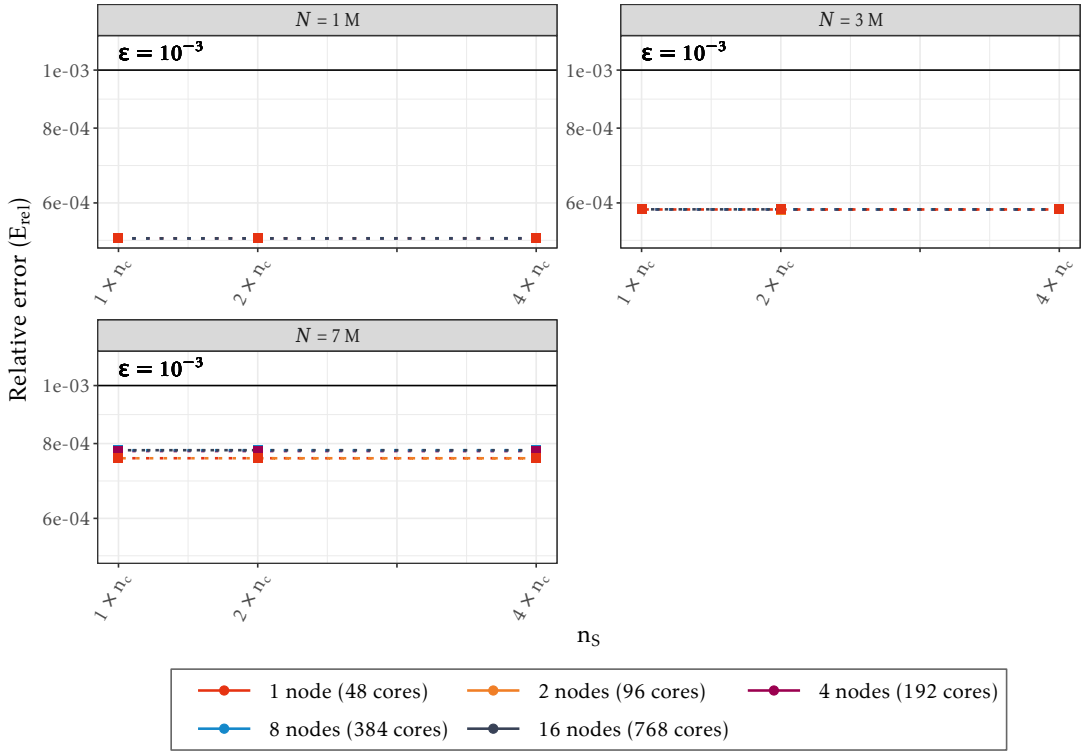


Figure 5.9: Relative error (E_{rel}) of **multi-solve** with MUMPS/HMAT on FEM/BEM linear systems of varying size N and for varying values of n_S. Parallel runs on 1 to 16 `skylake` nodes using 48 threads per node.

mance, the goal is therefore to lower n_b as much as possible before reaching the memory limit. For example, with N = 2,000,000 on one node, the computation time of multi-factorization with n_b = 3 (leading to 6 calls to *sparse factorization+Schur*) is more than eight times lower than with n_b = 11 (leading to 66 calls to *sparse factorization+Schur*). However, unlike in the case of distributed multi-solve (see Section 5.3.1.1), computation time of multi-factorization decreases only slightly with increasing number of nodes. This is especially noticeable for small values of n_b. The results of scalability and parallel efficiency in figures 5.11 and 5.12, respectively, confirm this observation. For example, considering 2,000,000 of unknowns, for both MUMPS/SPIDO and MUMPS/HMAT, the computation time remains almost constant for all the parallel configurations, when $n_b < 7$. Given that this happens in particular when n_b is 1, i.e. in the supposedly optimal performance condition, indicates a possible issue in the *sparse factorization+Schur* building block of the sparse solver. We further discuss this in the multi-metric study of the two-stage algorithms (see Section 6.2.3.2 in Chapter 6). The parallel efficiency of multi-factorization therefore remains low, i.e. below 20%.

Moreover, the results show that the per-node memory footprint of the distributed multi-factorization algorithm can be reduced when using a higher number of nodes. The larger the problem the more important the decrease. For example, in the case of a coupled system with 3,000,000 unknowns and n_b set to 3 blocks, the maximum RAM peak on 4 nodes is of approximately 125 GiB whereas on 16 nodes it is around 55 GiB. Compression of the dense Schur complement part in the MUMPS/HMAT coupling further favors this effect. However, this becomes less noticeable with increasing number of nodes. Indeed, the reduction of memory footprint brought by the compression is already very important. This way, distributing small amount of data for computation over a too large amount of nodes leads to duplication and, in terms of RAM usage, penalizes MUMPS/HMAT which looses its advantage over MUMPS/SPIDO. At the end, thanks to distributed-memory parallelism, multi-factorization allows us not only to

process larger problems but also to use smaller values of n_b compared to a single-node execution. This leads to faster processing of problems that can be treated on one node but only when considering large n_b values leading to poor performance. For example, let us consider the case of the system with 3,000,000 unknowns. Using MUMPS/SPIDO, it could not even be processed on one node. Using MUMPS/HMAT, the lowest possible n_b was 7, leading to 28 calls to the *sparse factorization+Schur* building block. Increasing the number of nodes to 4 allowed us to process this case with $n_b = 3$ and even with $n_b = 1$ which is the optimal situation.

Figures 5.13 and 5.14 then show the relative error for the test cases featured in Figure 5.10. The precision parameter ϵ was set to 10^{-3} for both MUMPS and HMAT solvers providing low-rank compression. The observations are the same as in the case of multi-solve (see Section 5.3.1.1). In summary, the relative error is below the selected threshold 10^{-3} which confirms that the algorithm allows us to reach the expected accuracy.

5.3.2 Solving larger problems

The goal of this second part of the experimental evaluation is to push the distributed multi-solve and multi-factorization algorithms to their limits on a given number of nodes and considering both the numerical compression and the out-of-core computation extensions proposed in chapters 3 and 4, respectively. In this case, we consider 16 `skylake` nodes and increase the total unknown count N of the problem until either the memory or the execution time limit on the platform is reached. As in the rest of this study, we consider both the *baseline* as well as the *compressed Schur* variants of the two-stage algorithms relying on the MUMPS/SPIDO and MUMPS/HMAT coupling, respectively. The tested values of the n_c and n_S parameters for multi-solve and of the n_b parameter for multi-factorization were analogous to those from the corresponding sections 5.3.1.1 and 5.3.1.2. As a reference from the state of the art, we consider the advanced vanilla solver coupling (see Section 2.2.2). From the implementation point of view, we rely on the *baseline multi-factorization* algorithm with the MUMPS/SPIDO coupling and n_b set to 1 (see further details in Section 3.3.1.2).

In Figure 5.15, we show the best computation times for all of the evaluated configurations and problem sizes. Figure 5.16 then shows the corresponding maximum RAM usage peaks. The algorithm allowing us to process the largest coupled sparse/dense FEM/BEM system is multi-solve for N as high as 40,000,000 unknowns in total. In the case of the *compressed Schur multi-solve*, this result was reached without necessity for out-of-core in the Schur complement part of the system. As we showed in Chapter 3, compressing the Schur complement part (see Algorithm 4, p. 42) is very efficient by itself and the usage of out-of-core has thus only a limited effect. On the contrary, in the case of the *baseline multi-solve*, we had to resort to out-of-core in all the parts of the system to prevent running out of memory. However, none of the variants reached the physical memory limit. Therefore, they could possibly process even larger FEM/BEM systems without the execution time restrictions of the platform, i.e. 3 days. With multi-factorization, we could reach 7,000,000 unknowns. Here, the numerical compression, out-of-core or distributed-memory computation did not allow us to push significantly further than what is possible to achieve with the state-of-the-art advanced vanilla coupling (see Section 2.2.2), i.e. 6,000,000 unknowns. Indeed, in this case, the bottleneck is not the storage of the Schur complement part but the superfluous storage and the loss of symmetry in the multi-factorization algorithm (see Section 3.2).

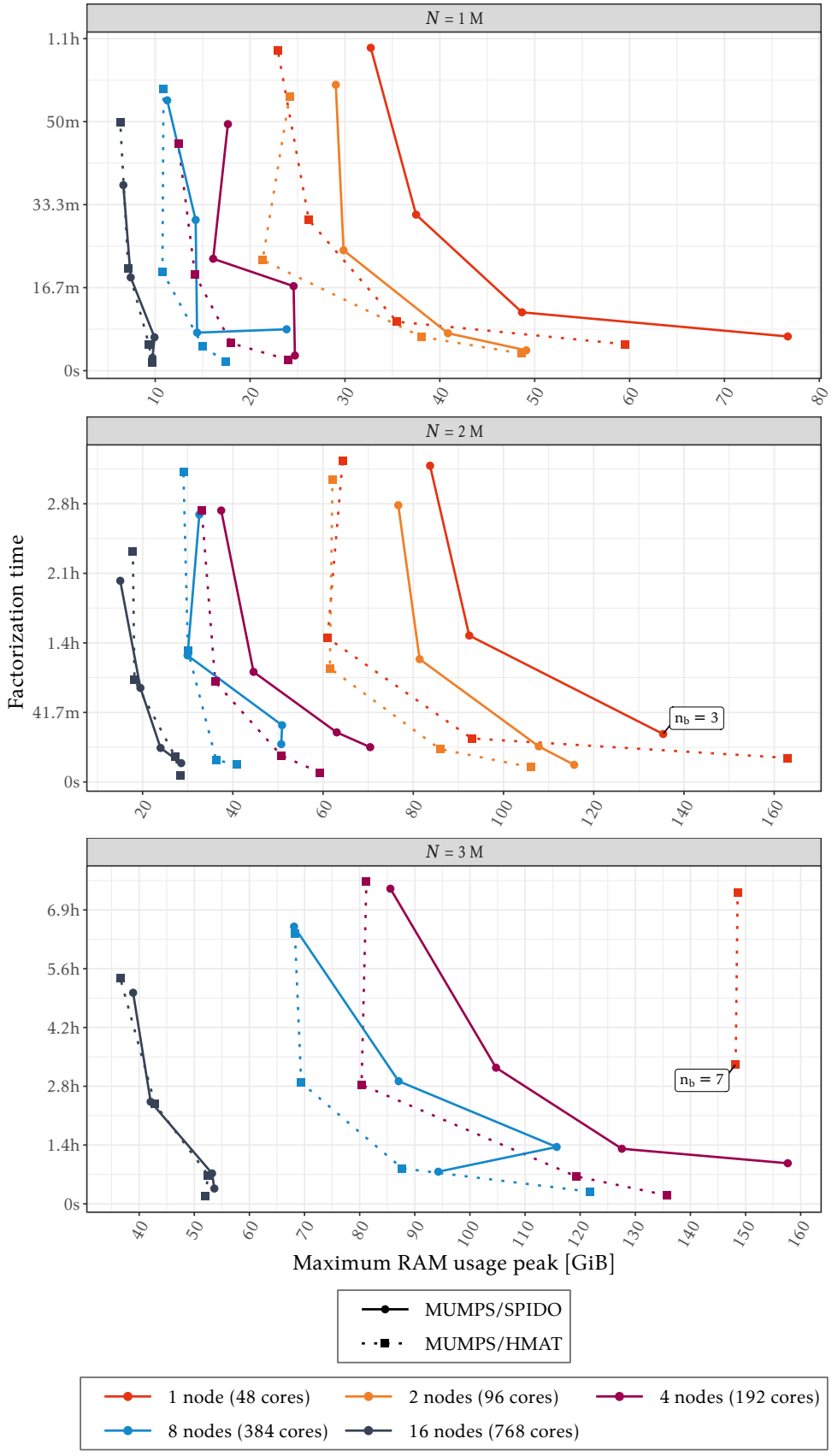


Figure 5.10: Computation times of **multi-factorization** with MUMPS/SPIDO and MUMPS/HMAT couplings on FEM/BEM linear systems of varying size N and for n_b from 11 to 1 or until the RAM limit is reached. Labels indicate the lowest n_b we could achieve. Parallel runs on 1 to 16 `skylake` nodes using 48 threads per node.

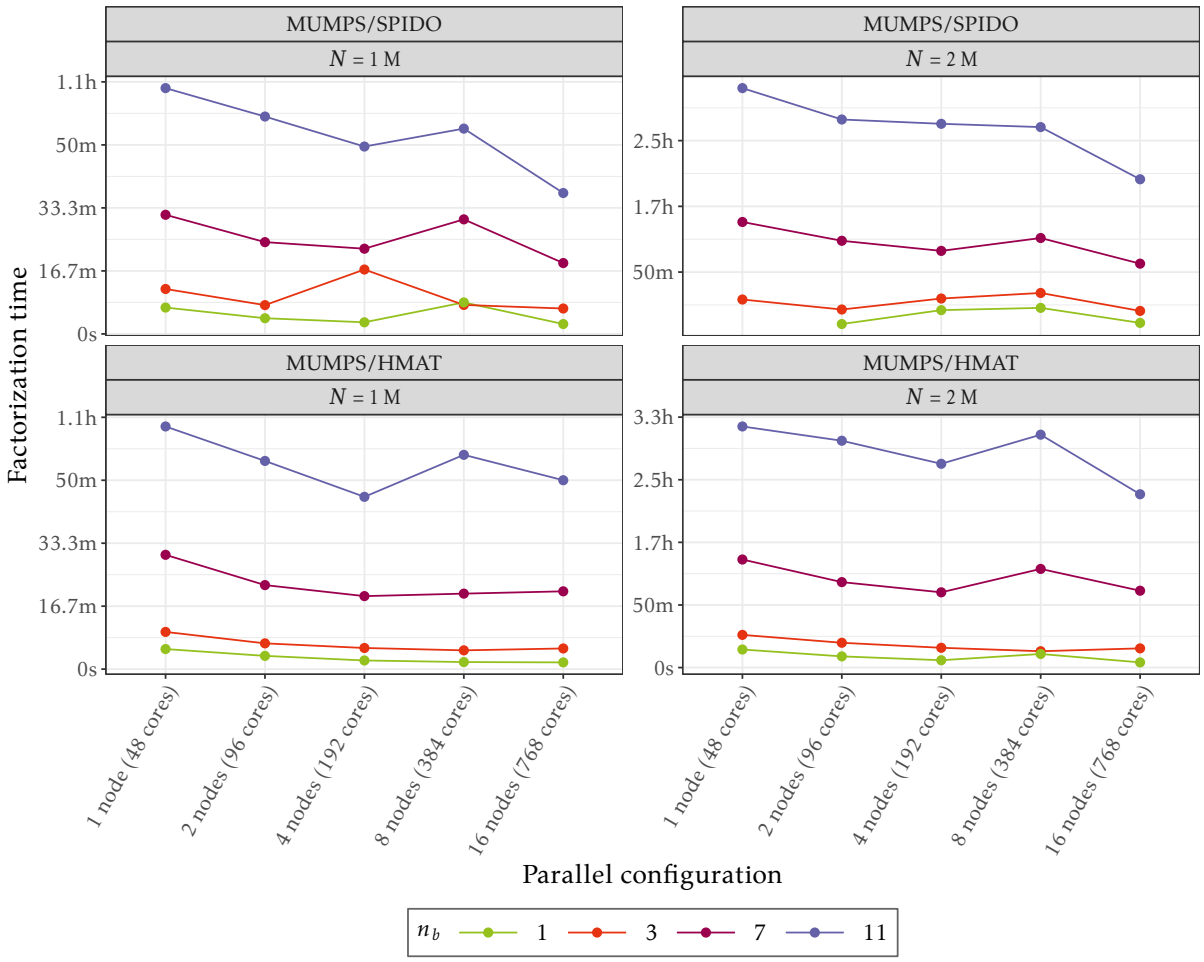


Figure 5.11: Scalability of **multi-factorization** for both the MUMPS/SPIDO and MUMPS/HMAT couplings on FEM/BEM linear systems of varying number of unknowns and for varying values of n_b. Parallel runs on 1 to 16 `skylake` nodes using 48 threads per node.

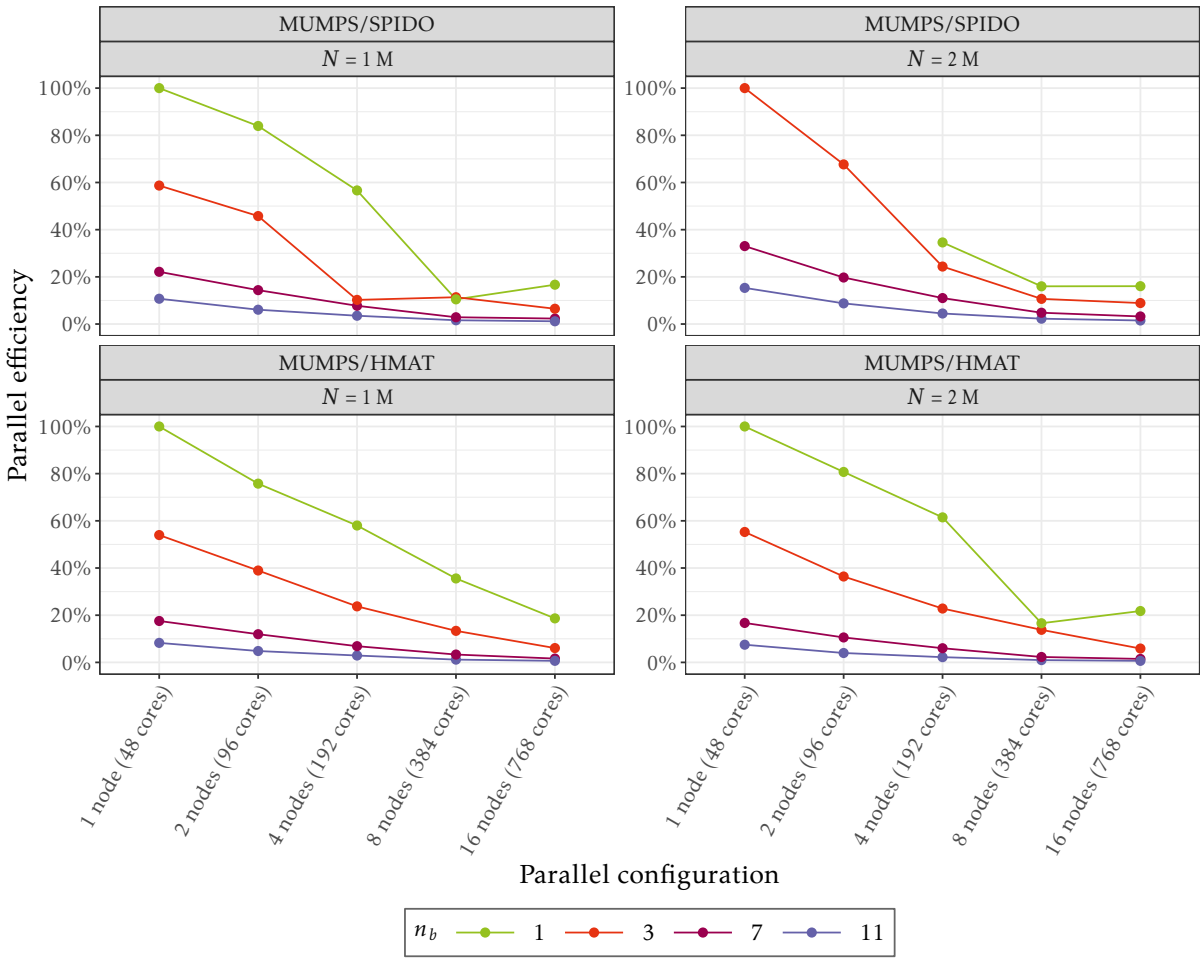


Figure 5.12: Parallel efficiency of **multi-factorization** for both the MUMPS/SPIDO and MUMPS/HMAT couplings on FEM/BEM linear systems of varying number of unknowns and for varying values of n_b. Parallel runs on 1 to 16 `skylake` nodes using 48 threads per node.

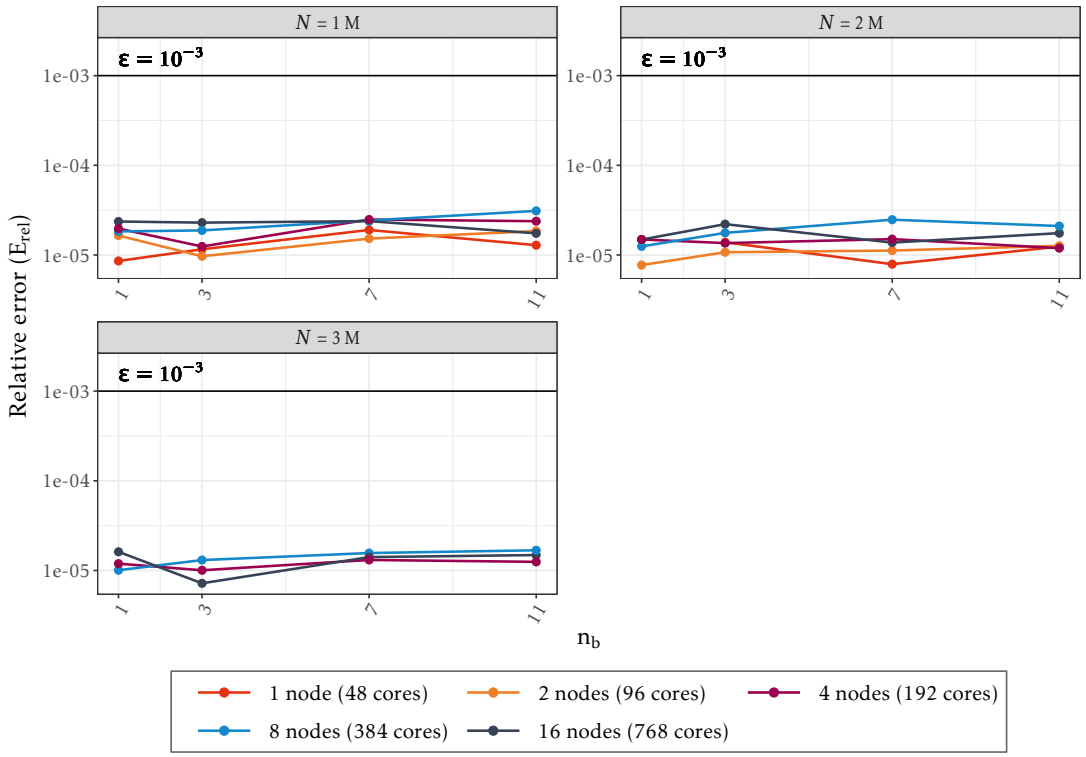


Figure 5.13: Relative error (E_{rel}) of **multi-factorization** with MUMPS/SPIDO on FEM/BEM linear systems of varying size N and for varying values of n_b. Parallel runs on 1 to 16 `skylake` nodes using 48 threads per node.

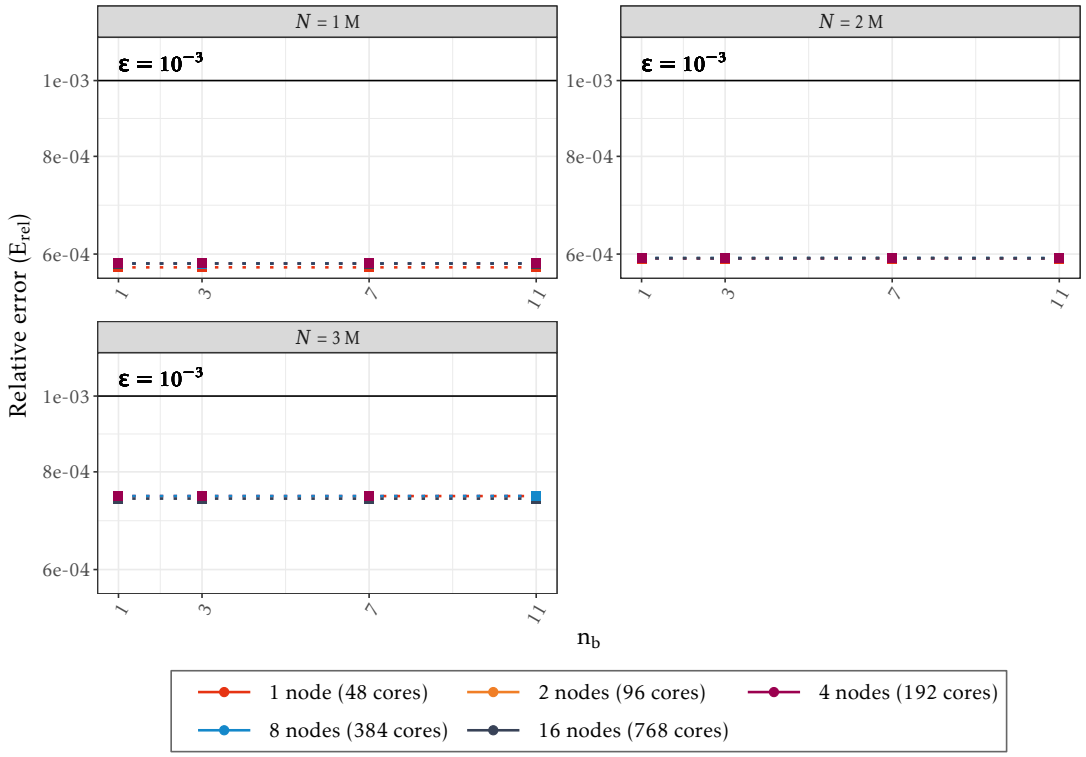


Figure 5.14: Relative error (E_{rel}) of **multi-factorization** with MUMPS/HMAT on FEM/BEM linear systems of varying size N and for varying values of n_b. Parallel runs on 1 to 16 `skylake` nodes using 48 threads per node.

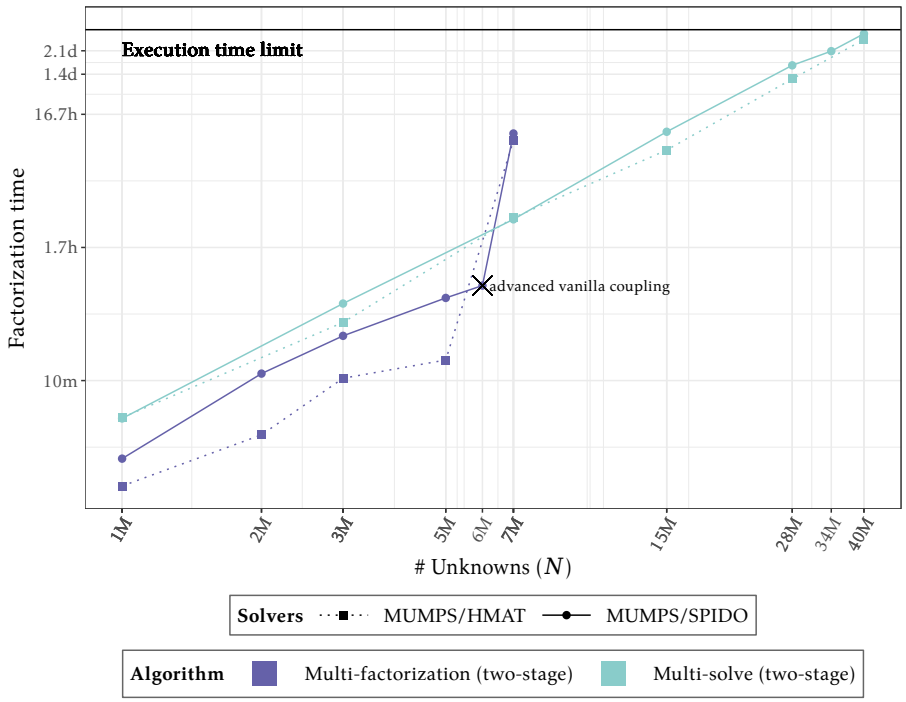


Figure 5.15: Best computation times of **multi-solve** and **multi-factorization** for both MUMPS/HMAT and MUMPS/SPIDO with out-of-core optionally enabled. The state-of-the-art is represented by the advanced vanilla coupling. Parallel runs using 768 threads on 16 `skylake` nodes.

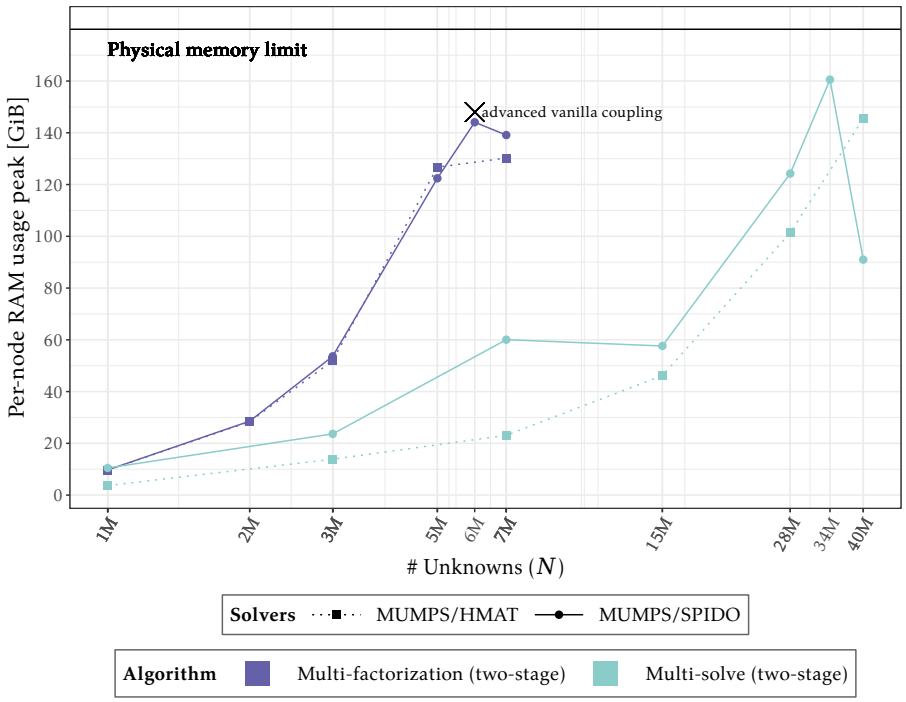


Figure 5.16: Per-node RAM usage peaks, corresponding to the results in Figure 5.15, of **multi-solve** and **multi-factorization** for both MUMPS/HMAT and MUMPS/SPIDO with out-of-core optionally enabled. The state-of-the-art is represented by the advanced vanilla coupling. Parallel runs using 768 threads on 16 `skylake` nodes.

5.4 Conclusion

Following the implementation of out-of-core techniques in the two-stage multi-solve and multi-factorization algorithms in Chapter 4, we proposed their extension to distributed-memory parallelism in this chapter. The subsequent experimental study showed that both of the algorithms can make use of the available memory to increase their performance and allow for processing larger coupled FEM/BEM systems compared to their shared-memory counterparts as well as to the state of the art. The experiments further revealed that the multi-solve algorithm is also able to take advantage of an increasing number of parallel computational nodes to accelerate the computation. This was not the case of multi-factorization. The issue might be coming from its core component, i.e. the *sparse factorization+Schur* building block of the sparse direct solver, but a deeper investigation is required to confirm or contradict this hypothesis. Nevertheless, compared to the state of the art, we were eventually able to process more than 6.5× larger (40,000,000 unknowns against 6,000,000) coupled systems thanks to the proposed two-stage approach without even going out of memory. Without the 3-day execution time limit on the target platform, we might be able to process even larger coupled systems.

The next step of this study would be to deploy the parallel distributed versions of multi-solve and multi-factorization at Airbus. They could allow for processing of more demanding industrial simulations. Also, the multi-factorization scheme may appear much more interesting for certain configurations as we saw in chapters 3 and 4.

Finally, the multi-metric study of both multi-solve and multi-factorization presented in Chapter 6, further analyzes the performance but also the energy consumption as well as the flop rate of the algorithms for a more complete insight on their behavior in both shared-memory and distributed-memory parallel environments.

Chapter 6

Multi-metric study of two-stage algorithms

In Chapter 3, we introduced the two-stage algorithms, namely multi-solve and multi-factorization, for solving coupled FEM/BEM systems such as (1.2) defined in Section 1.3. Compared to the state-of-the-art approaches (see Section 2.2), they allow for processing larger problems thanks to a more efficient usage of numerical compression (see Chapter 3), out-of-core computation (see Chapter 4) and distributed memory parallelism (see Chapter 5) through the coupling of fully-featured and well-optimized sparse and dense direct solvers (see Section 2.6).

Until now, the dimensioning variables for such a calculation were the computation time, the consumed RAM and disk space. The study of these quantities allowed us to understand the behavior of the software and to push its limits in order to handle larger cases. The consideration of carbon footprint issues in industry in general and in computing centers in particular leads both players to consider another physical dimension: the energy consumption of computations. In this chapter, we present an energy profile of the solution of a coupled FEM/BEM linear system. We assess the total energy consumption of the solver as well as how the power consumption varies with the computation time, the flop rate, the amount of memory used and the available algorithmic choices. We consider both shared-memory multi-core machines and small clusters of such nodes, typically used for relatively large problems in aeroacoustic industry. We want to establish whether the improvements brought by the proposed two-stage algorithms in terms of time to solution and memory usage also translate to the energy consumption, detect potential performance bottlenecks and share a detailed analysis of the resulting profile with the community.

In Section 6.1, we present related work on the analysis of the energy consumption in the context of numerical simulation in general and numerical linear algebra in particular. The hardware, software and instrumentation setup we employ for this study is detailed in Section 6.2. The energy profile of the multi-solve and multi-factorization algorithms on a shared-memory node is then analyzed in sections 6.2.1 and 6.2.2, respectively, before being prolonged to a multi-node context in Section 6.2.3. The overhead of the software probes used is studied in Section 6.2.4. We discuss the experimental results in Section 6.3 and conclude in Section 6.4.

6.1 Related work

While many studies like [57, 122] analyze the energy consumption of various applications on different architectures, fewer studies focus on dense or sparse solvers. [43] presents the energy consumption of OpenMP runtime systems on three dense linear algebra kernels. [33] and [42] focus on sparse solvers. While the former studies the behavior of the Conjugate Gradient method on different CPU-only architectures, the latter focuses on sparse solvers on heterogeneous architectures. In [41], the authors present an energy and performance study of state-of-the-art sparse matrix solvers on GPUs. Note that many studies have been conducted regarding the improvement of the energy consumption of sparse or dense linear algebra algorithms [40, 25]. In our case, we analyze an application coupling both sparse and dense techniques.

6.2 Experimental results

We conducted an energy consumption study of the two-stage multi-solve and multi-factorization algorithms (see Chapter 3) for solving larger coupled sparse/dense FEM/BEM linear systems such as (1.2) involving also measures of memory usage and flop rate. For the purpose of this evaluation, we used the *wide pipe* test case (see Section 1.4.1).

We rely on the same implementation of the multi-solve and multi-factorization algorithms as in Section 5.3 on top of the MUMPS/SPIDO and MUMPS/HMAT solver couplings. We use double precision accuracy and in the case of MUMPS and HMAT, the precision parameter ϵ set to 10^{-3} (see Section 1.5.2.3). We systematically turn on numerical compression in MUMPS for both MUMPS/SPIDO and MUMPS/HMAT (see Section 3.3.1.1). However, the out-of-core extension of multi-solve and multi-factorization (see Chapter 4) is not considered here and the corresponding feature in MUMPS, SPIDO and HMAT was disabled for all the benchmarks.

The experiments were carried on the PlaFRIM platform [19] where we used the `miriel` computing nodes equipped with 2 × 12-core Haswell Intel(R) Xeon(R) E5-2680 v3 at 2.5 GHz with a Thermal Design Power (TDP) of 120 W, Hyper-Threading and Turbo-Boost deactivated, 128 GiB (5.3 GiB/core) RAM bank at 2933 MT/s, an OmniPath 100 Gbit/s and a 10 Gbit/s Ethernet network links. Note that TDP refers to the power consumption under the maximum theoretical load [24]. The solver test suite is compiled with GNU C Compiler (gcc) 9.4.0, OpenMPI 4.1.1, StarPU 1.3.8, Intel(R) MKL library 2019.1.144, and MUMPS 5.2.1.

Measurements of computation time and RAM usage are done as described in Section 3.3.2, p. 47. In this study, the reported 'Execution time' covers the overall computation time of the application. We measure the power consumption of our application with `energy_scope`, a software package dedicated to creating energy profiles of HPC codes [4]. It has an acquisition and statistics delivering module running on the cluster and a post-processing and data analysis module running on a dedicated server. Measurements are performed at a user-defined frequency on both the processor and the RAM. We monitor also flop rate using the `likwid` [80] software tool. All of the software probes run as separate threads which regularly access the corresponding counters and sleep inbetween measures. Data acquisition was performed at the frequency of 1 Hz by all of the software probes and for all the benchmarks presented in sections 6.2.1, 6.2.2 and 6.2.3. Data acquisition frequency then varies for selected test cases in Section 6.2.4 discussing the overhead of the monitoring software.

6.2.1 Study of multi-solve in shared-memory

At first, we study the evolution of power consumption (in Watts) with respect to execution time of the multi-solve algorithm (see Section 3.1) on a single computational node. We consider a coupled FEM/BEM linear system with 1,000,000 unknowns in total.

Regarding the *baseline multi-solve* algorithm relying on the MUMPS/SPIDO coupling, we set the size n_c of the $A_{sv_j}^T$ and S_i blocks to 256 columns (see Figure 3.1 in Section 3.1.1). For the *compressed Schur multi-solve* relying on the MUMPS/HMAT coupling, the size of the S_i and the size of the $A_{sv_j}^T$ blocks are handled respectively by the n_S and n_c parameters (see Figure 3.2 in Section 3.1.2). In this case, we set n_c to 256 and n_S to 1,024 columns so that compression is delayed until four blocks involving $A_{sv_j}^T$ are completed. The choice of these values is motivated by the previous study of the multi-solve algorithm in Section 3.3. In Figure 6.1, the top plot shows the evolution of the CPU and RAM power consumption as well as the flop rate for the multi-solve algorithm with the MUMPS/SPIDO coupling. Then, the bottom plot of the figure, shows the corresponding RAM usage evolution. The labels mark the alpha and the omega, i.e. the beginning and the end, of the most important computation phases. In the same way, Figure 6.2 then shows the results for the multi-solve algorithm with the MUMPS/HMAT coupling.

In the case of multi-solve, the Schur complement computation dominates the execution time as well as the RAM usage. From the point of view of the computational intensity, illustrated by the flop rate, it is the opposite. The factorization phase of the dense Schur complement matrix is very short for the MUMPS/HMAT coupling thanks to the usage of low-rank compression. Nevertheless, as of the MUMPS/SPIDO coupling, this phase consists of a *dense factorization* which is a computationally intensive operation. Indeed, in this case, we reach the flop rate as well as the power consumption peaks.

Figures 6.3 and 6.4 represent a zoom on figures 6.1 and 6.2 between the execution times 498 and 525 s, i.e. within the Schur complement computation phase. In these figures, we can observe cycles of high and low power consumption and flop rate. As of the RAM usage, the cycles are also present but less noticeable. Based on the labels, we can see that these cycles are almost entirely due to the *sparse solve* operation involved in the computation of each of the Schur complement blocks S_i (see Chapter 3). In case of fork-join algorithms such as multi-solve and multi-factorization, i.e. where the computations within one iteration are performed in parallel but the iterations themselves are not, it is not surprising to see this kind of cycles. From performance analysis point of view, they represent a waste of computation time. However, when we consider also the energetic point of view, the cycles represent a waste of energy for powering an idle CPU or RAM module.

Figure 6.5 finally compares the total energy consumption (in Joules), the total execution time and the peak RAM usage, once again of both variants (*baseline* MUMPS/SPIDO and further *compressed* MUMPS/HMAT) of the multi-solve algorithm. We consider three coupled FEM/-BEM systems of a total of 1,000,000, 3,000,000 and 5,000,000 unknowns, respectively. The results confirm, as one may expect, that the energy consumption, the execution time as well as the peak RAM usage rise with increasing size of the linear system. Moreover, the results show that the compressed MUMPS/HMAT variant (the one that also performs compression within the Schur) consumes less energy, in addition to being faster. This is an interesting result from an industrial point of view, further motivating the usage of low-rank compression techniques.

6.2.2 Study of multi-factorization in shared-memory

We now consider the multi-factorization algorithm. For the 1,000,000 unknown test case (see figures 6.6 and 6.7), we set the number of Schur complement block rows and columns n_b to 3

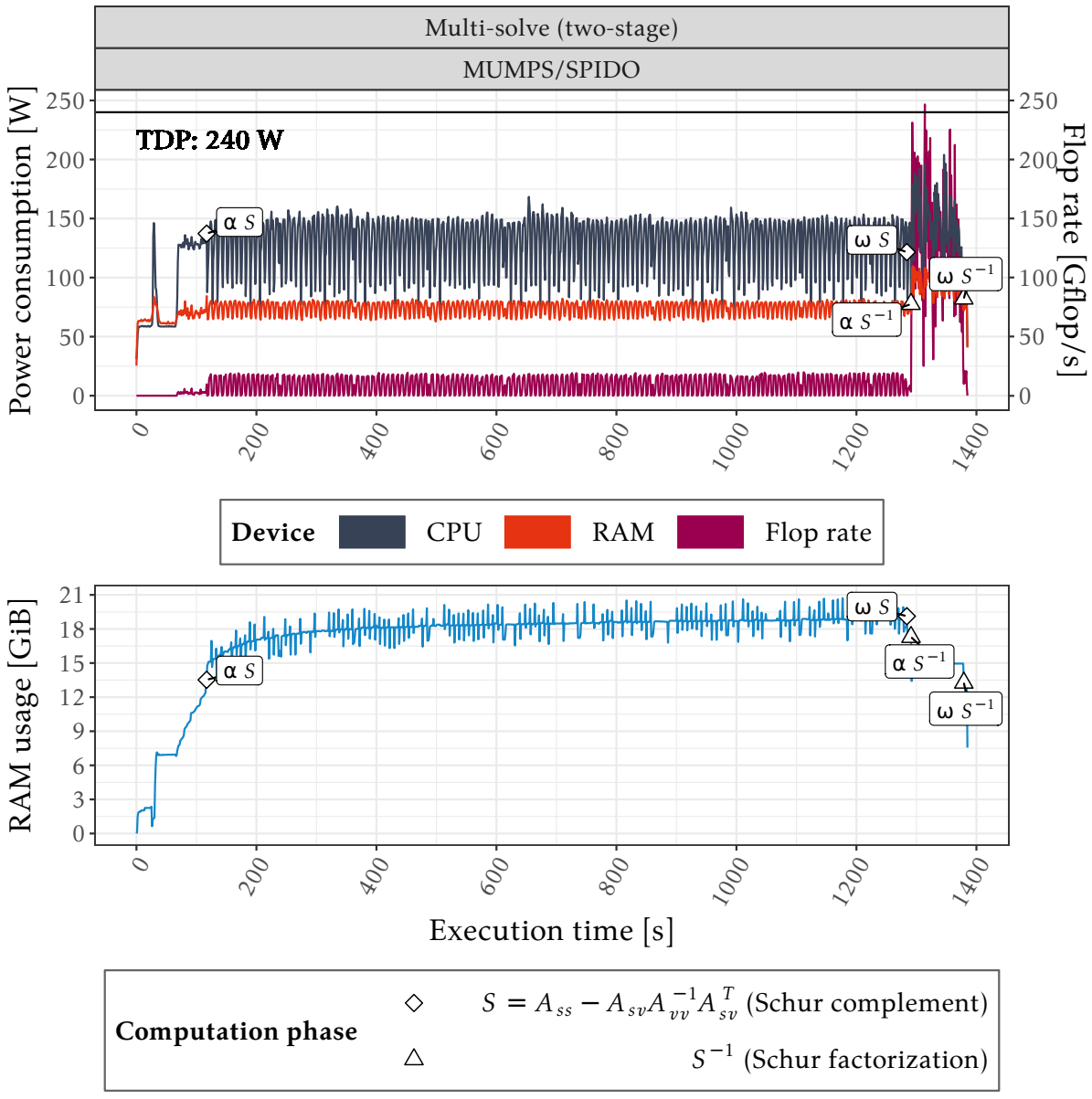


Figure 6.1: Power consumption, flop rate and RAM usage evolution of **multi-solve** with MUMPS/SPIDO on a FEM/BEM system with 1,000,000 unknowns. Parallel runs using 24 threads on a single `miriel` node. α and ω mark the beginning and the end of a computation phase.

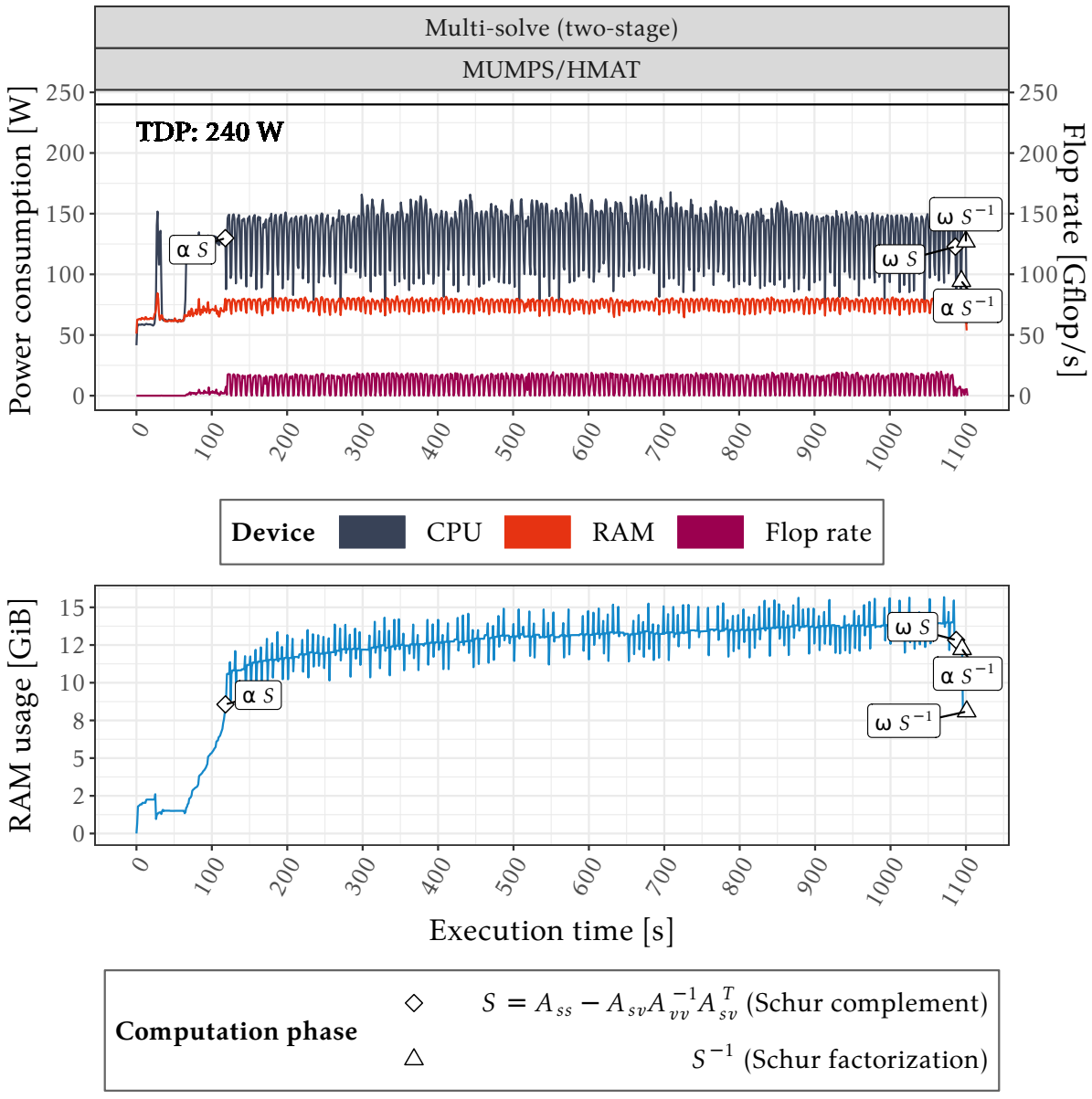


Figure 6.2: Power consumption, flop rate and RAM usage evolution of **multi-solve** with MUMPS/HMAT on a FEM/BEM system with 1,000,000 unknowns. Parallel runs using 24 threads on a single `miriel` node. α and ω mark the beginning and the end of a computation phase.

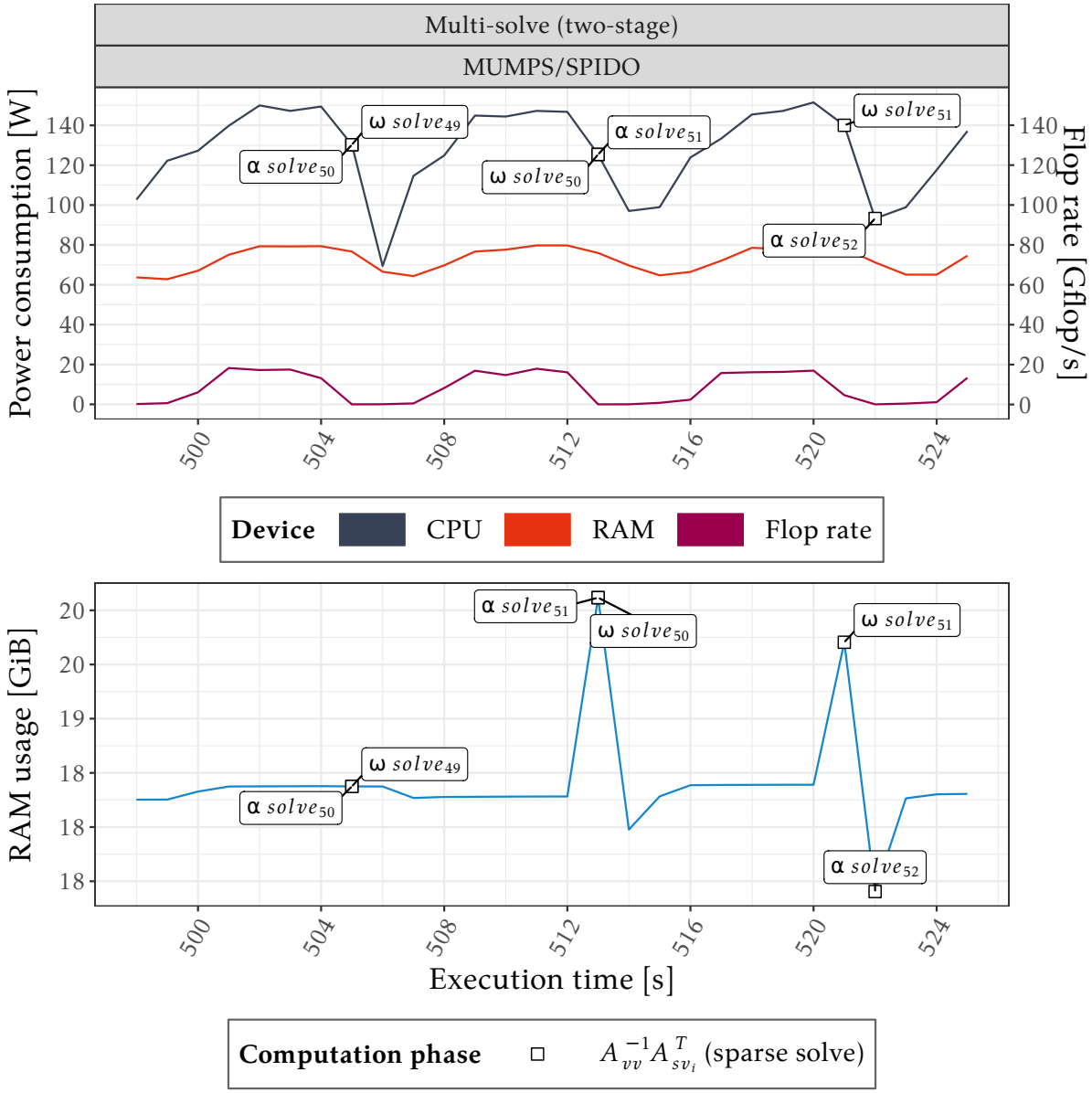


Figure 6.3: Zoom on Figure 6.1 between the execution times 498 and 525 s. The y-axis has been adapted to the extremum values in the selected time span.

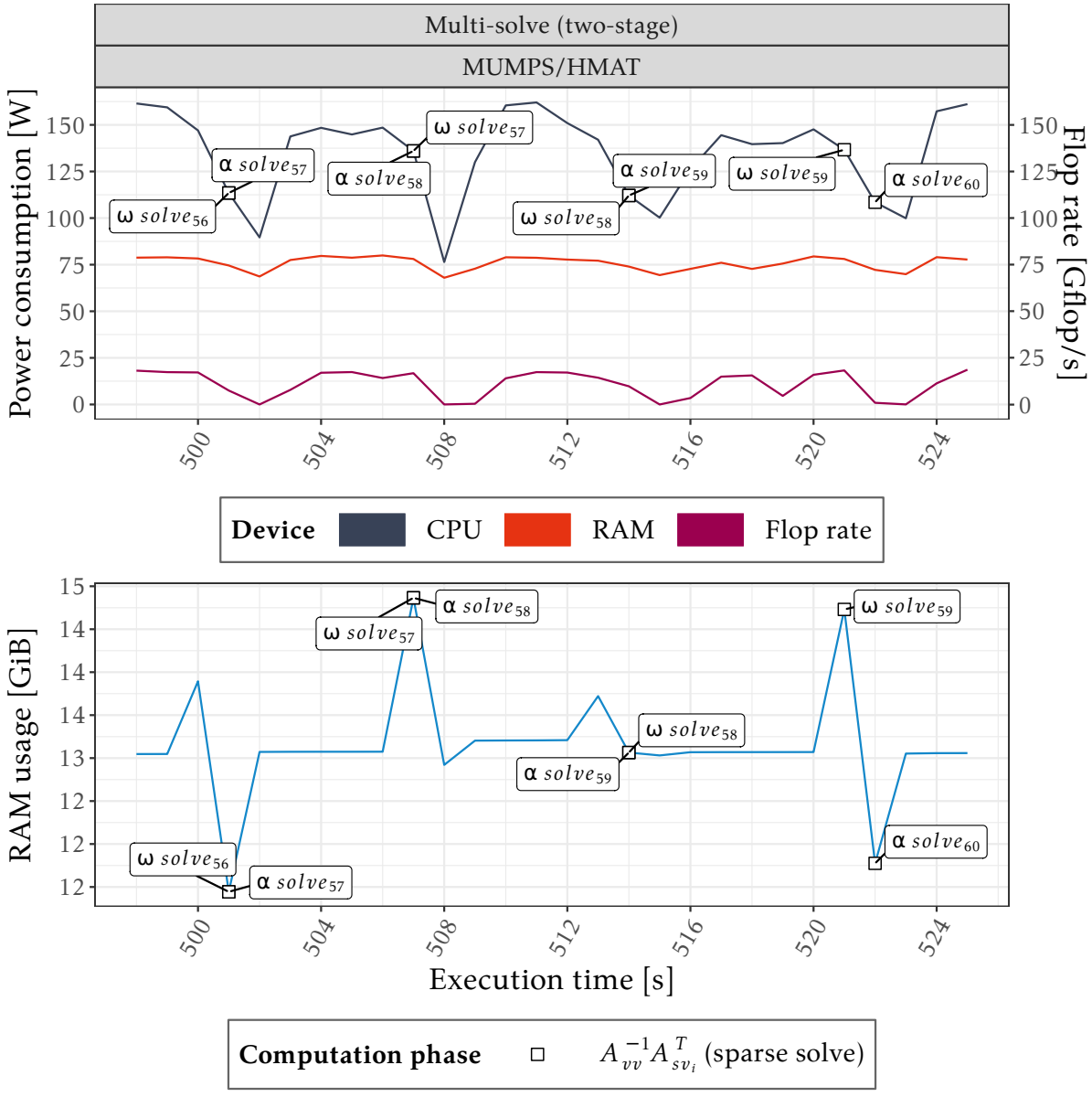


Figure 6.4: Zoom on Figure 6.2 between the execution times 498 and 525 s. The y-axis has been adapted to the extremum values in the selected time span.

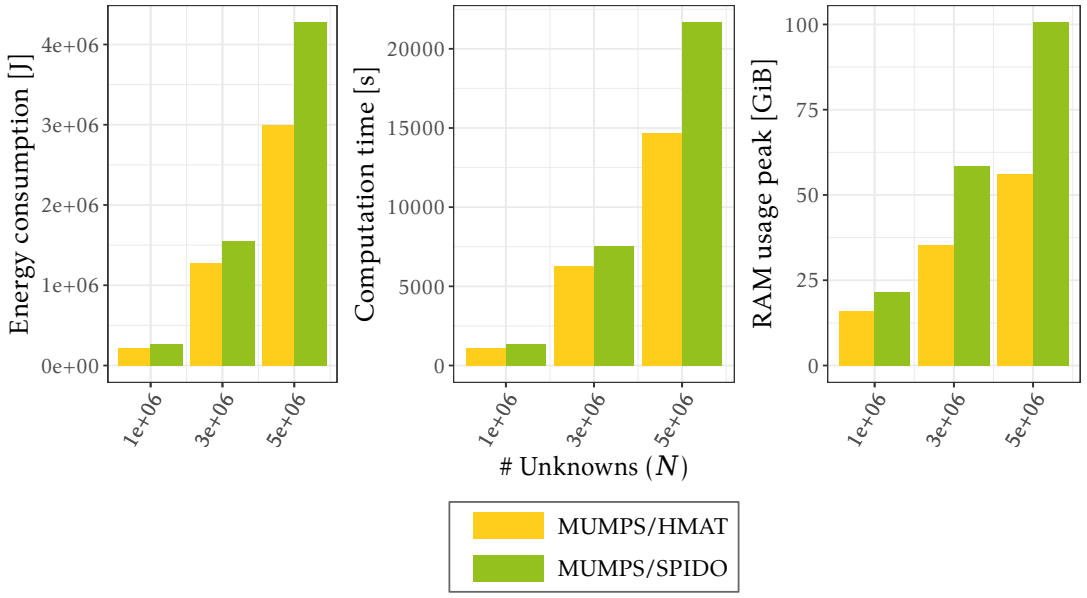


Figure 6.5: Total energy consumption, computation time and peak RAM usage of **multi-solve** for MUMPS/SPIDO and MUMPS/HMAT on FEM/BEM systems with 1,000,000, 3,000,000 and 5,000,000 unknowns. Parallel runs using 24 threads on a single `miriel` node.

for both MUMPS/SPIDO and MUMPS/HMAT solver couplings. As for multi-solve, in the case of multi-factorization, the execution time is dominated by the Schur complement computation. With $n_b = 3$, we have a total of 6 Schur complement blocks S_{ij} to compute. We can identify the moment of computation of each S_{ij} thanks to the apparent cycles of high and low power consumption and flop rate and especially of RAM usage. Here, the Schur complement computation phase again consumes most of the RAM but is not the most computationally intensive part of the algorithm. The peak power consumption and flop rates are met within the *dense factorization* of S in case of the MUMPS/SPIDO coupling.

Figure 6.8 compares the total energy consumption (in Joules), the total execution time and the peak RAM usage of the multi-factorization algorithm with problems of 1,000,000, 1,500,000 and 2,000,000 total unknowns. We can draw the same conclusion as in the case of multi-solve, i.e. the energy consumption, the execution time as well as the peak RAM usage rise with increasing linear system size. Nevertheless, the advantage of compressing the Schur complement matrix S is less important in the case of multi-factorization.

6.2.3 Multi-node study

We now consider a platform composed of four computational nodes and assess a coupled system of 2,000,000 total unknowns. Larger systems resulting in models with better resolution are analyzed in chapters 3, 4 and 5 dedicated to the design of the multi-solve and the multi-factorization algorithms. Here, we choose the system size large enough for leading to representative results when processed in parallel on multiple nodes.

6.2.3.1 Multi-solve algorithm

Figures 6.9 and 6.10 show the processor and RAM power consumption over all four monitored nodes of the multi-solve algorithm for the MUMPS/SPIDO and the MUMPS/HMAT couplings, respectively. The values of the block size parameters n_c and n_S are the same as defined in

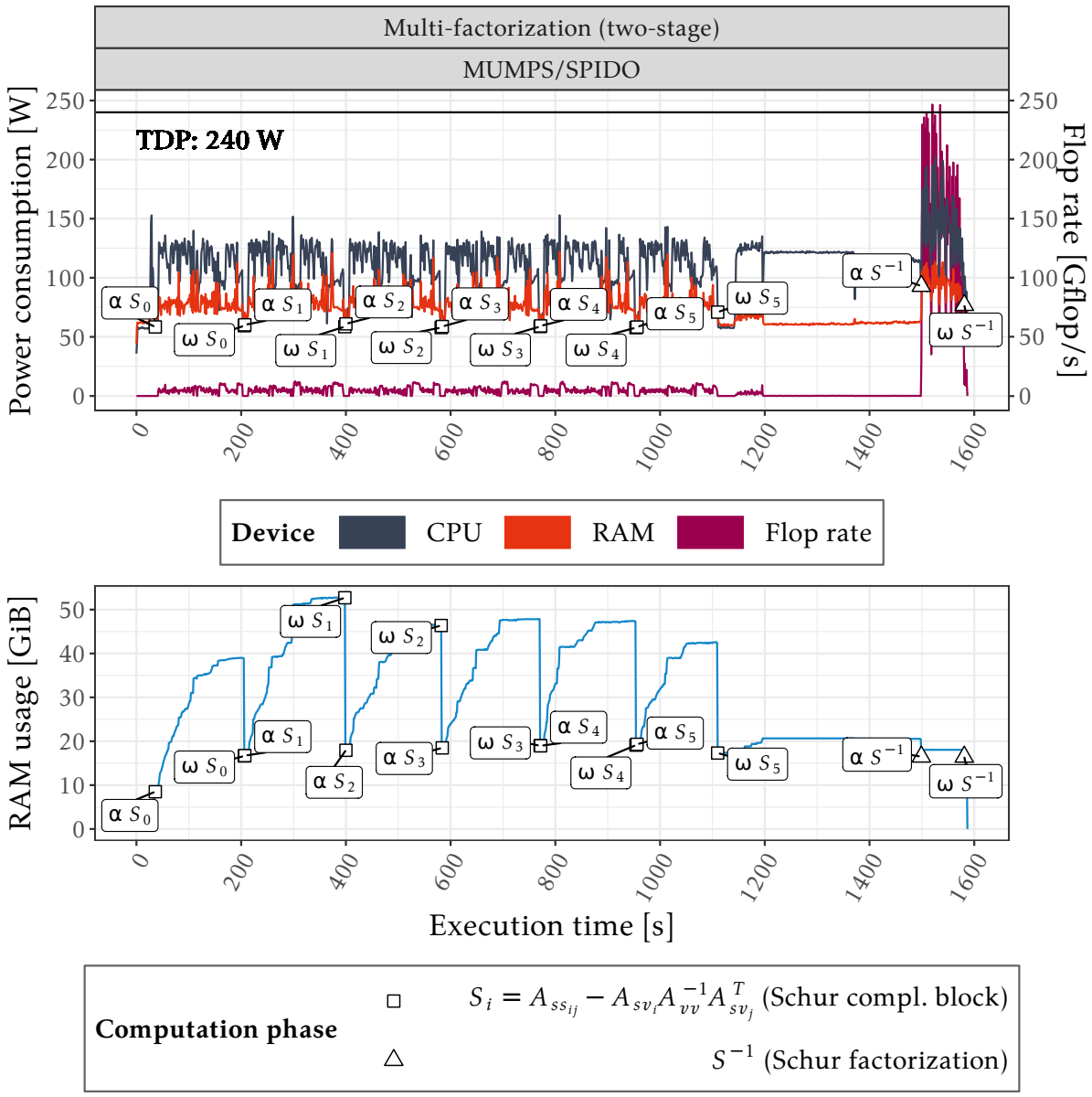


Figure 6.6: Power consumption, flop rate and RAM usage evolution of **multi-factorization** for MUMPS/SPIDO on a FEM/BEM system with 1,000,000 unknowns. Parallel runs using 24 threads on single `miriel` node. α and ω mark the beginning and the end of a computation phase.

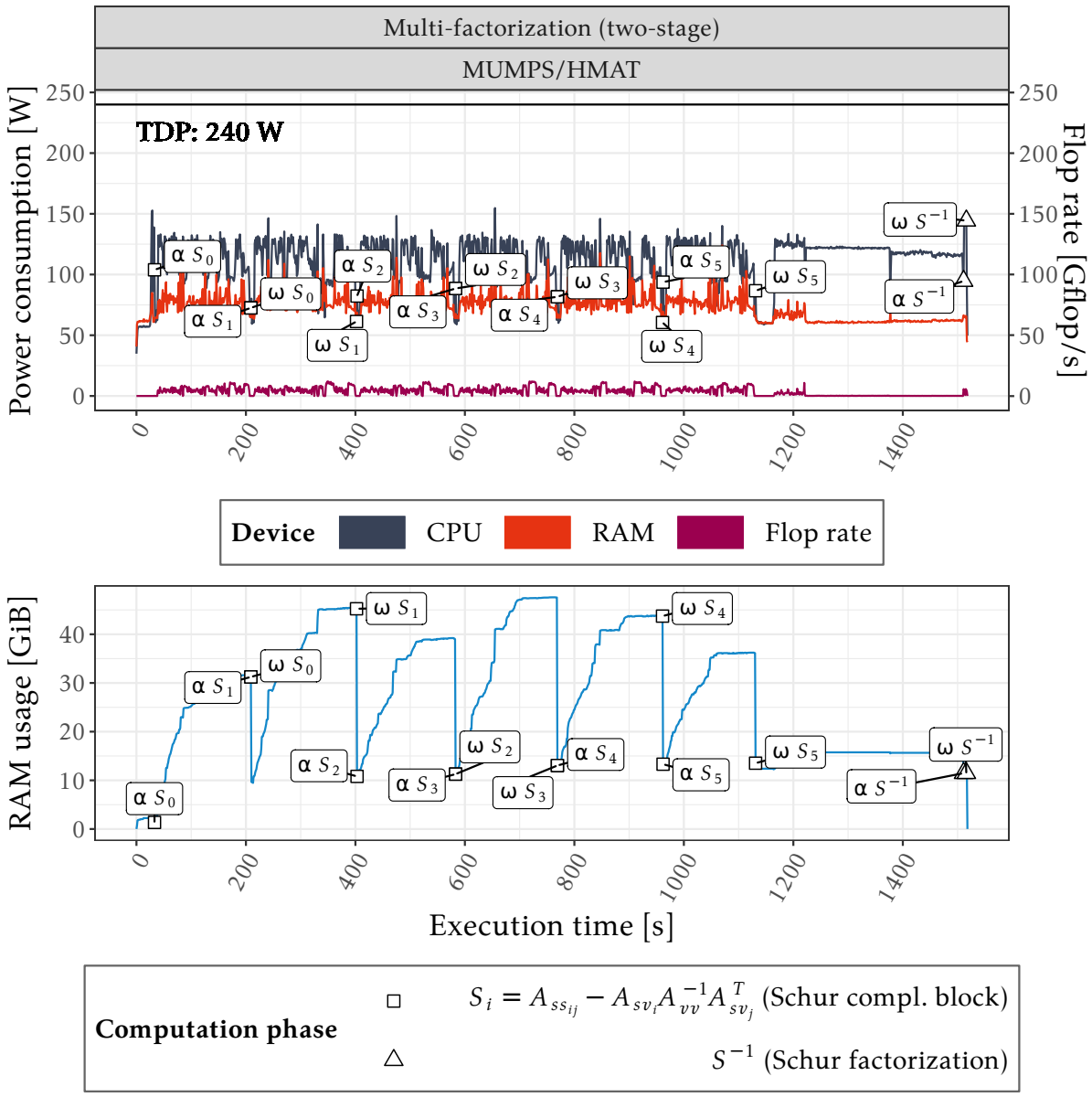


Figure 6.7: Power consumption, flop rate and RAM usage evolution of **multi-factorization** for MUMPS/HMAT on a FEM/BEM system with 1,000,000 unknowns. Parallel runs using 24 threads on single `miriel` node. α and ω mark the beginning and the end of a computation phase.

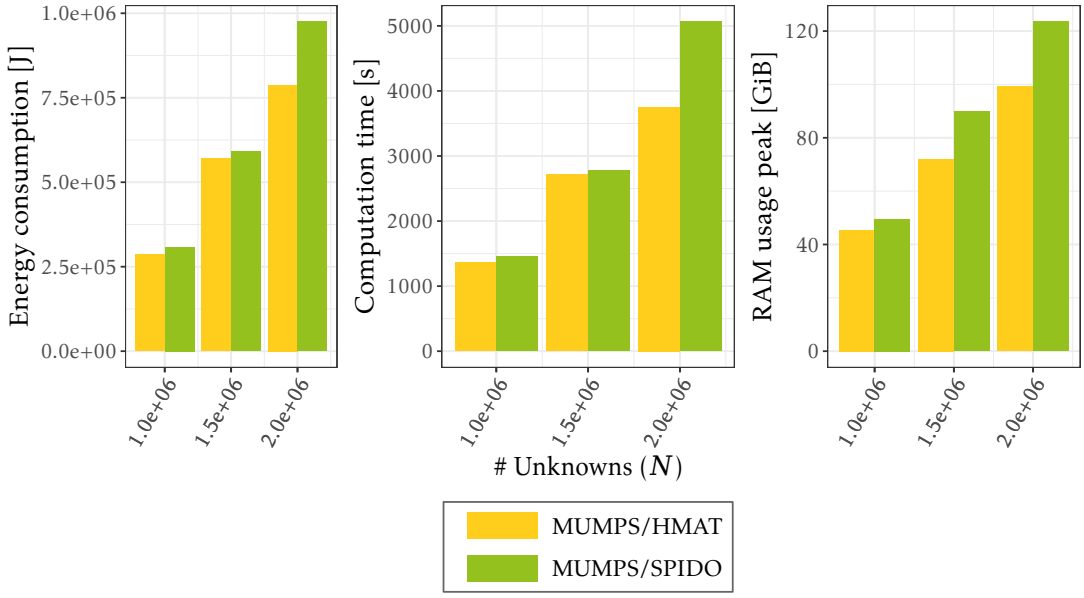


Figure 6.8: Total energy consumption, computation time and peak RAM usage of **multi-factorization** for MUMPS/SPIDO and MUMPS/HMAT on FEM/BEM systems with 1,000,000, 1,500,000 and 2,000,000unknowns. Parallel runs using 24 threads on a single `miriel` node.

Section 6.2.1. The results confirm that the power consumption in a distributed parallel test case evolves similarly to the single node test case (see Section 6.2.1). The RAM usage also follows the pattern of the single node test case.

6.2.3.2 Multi-factorization algorithm

Figures 6.11 and 6.12 show the behavior of the multi-factorization algorithm for the MUMPS/-SPIDO and the MUMPS/HMAT couplings, respectively. The number of Schur complement block rows and columns n_b is again set to 3 for both couplings as in Section 6.2.2. The evolution of RAM usage of multi-factorization follows the pattern of the single node test case (see Section 6.2.2). Regarding the power consumption, the high and low cycles corresponding to the computation of different Schur complement blocks differ considerably compared to the single-node test case. At the beginning of each cycle, there is a peak but the consumption falls down long before the end of the *sparse factorization+Schur* step which may indicate an under-optimized usage of this routine in a parallel distributed environment, e.g. a load imbalance. An analysis of execution traces might give us a better insight on the exact cause of this behavior. However, the present study accentuates our previous observation from Chapter 5 reporting a potential bottleneck in the usage of the distributed-memory parallelization of a key component in the solver stack. This increases the interest of such multi-metric profiles beyond the assessment of the overall energy consumption.

6.2.4 Overhead

In this section, we study the overhead of the three software probes `rss.py`, `energy_scope` and `likwid` used to take memory usage, power consumption and flop rate measures. For this we run the application with and without the monitoring software on a coupled FEM/BEM linear system with 1,000,000 unknowns for both multi-solve and multi-factorization algorithms and for both solver couplings, i.e. MUMPS/SPIDO and MUMPS/HMAT. We consider the same block sizes as defined in sections 6.2.1 and 6.2.2.

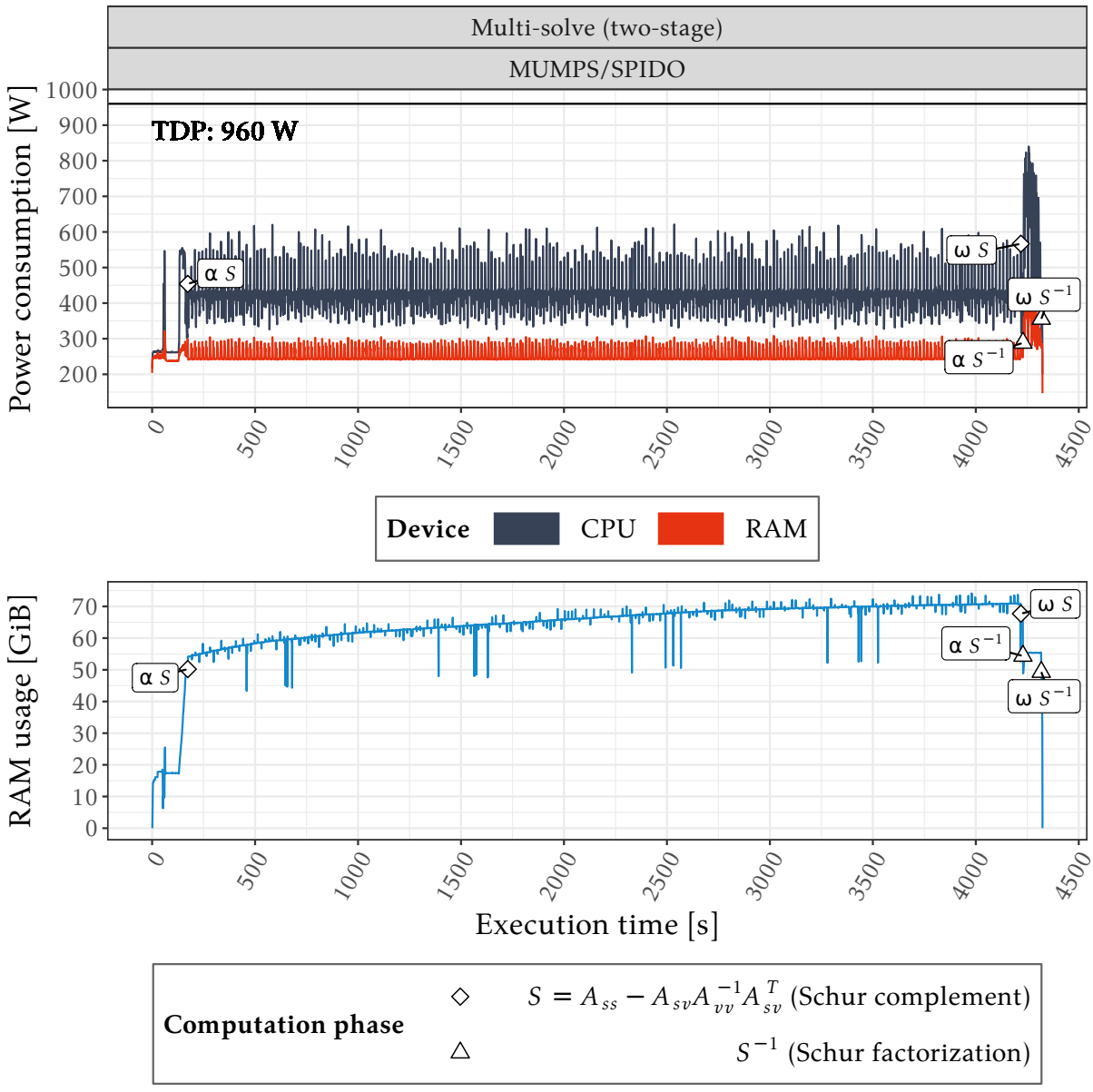


Figure 6.9: Power consumption, flop rate and RAM usage evolution of **multi-solve** for MUMPS/SPIDO on a FEM/BEM system with 2,000,000 unknowns. Distributed parallel runs using a total of 96 threads on 4 `miriel` nodes. α and ω mark the beginning and the end of a computation phase.

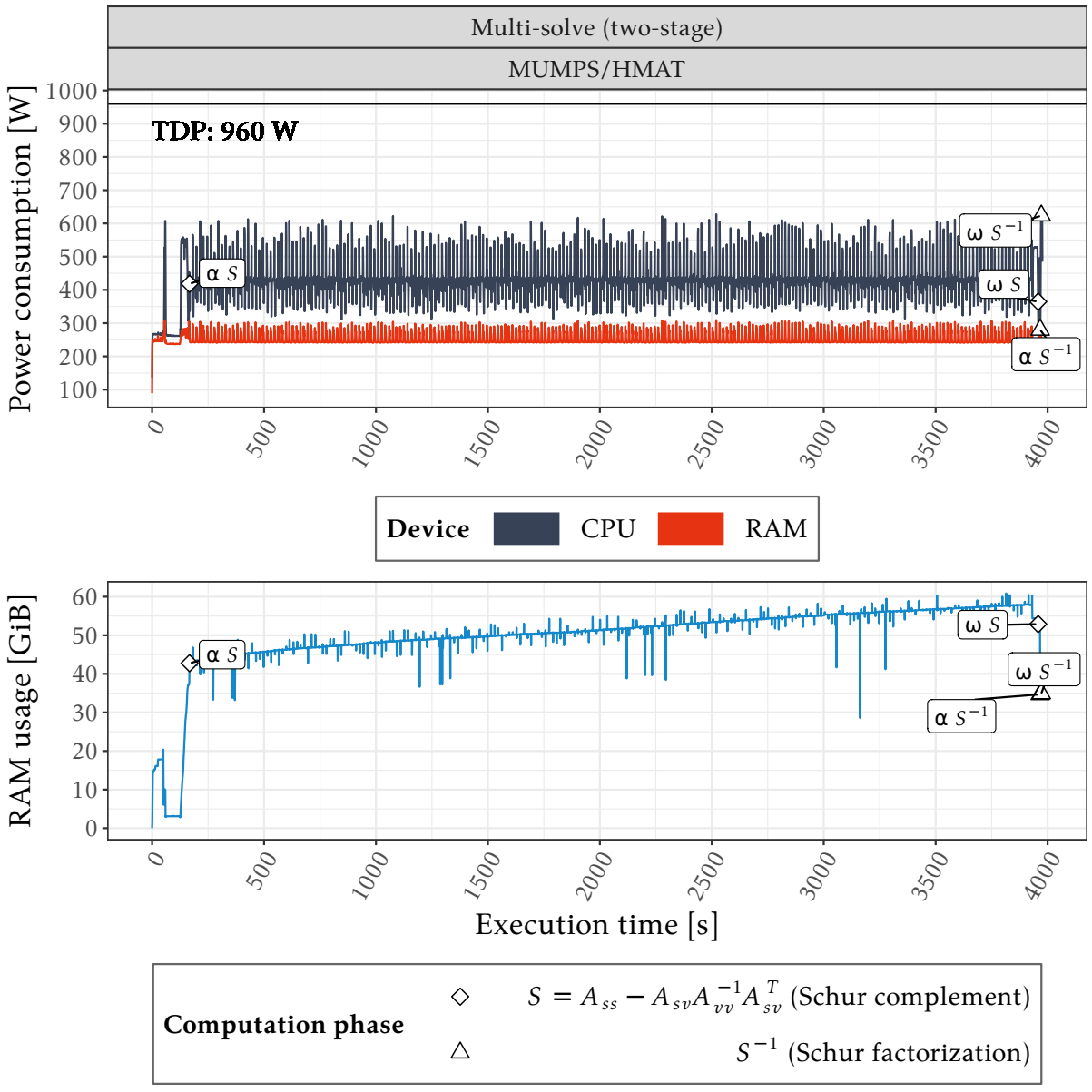


Figure 6.10: Power consumption, flop rate and RAM usage evolution of **multi-solve** for MUMPS/HMAT on a FEM/BEM system with 2,000,000 unknowns. Distributed parallel runs using a total of 96 threads on 4 `miriel` nodes. α and ω mark the beginning and the end of a computation phase.

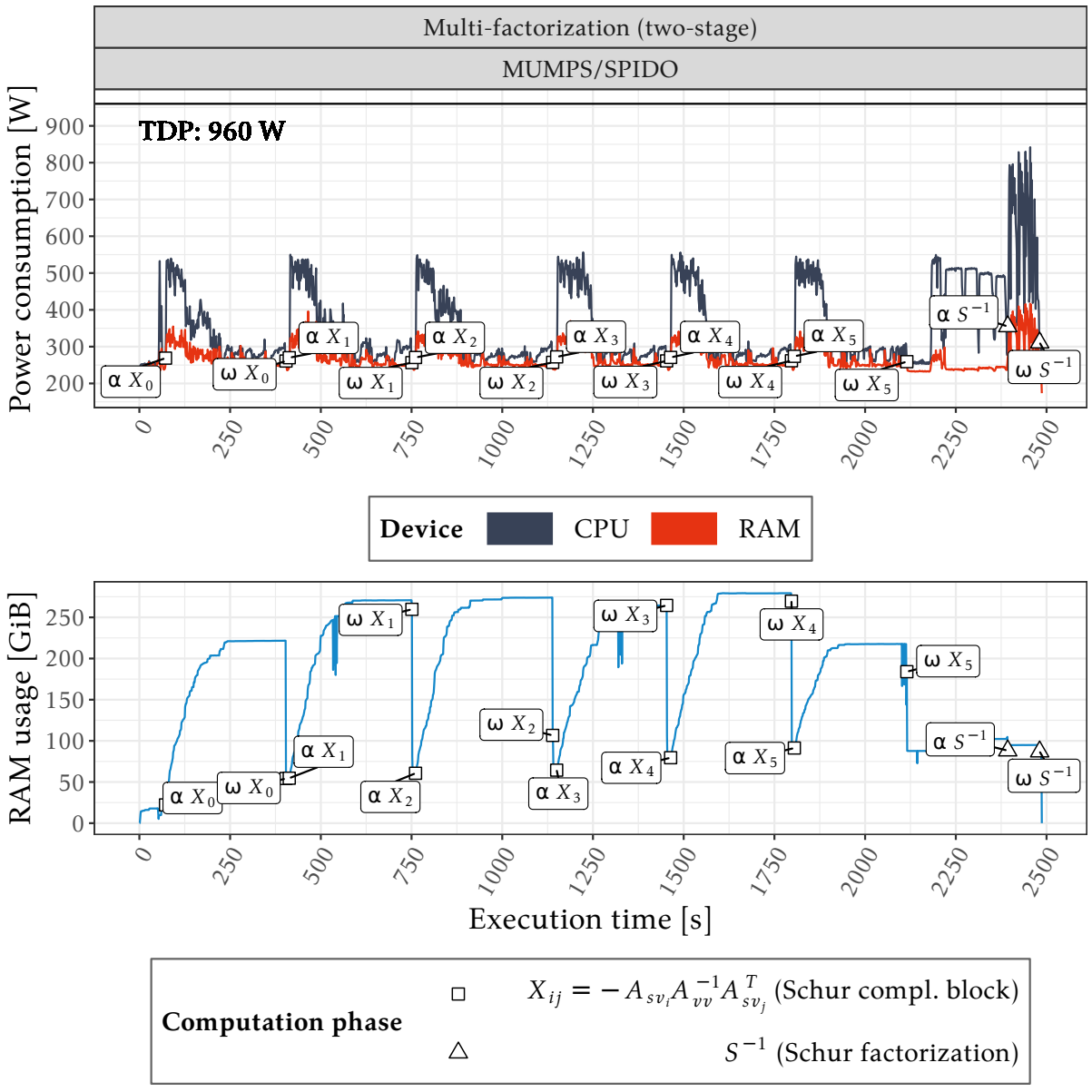


Figure 6.11: Power consumption, flop rate and RAM usage evolution of **multi-factorization** for MUMPS/SPIDO on a FEM/BEM system with 2,000,000 unknowns. Distributed parallel runs using a total of 96 threads on 4 `miriel` nodes. α and ω mark the beginning and the end of a computation phase.

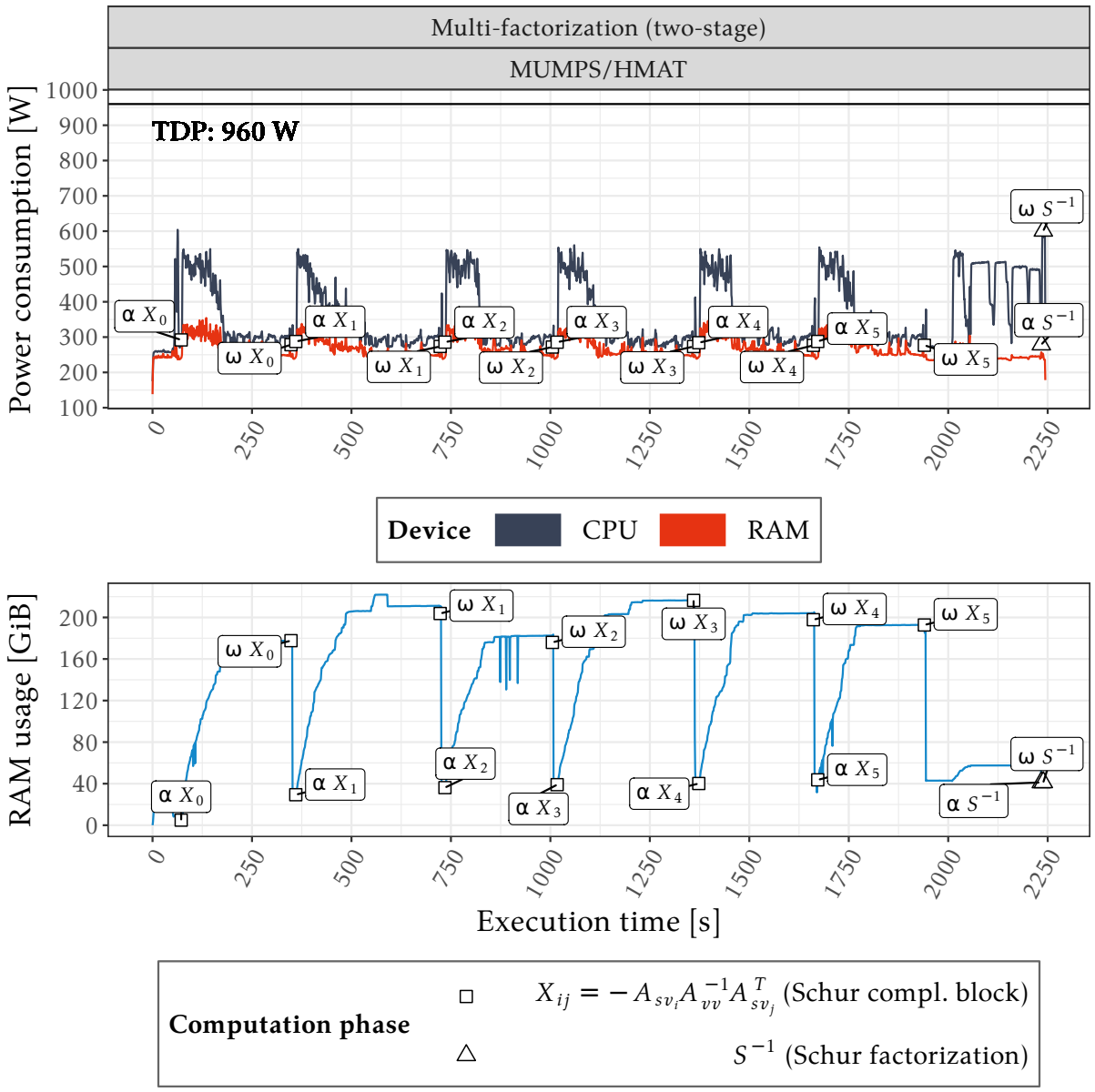


Figure 6.12: Power consumption, flop rate and RAM usage evolution of **multi-factorization** for MUMPS/HMAT on a FEM/BEM system with 2,000,000 unknowns. Distributed parallel runs using a total of 96 threads on 4 `miriel` nodes. α and ω mark the beginning and the end of a computation phase.

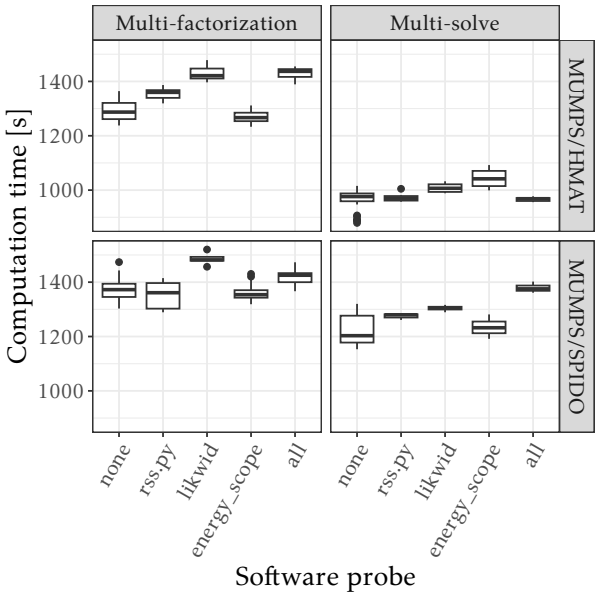


Figure 6.13: Computation times of different levels of instrumentation for **multi-solve** and **multi-factorization** and for both the MUMPS/SPIDO and the MUMPS/HMAT solver couplings on a FEM/BEM linear system with 1,000,000 unknowns. Parallel runs using 24 threads on single `miriel` node.

In Figure 6.13, we rely on a box plot representation to compare the computation times of the application running with different levels of instrumentation; that is instrumented with each software probe separately, with all the software probes at once and with no instrumentation. The acquisition frequency is always 1 Hz. All the runs related to a given algorithm, solver coupling and instrumentation level, e.g. multi-solve with MUMPS/SPIDO and `rss.py`, are run on the same computational node. Each test case is repeated 6 times with and without instrumentation. In other words, we have 4 possible algorithm and solver coupling combinations. Each combination is run 6 times with and without instrumentation. Finally, there are 4 different levels of instrumentation, i.e. running separately with `rss.py`, `likwid` and `energy_scope` and with all probes together. In total, we performed 192 ($4 \times 2 \times 6 \times 4$) runs on at most 16 ($4 \times 4$) different `miriel` nodes. The results lead to three main observations. One, there is a non-negligible computation time variability at all levels, even when no instrumentation is used, which makes the interpretation of the overhead a complex matter. The variability could be reduced by making all the 192 tests on the same node. Two, the fact that instrumentation with `likwid` all alone can cause a more important overhead than instrumentation with all the software probes at once, e.g. in case of multi-factorization with MUMPS/SPIDO, will require a deeper investigation. Three, the overhead of all the probes used at once is 6% to 8%.

`likwid` and `energy_scope` support custom acquisition profiles. Throughout the study, we were monitoring only one metric at a time and at the frequency of 1 Hz with each of the software probes. To evaluate the impact on the application overhead, we also run `likwid` and `energy_scope` with alternative acquisition profiles, i.e. with up to 2 Hz frequency and monitoring up to 2 metrics at once. For `likwid`, we considered RAM bandwidth in addition to flop rate and for `energy_scope`, we considered CPU temperature in addition to CPU and RAM power consumption. According to results in Table 6.1, considering higher acquisition frequency or multiple metrics at once most of the time leads to an individual overhead of each software probe which is higher than the combined overhead of all the probes in the case of our initial acquisition profile, i.e. 6% to 8% with 1 Hz frequency and 1 metric per probe.

Note that, within this study, we wanted to perform a high-level profiling with non-intrusive

Software probe	Frequency	Metrics	Max. overhead
`likwid`	2 Hz	1	10%
`likwid`	1 Hz	2	11%
`likwid`	2 Hz	2	17%
`energy_scope`	2 Hz	1	9%
`energy_scope`	1 Hz	2	4%
`energy_scope`	2 Hz	2	9%

Table 6.1: Maximal overhead of `likwid` and `energy_scope` on the application for varying acquisition frequencies and metrics.

tools. Here, the term 'non-intrusive' does not refer to the performance but to the fact that there is no need for modifying the source code of our application to perform the measures. In the case we want a finer analysis with a lower overhead, we might have to resort to lower-level libraries such as PAPI whose overhead is known to be very low [90].

6.3 Discussion

The experimental results confirmed that the performance advantages of the *compressed Schur multi-solve* and *compressed Schur multi-factorization* algorithms also translate into energy consumption. In addition to that, the study allowed us to pinpoint a major performance bottleneck represented by the unwanted cycles of high and low power consumption in case of both multi-solve and multi-factorization schemes. It is not exactly within the sparse solver itself, but more on the fact that the schemes make many calls to the solvers and that their API (the one of MUMPS but also of other fully-featured solvers) is synchronous. We anticipated that it might lead to a performance penalty but we did not expect the impact to be that significant. After presenting this study to developers of sparse direct solvers, they agreed that alleviating the synchronizations through asynchronous calls would certainly be a nice feature. Because fully-featured sparse direct solvers have hundreds of thousands lines of code and changing their algorithms and API is a complex task, this can only be a long-term work. However, this could be achieved within a non fully-featured prototype. Chapter 7 is therefore dedicated to the exploration of an alternative solver coupling allowing for an asynchronous management so that we can change the flow with which we call the solvers and avoid synchronizations. The idea is to rely on a novel task-based design proposing asynchronous calls in the API of the solvers so that the calls to the sparse and the dense solver can be pipelined. We expect such an asynchronous scheme to alleviate the potential load balancing issues in the advanced *sparse factorization+Schur* functionality of the sparse direct solver highlighted within the multi-node study.

6.4 Conclusion

We have studied the energetic profile of a complex HPC application, involving dense, sparse, and compressed operations. The study established that the further compressed algorithms (MUMPS/HMAT) are also worth from an energetic perspective. The energy profiles together with the memory usage and flop rate allowed us to better understand the behavior of the application, up to the point that we identified a major performance bottleneck in our implementation as well as a potential load balancing issue in the sparse direct solver.

These results offer new directions to explore and may as well serve as feedback for the developers of sparse direct solvers. While the main goal of Chapter 7 is to implement an alternative

single-stage algorithm class, it could also allow us to alleviate the aforementioned drawbacks.

Chapter 7

Towards a single-stage algorithm

We dedicated the first part of this thesis to the design and extension of the two-stage algorithms, namely multi-solve and multi-factorization, for the solution of large coupled sparse/-dense FEM/BEM systems presented in Section 1.3. The core idea of these algorithms is to rely on the existing API of fully-featured and well-optimized sparse and dense direct solvers implementing in their building blocks advanced features such as numerical compression, out-of-core and parallel distributed computation. The experimental studies in chapters 3, 4 and 5 showed that the two-stage algorithms allow for more efficient couplings of fully-featured direct solvers compared to a straightforward usage of the latter within the vanilla baseline and advanced approaches (see Section 2.2). However, in Section 7.1, we explain why they remain suboptimal with respect to the ideal implementation which would lead to optimal performance and memory consumption. Therefore, in Section 7.2, we discuss the possible design of a single-stage algorithm leveraging the limitations of the vanilla couplings which the two-stage algorithms allowed us to cope with, nevertheless, without fully avoiding them.

The rest of this chapter is organized as follows. In Section 7.3, we propose a task-based single-stage algorithm meant for fully exploiting the symmetry and the sparsity of the target linear system (1.2) within the solution process. A non fully-featured prototype implementing the proposed algorithm is introduced in Section 7.4. We conduct a preliminary experimental evaluation of our implementation in Section 7.5. We conclude in Section 7.6.

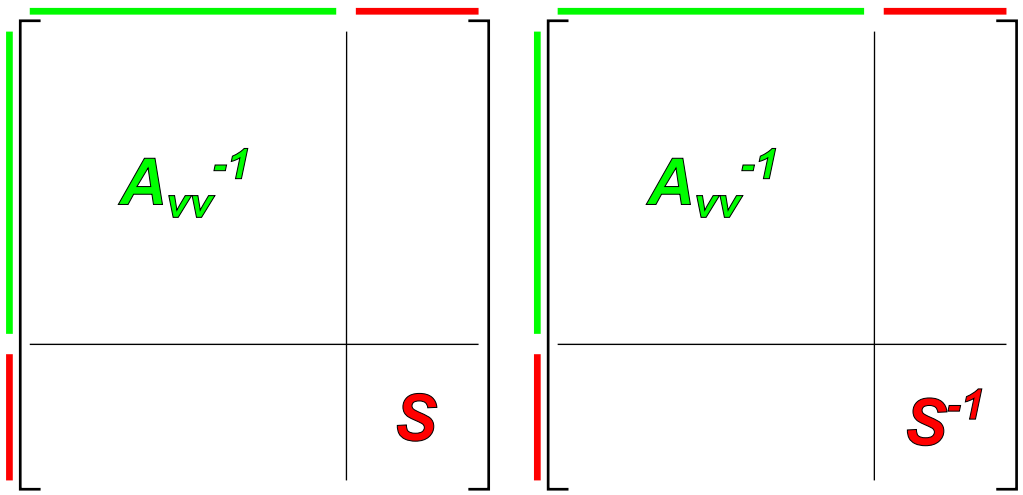
7.1 Design limitations of two-stage algorithms

The two-stage algorithms are designed to cope with the limitations in the state-of-the-art vanilla solver couplings related to a straightforward usage of the fully-featured direct solvers. Derived from the baseline coupling (see Section 2.2.1), multi-solve avoids the explicit storage of the entire A_{sv}^T in a non-compressed dense matrix. Together with multi-factorization, derived from the advanced coupling (see Section 2.2.2), the algorithms aim at assembling the Schur complement matrix S in multiple smaller blocks. The common motivation is to enable the application of numerical compression and out-of-core computation techniques also on the dense Schur complement part of the system. Although the two-stage algorithms allow for bypassing the limitations of their vanilla counterparts, they cannot eliminate them completely and reach the ideal approach (see Section 1.6.3) while relying on the existing API of direct solvers. As we discussed in Section 3.1, multi-solve cannot fully preserve the symmetry of the system as it needs to store a block Y_i of columns of A_{sv}^T in a dense matrix at each iteration (see Figure 3.1 on p. 41

and Figure 3.2 on p. 42). It does not benefit from the optimal performance of the *sparse factorization+Schur* building block either. Multi-factorization (see Section 3.2) implies an important data duplication in A_{vv} for the computation of extra-diagonal blocks of S and suffers from superfluous re-factorizations of A_{vv} in W (see Figure 3.3 on p. 44 and Figure 3.4 on p. 46) at each iteration. Moreover, the fork-join character of the algorithms induced by synchronous API calls leads to an important performance degradation as was pointed out in the multi-metric study in Chapter 6. In the aim to reach the ideal implementation of a sparse/dense coupled solver described in Section 1.6.3 and avoid the aforementioned drawbacks related to the usage of the existing solver API, we propose the single-stage algorithm scheme relying on an alternative API.

7.2 Single-stage approach

The advanced vanilla solver coupling (see Section 2.2.2) preserves and exploits the symmetry and the sparsity of the linear system to solve (1.2). When it fits in memory, it represents an optimal implementation from the performance point of view too. However, as discussed in Section 2.3.2, there is a major drawback in terms of memory footprint of the algorithm on the articulation between the sparse and the dense direct solver, i.e. on the Schur complement part S. Before the dense solver takes over with the *dense factorization* of S in the second stage, the latter must be fully assembled in RAM in a non-compressed dense matrix by the sparse solver in the first stage of the computation (see Figure 7.1).



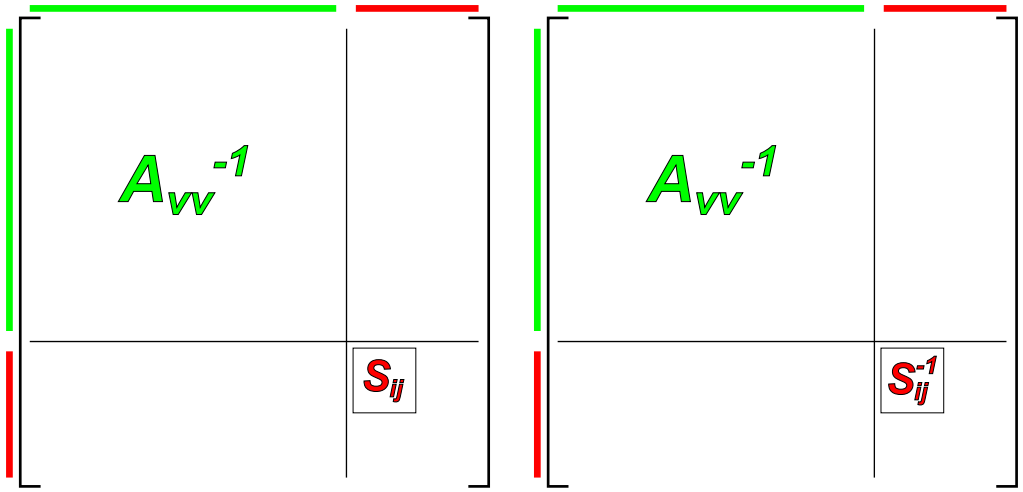
(a) *first stage:* assembly of the entire S by the sparse solver

(b) *second stage:* factorization of S by the dense solver

Figure 7.1: Core phases of the advanced vanilla coupling (see Section 2.2.2) in two stages.

The goal of the single-stage approach is multi-faceted. One, we want to preserve and take advantage of the ideal symmetry and sparsity condition of the coupled system. Two, similarly to the multi-solve and multi-factorization algorithms but without their remaining drawbacks (see Section 7.1), we want to avoid the assembly of the entire Schur complement matrix S in RAM by resorting to out-of-core techniques. Three, we want to be able to begin the *dense factorization* of S before it is fully assembled and this way perform the entire computation asynchronously in a single stage instead of two synchronous stages (see Figure 7.2).

To achieve such an implementation, we can rely either on a unique solver capable of applying both sparse and dense operations according to the partitioning of the target linear system or on a coupling of a sparse and dense direct solvers having APIs compatible enough to ensure the



(a) Beginning of the *sparse factorization+Schur* and computation of the first tile of the Schur complement matrix S.

(b) Factorization of the first tile of S while the *sparse factorization+Schur* continues.

Figure 7.2: Principle of the single-stage approach.

composability between the sparse and dense operations as described above. The first option seems to be the one that favors such a composability the most. However, it practically implies the development of a new solver having all the capabilities and optimizations of sparse and dense direct solvers. It could be done only as a long-term work. Therefore, in this thesis, we choose to focus on the second option. This way, we can adapt existing solvers and take advantage of all their available features. While modifying the API of a fully-featured direct solver can also be only a long-term work due to complex code base, a proof-of-concept can be implemented within a non fully-featured prototype.

7.3 Task-based algorithm

We consider a coupling of task-based (see Section 2.6.3) sparse direct and dense direct solvers. To improve the composability of the solvers as well as to ease the design of compatible APIs allowing us to avoid the synchronization on the link between sparse and dense operations, we resort to solvers using the same runtime.

Like the advanced vanilla solver coupling (see Section 2.2.2), the task-based single-stage algorithm builds on the *sparse factorization+Schur* building block of the sparse direct solver to compute the Schur complement $X = -A_{sv}A_{vv}^{-1}A_{sv}^T$ associated with the $\begin{bmatrix} A_{vv} & A_{sv}^T \\ A_{sv} & 0 \end{bmatrix}$ matrix. However, instead of returning the entire Schur complement in a unique dense matrix, the building block has been adapted so as to split the latter into multiple blocks $X_{ij} = -A_{sv_i}A_{vv}^{-1}A_{sv_j}^T$ of equal size n_{BEM}/n_b where A_{sv_i} is a subset of n_{BEM}/n_b rows of A_{sv} and $A_{sv_j}^T$ a subset of n_{BEM}/n_b columns of A_{sv}^T. The blocks can then be assembled in an asynchronous manner and stored out-of-core before triggering the computation of the corresponding equally-sized blocks $S_{ij} = A_{ss_{ij}} + X_{ij}$ of the Schur complement S. The number of blocks per block row and block column of X and S is determined by the n_b parameter. Finally, based on the definition of S in (1.8), S_{ij} is a block of n_{BEM}/n_b rows and columns of S such as:

$$S_{ij} = A_{ss_{ij}} \overbrace{-A_{sv_i}A_{vv}^{-1}A_{sv_j}^T}^{X_{ij}} . \tag{7.1}$$

We therefore propose a task-based single-stage approach in Algorithm 7. It begins by the construction of a temporary submatrix W from A_{vv} and A_{sv} (line 2). Initial A_{vv} and A_{sv} are saved to disk in order to prevent data duplication. Then, we call the *sparse factorization+Schur* step on W (line 3) relying on the Schur complement feature provided by the sparse direct solver (see Section 2.1.1). This call returns only references to blocks of the Schur complement X associated with the submatrix W. Meanwhile, each block $X_{ij} = -A_{sv_i}(L_{vv}L_{vv}^T)^{-1}A_{sv_j}^T$ can be assembled asynchronously. Then, we trigger the $A_{ss_{ij}} + X_{ij}$ operations (line 6) to determine the blocks S_{ij} of the Schur complement S following (7.1). The instructions on lines 7 to 11 leading to the final solution in x_v and x_s can also be triggered immediately. The runtime the solvers are built on shall handle the execution of all the tasks and ensure the data dependencies between them.

Algorithm 7: Single-stage algorithm for the solution of (1.2) relying on the coupling of task-based sparse and dense direct solvers.

1 **Function** `SingleStage`(A, b)**:**
2 $W \leftarrow \begin{bmatrix} A_{vv} & 0 \\ A_{sv} & 0 \end{bmatrix}$
▷ Below operations can all be triggered immediately and performed asynchronously:
3 $X \leftarrow$ `SparseFactorization+Schur`(W) ▷ Retrieve block references.
4 **for** $i = 1$ **to** n_b **do**
5 **for** $j = 1$ **to** i **do**
6 $A_{ss_{ij}} \leftarrow A_{ss_{ij}} + X_{ij}$ ▷ AXPY
7 $A_{ss} \leftarrow$ `DenseFactorization`(A_{ss})
▷ Reusing the factorized A_{vv} within W:
8 $b_v \leftarrow$ `SparseSolve`(A_{vv}, b_v)
9 $b_s \leftarrow b_s - A_{sv}b_v$
10 $x_s \leftarrow$ `DenseSolve`(A_{ss}, b_s)
11 $x_v \leftarrow$ `SparseSolve`$(A_{vv}, b_v - A_{sv}^T x_s)$

In this case, avoiding a dramatic increase in RAM usage during the computation, especially due to the Schur complement assembly, can be achieved thanks to two mechanisms. On the one hand, we can rely on the out-of-core feature (see Section 2.6.1) of the underlying runtime (see Section 2.6.3) to respect a given RAM usage limit by evicting currently unused data on disk. On the other hand, we can limit the number of tasks submitted to the runtime to prevent the latter from further allocating memory if a given threshold is reached. In this case, one must ensure that delaying task submission will not lead to a deadlock due to an unsatisfied inter-task dependency. Also, compared to the first option, this strategy is likely to limit the parallelism whereas disk usage in the case of out-of-core is likely to degrade the performance of the application due to higher latency of input/output operations. [28, 26, 111, 102, 120] provide deeper insight on the problem.

7.4 Prototype implementation

As we explained in Section 7.2, implementation of the single-stage algorithm (see Algorithm 7) in fully-featured direct solvers is a complex task and can only be done as a long-term work. In this thesis, we thus choose to implement a proof-of-concept in a non fully-featured prototype. For this, we rely on the sparse direct solver `qr_mumps` (see Section 2.6.4.1) and the dense direct solver HMAT (see Section 2.6.4.2). Both solvers are task-based and use the StarPU runtime

[44]. Also, the solvers expose synchronous and asynchronous API. Following our initiative to implement the single-stage algorithm, `qr_mumps` was extended with an LL^T factorization routine for complex symmetric matrices as well as with the crucial *sparse factorization+Schur* building block.

There are some sacrifices to be made in a non fully-featured framework. The `qr_mumps` solver does not implement either distributed memory parallelism (although it is an ongoing work [27]) or numerical compression. Although it exposes a parameter to control its memory consumption [20, 28], it has no effect on the *sparse factorization+Schur* building block so far. This is due to the memory management and currently implemented memory allocation granularity. In the current implementation, the Schur complement must be fully allocated in RAM. Another limitation prevents us from reusing the factorized A_{vv} within W for the *sparse solve* operation on line 8. In the prototype implementation, we thus have to restore the original A_{vv} from disk, which also makes us put a synchronization barrier after the *dense factorization* on line 7, re-factorize A_{vv} and perform the remaining operations in a synchronous manner.

However, we seek to implement a proof-of-concept and this solver configuration allows us to implement a prototype of the single-stage algorithm leading to a better composability of sparse and dense operations thanks to the common usage of the StarPU runtime. This also favors asynchronous execution of sparse and dense computation tasks, at least up to the *dense factorization* of the Schur complement S. Note that, as a consequence of using HMAT as dense solver, the X_{ij} tiles, retrieved on line 3, need to be compressed before the AXPY operation in line 6 which thus becomes a compressed AXPY.

7.5 Preliminary experimental evaluation

We conducted a preliminary experimental study of the prototype single-stage implementation (see Section 7.4) for solving coupled sparse/dense FEM/BEM linear systems such as defined in (1.2). For the purpose of this evaluation, we used the *wide pipe* test case (see Section 1.4.1). We remind the reader that the test case is designed so that we can compute the relative error of the numerical solution (see Section 1.5.2.1). Computation time and RAM usage are measured as described in Section 3.3.2. Like for two-stage algorithms, the reported 'Factorization time' in figures corresponds to the execution time of all the steps of the proposed single-stage prototype algorithm except for the computation of the solution vectors x_s and x_v (see the last two lines of Algorithm 7).

We use double precision accuracy. The precision parameter ϵ for HMAT providing low-rank compression is set to 10^{-3} (see Section 1.5.2.3). `qr_mumps` currently implements no compression mechanism.

We have conducted our experiments on either a `laptop` having an Intel(R) Core i7-8650U with 4 cores running each at 4.2 GHz and 32 GiB of RAM or a single `bora` node on the PlaFRIM platform [19]. A `bora` node has a total of 36 processor cores running each at 2.5 GHz and 192 GiB of RAM. Hyper-Threading and Turbo-Boost are deactivated. The solver test suite is compiled with GNU C Compiler (gcc) 12.2.0, Intel(R) MKL library 19.0.5.281 and StarPU 1.3.8.

In the first part of the study, in Section 7.5.1, we evaluate the impact of the asynchronous API on the single-stage algorithm. Then, in Section 7.5.2, we compare the performance of the prototype single-stage implementation with the two-stage algorithms.

7.5.1 Asynchronous execution

We consider coupled FEM/BEM linear systems with N, the total unknown count, ranging from 100,000 to 2,000,000. The number n_b of blocks per block row and block column of the Schur complement matrices X and S (see Section 7.3) is set to either 10 or 30. Note that in the hereby case of the single-stage algorithm, the n_b parameter is not the same as in the two-stage multi-factorization algorithm (see Section 3.2). In particular, values higher than 1 do not lead to superfluous re-factorizations of the A_{vv} submatrix within the *sparse factorization+Schur* building block. Here, we make only one call to the latter. Finally, the computation is done either in synchronous or in asynchronous execution mode.

Figure 7.3 shows the computation times and Figure 7.4 the corresponding RAM usage peaks. At first, let us focus on the number n_b of Schur complement blocks per block row and block column of S. In the `qr_mumps` solver, this parameter also defines the granularity of parallel tasks. Smaller blocks lead to higher concurrency. However, when the blocks are too small, it can lead to the generation of a large amount of short tasks without enough of workload to compensate the cost of their compression by HMAT as well as the related scheduling cost. When the blocks are too large, it may negatively impact the RAM usage. Just like in the case of the blocking parameters of the two-stage algorithms, there is a trade-off to be found based on different quantities, such as the total unknown count, amount of available RAM and processing units. Here, for smaller problems, i.e. for $N \leq 1{,}000{,}000$, it is better to consider fewer, but larger, blocks (n_b set to 10). However, for a larger problem considering 2,000,000 unknowns, n_b set to 10 leads to physical memory overflow. Operating system then resorts to the swap disk space which degrades the performance. Note that 2,000,000 unknowns is the maximum we can achieve on a `bora` node before reaching its memory limit. As of the execution mode, asynchronous execution seems to be the best performing one. However, in the 2,000,000-unknown case with n_b set to 10, we can observe the negative impact of swapping which makes asynchronous mode lose its advantage over synchronous execution.

Figure 7.5 then shows the relative error for the benchmarks featured in Figure 7.3. The precision parameter ϵ was set to 10^{-3} for the HMAT solver providing low-rank compression. For the problem with 250,000 unknowns, the relative error slightly exceeds the given threshold. In this case, the deviation is not important enough to be considered as significant. This situation does not seem to be specific to the prototype single-stage implementation as it arises also when comparing the latter to other approaches (see further in Section 7.5.2). However, it requires a deeper investigation. Eventually, for all the other benchmarks, the relative error is below the selected threshold 10^{-3} which confirms that the algorithm allows us to reach the expected accuracy.

In figures 7.6 and 7.7, we present the execution traces of respectively a synchronous and an asynchronous execution of the prototype single-stage implementation. In this case, the benchmarks were performed using a `laptop` on a small problem, i.e. counting 50,000 unknowns in total. Each of the figures features two plots. The top plot shows the computation tasks of the `qr_mumps` (blue) and the HMAT (red) solvers executed by each of the processor cores. The bottom plot represents the activity of the underlying StarPU runtime for each of the processor cores. The total execution time is displayed at the right end of the top plot in milliseconds.

On the synchronous execution trace, we distinguish only continuous blocks of either `qr_mumps` or HMAT tasks. The first block of HMAT tasks corresponds to matrix initialization. Follows a block of `qr_mumps` tasks corresponding to the *sparse factorization+Schur* step. Note that the first part of this block represents a symbolic factorization (see sections 1.5.1.2 and 1.6.3) performed in sequential at the beginning of the *sparse factorization+Schur* step. The next block of HMAT tasks represents the compression and the *dense factorization* of the Schur complement matrix S. The blocks at the end match the *sparse factorization* of A_{vv} (including a sequential sym-

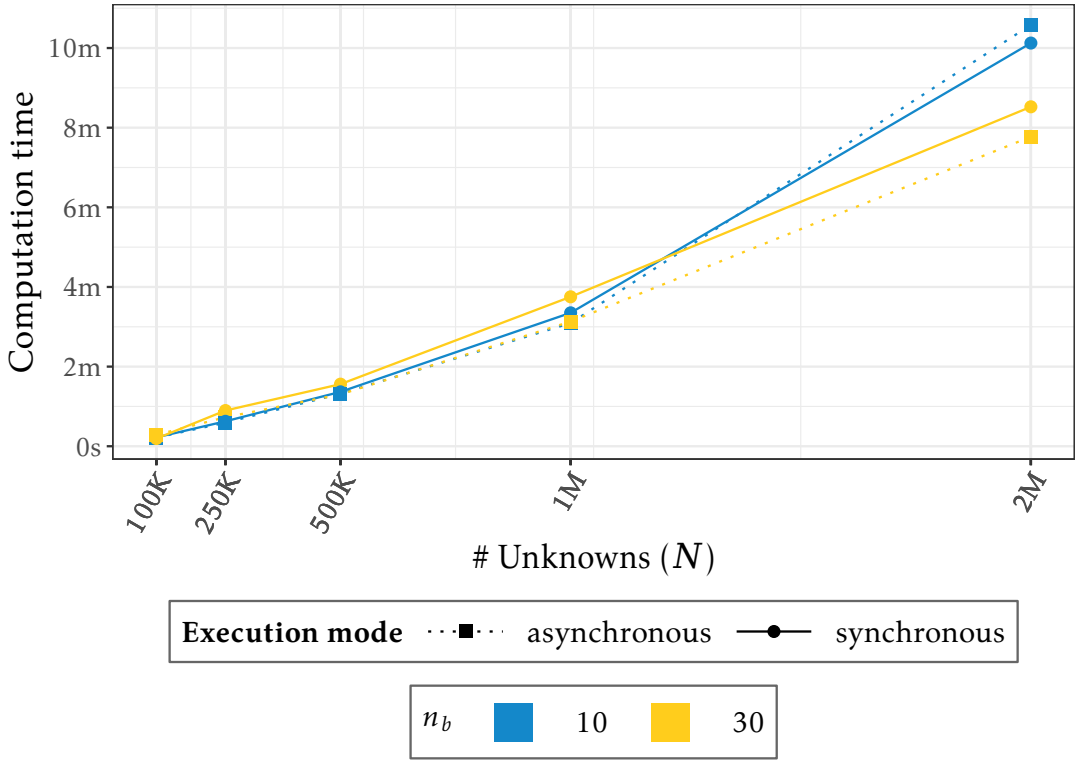


Figure 7.3: Computation times of the **prototype single-stage** implementation using the `qr_mumps`/HMAT solver coupling. Parallel runs using 36 threads on single `bora` node.

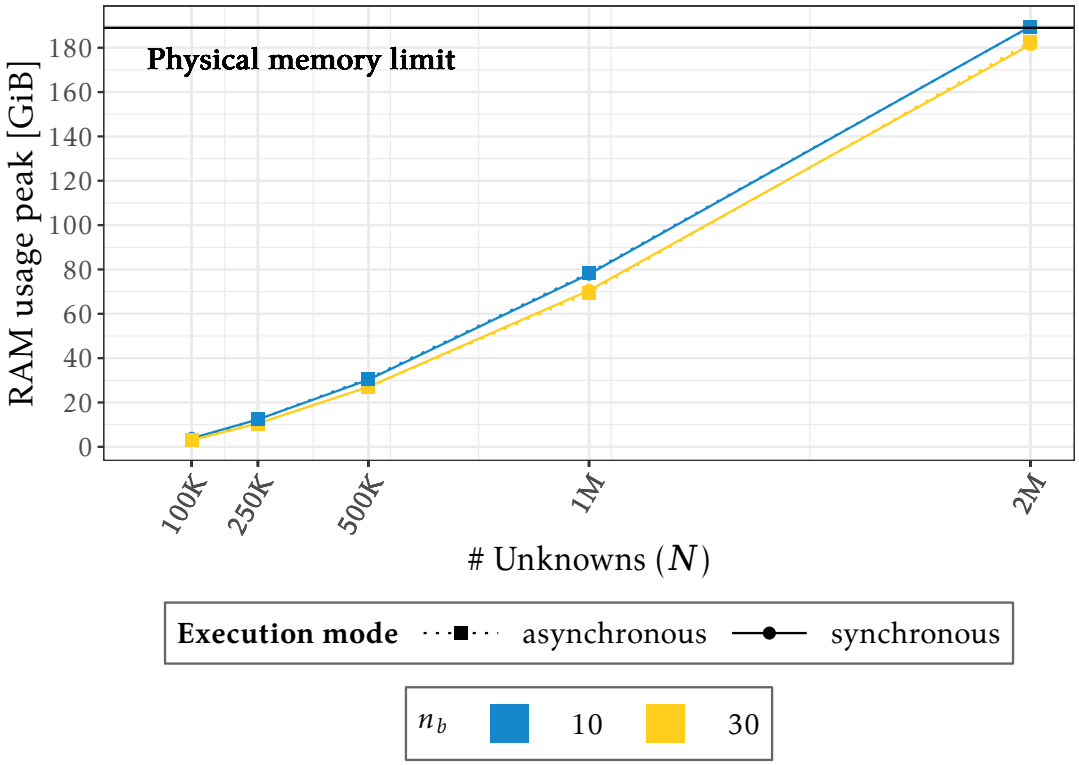


Figure 7.4: RAM usage peaks of the **prototype single-stage** implementation using the `qr_mumps`/HMAT solver coupling. Parallel runs using 36 threads on single `bora` node.

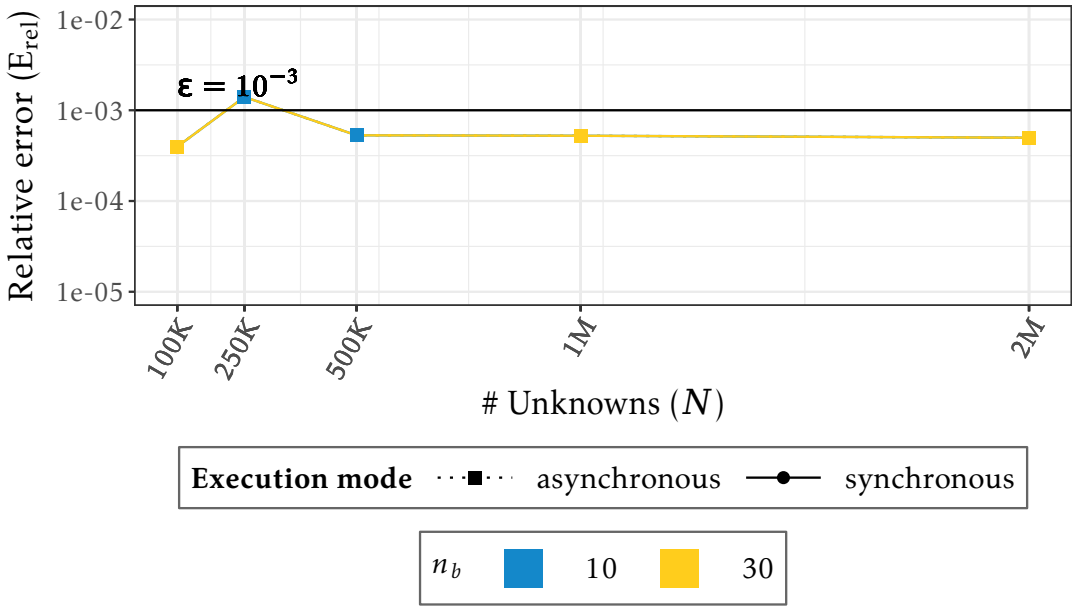


Figure 7.5: Relative error E_{rel} of the **prototype single-stage** implementation using the `qr_mumps`/HMAT solver coupling. Parallel runs using 36 threads on single `bora` node.

bolic factorization) and the final computation steps to obtain x_s. Note that the *dense solve* step of HMAT involved in the computation of x_s is extremely short and virtually invisible on the trace. The computation of x_v does not appear on these traces. On the asynchronous execution trace, the `qr_mumps` block representing the *sparse factorization+Schur* step and the HMAT block corresponding to the compression and the *dense factorization* of S overlap. This illustrates the asynchronous single-stage approach (see Section 7.2). However, the sparse and the dense operations are not as superposed as they could. This can be improved by harmonizing the priorities of HMAT tasks with respect to `qr_mumps`'s ones as well as by investigating alternative scheduling policies. Eventually, from the runtime activity in the case of the synchronous execution, we can see that at the beginning of the block of HMAT tasks representing the compression and the *dense factorization* of S the processor cores were often idle between the different tasks. The runtime was thus spending time in inactivity (sleeping). However, it was possible to prevent these inactivity periods in the case of the asynchronous execution.

7.5.2 Comparison with two-stage algorithms

We again consider coupled FEM/BEM linear systems with N, the total unknown count, ranging from 100,000 to 2,000,000. As of the single-stage implementation, the number of blocks per row and column of the Schur complement matrices X and S (see Section 7.3) is set to either 10 or 30. Regarding the two-stage algorithms (see Chapter 3), we rely on their *compressed Schur* variants using the MUMPS/HMAT coupling. For multi-solve, we set the size n_c of block $A_{sv_i}^T$ of columns of the A_{sv}^T submatrix to 256 and the size n_S of blocks S_i of columns of S to 1,024 (motivated by the results presented in Section 3.3.4.1). For multi-factorization, we set the n_b parameter (see Figure 3.4, p. 46) handling the count of square blocks S_{ij} per block row and block column of the Schur complement submatrix S between 1 and 3. To ensure a fair comparison of the two-stage algorithms to the non fully-featured prototype of the single-stage algorithm, we disable low-rank compression in MUMPS as well as out-of-core in MUMPS and HMAT for the benchmarks below. As a reference, we consider the advanced vanilla solver coupling (see Section 2.2.2). From the implementation point of view, we rely on the *baseline multi-factorization* algorithm with the MUMPS/SPIDO coupling (without compression in the

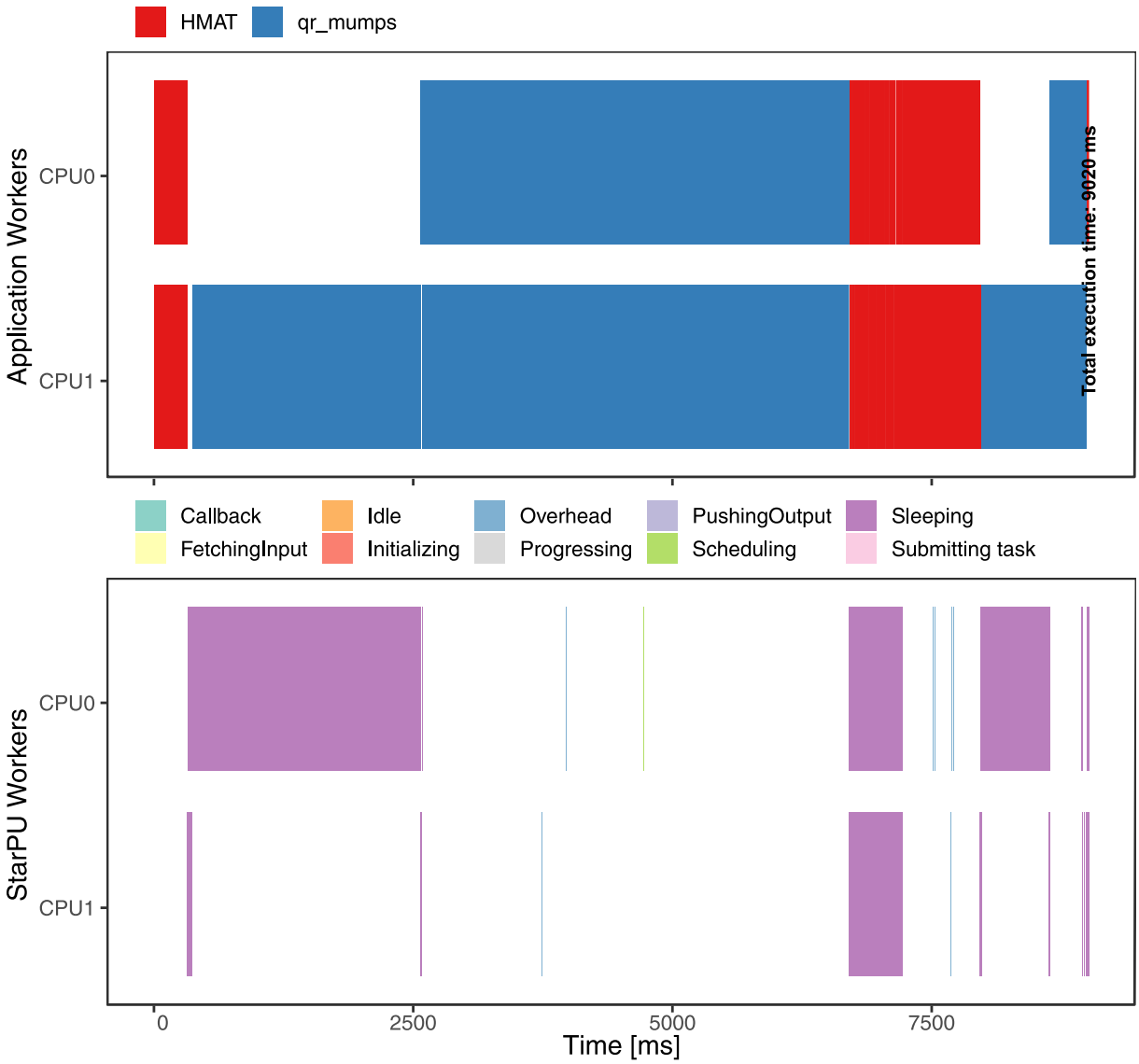


Figure 7.6: Execution trace of the **prototype single-stage** implementation using the `qr_mumps`/HMAT solver coupling. Synchronous parallel runs using 2 threads on a `laptop`. The top plot represents the computation tasks of `qr_mumps` and HMAT. The bottom plot represents runtime activity. Blank spaces correspond to inactivity.

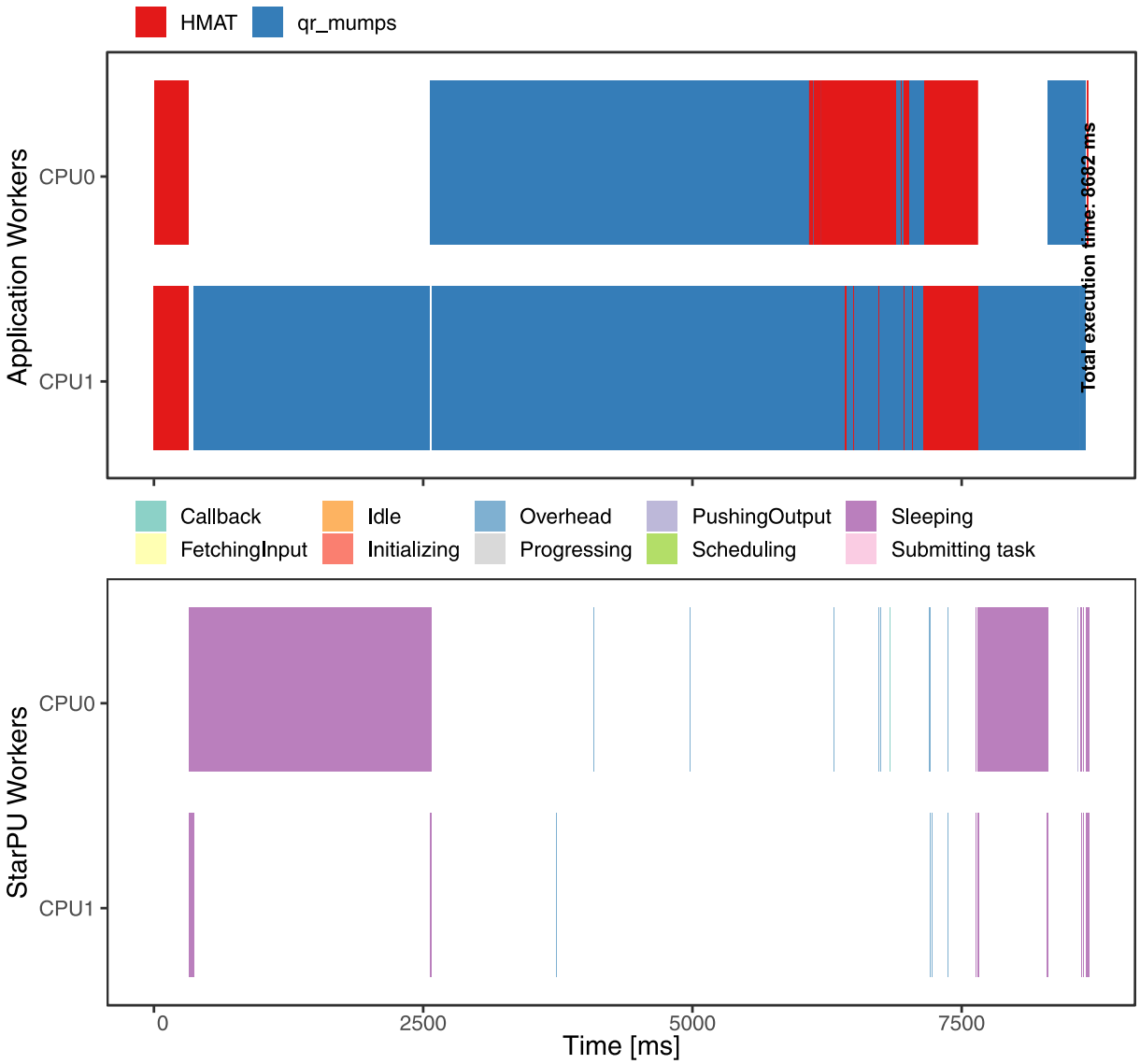


Figure 7.7: Execution trace of the **prototype single-stage** implementation using the `qr_mumps`/HMAT solver coupling. Asynchronous parallel runs using 2 threads on a `laptop`. The top plot represents the computation tasks of `qr_mumps` and HMAT. The bottom plot represents runtime activity. Blank spaces correspond to inactivity.

dense Schur complement part) and n_b set to 1 (see further details in Section 3.3.1.2).

In Figure 7.8, we show the best computation times of all of the evaluated algorithms, parameter variations and problem sizes. Figure 7.9 then shows the corresponding RAM usage peaks. Unlike with multi-factorization and with the advanced vanilla coupling, both the prototype single-stage and the multi-solve algorithms can process FEM/BEM systems with N up to 2,000,000. In terms of computation time, the single-stage implementation outperforms all the other concurrents. Especially, it has a large advance over the multi-solve and the state-of-the-art advanced vanilla coupling. Although multi-solve reported to be the slowest one, it is the less limited of the three approaches regarding RAM usage. It reached only 40 GiB (20 %) of available RAM for $N = 2{,}000{,}000$. The advanced vanilla coupling and multi-factorization went out of memory before arriving at 2,000,000 unknowns. This also represents the limit for the prototype single-stage implementation lacking proper out-of-core feature. Eventually, there is no significant difference between the single-stage prototype and multi-factorization. Indeed, up to $N = 1{,}000{,}000$, multi-factorization could run with n_b set to 1. As we do not consider out-of-core assembly of the Schur complement part, from the algorithmic point of view, the memory requirements of the single-stage prototype are very close to those of multi-factorization with $n_b = 1$ and numerical compression active in the Schur complement part. However, with the Schur complement matrix split into multiple tiles using the n_b parameter (see above), the single-stage implementation can benefit from asynchronous execution and becomes faster than the multi-factorization algorithm.

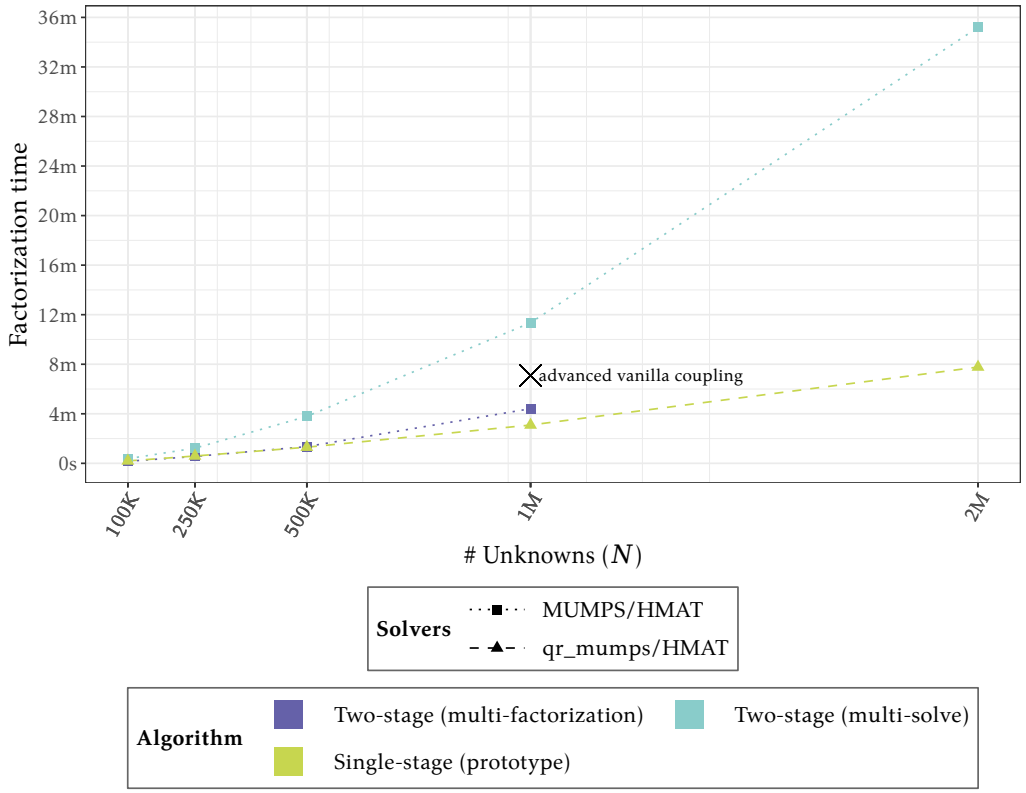


Figure 7.8: Best computation times of **prototype single-stage**, **multi-solve** and **multi-factorization**. Parallel runs using 36 threads on single `bora` node.

Figure 7.10 then shows the relative error for the benchmarks featured in Figure 7.8. The precision parameter ϵ was set to 10^{-3} for the HMAT solver providing low-rank compression. Like in the evaluation of the prototype single-stage implementation in Section 7.5.1, the relative error slightly exceeds the given threshold when $N = 250{,}000$. Still, the deviation remains low to be considered as significant. For all the other benchmarks, the relative error is below the selected

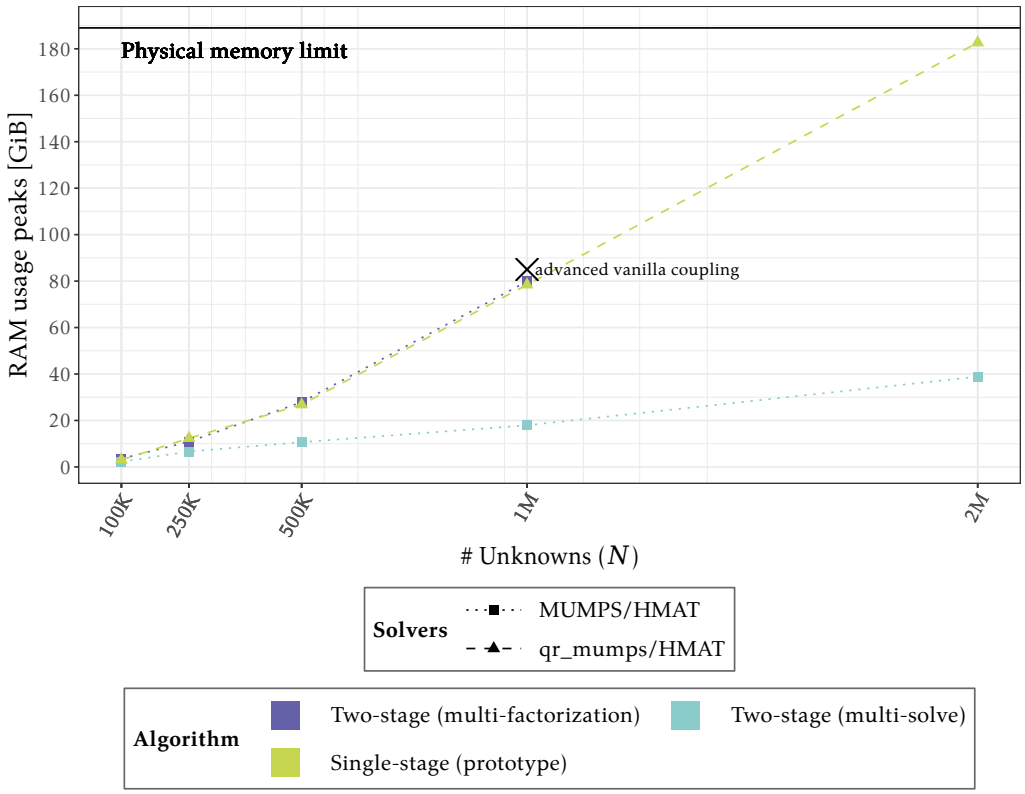


Figure 7.9: RAM usage peaks of **prototype single-stage**, **multi-solve** and **multi-factorization** corresponding to the benchmarks featured in Figure 7.8. Parallel runs using 36 threads on single `bora` node.

threshold 10^{-3}.

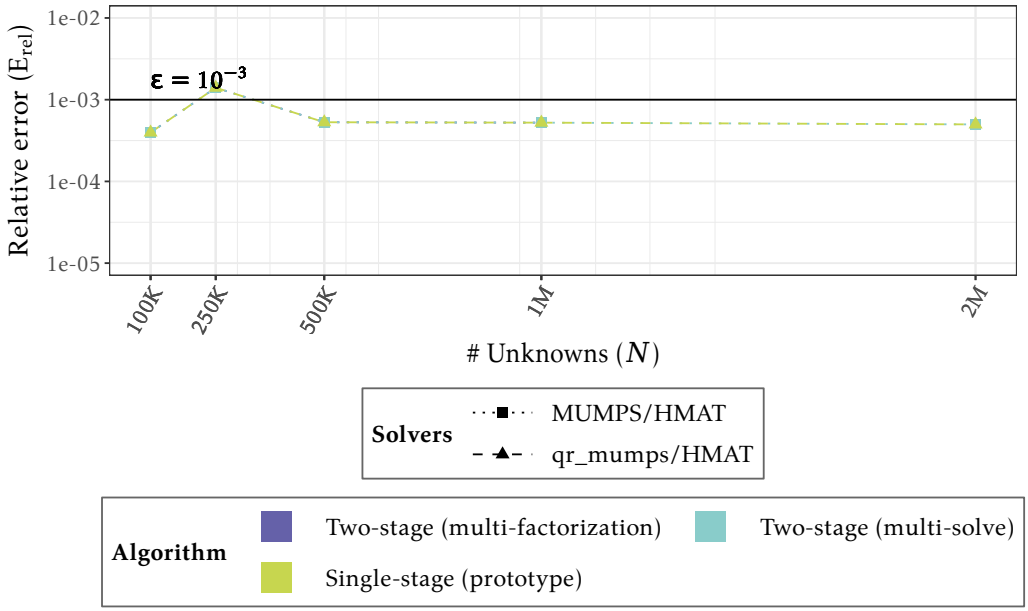


Figure 7.10: Relative error E_{rel} of **prototype single-stage**, **multi-solve** and **multi-factorization** corresponding to the benchmarks featured in Figure 7.8. Parallel runs using 36 threads on single `bora` node.

7.6 Conclusion

The major part of this thesis was dedicated to the two-stage algorithms for solving coupled sparse/dense systems such as (1.2). Relying on the existing API of some fully-featured direct solvers, the main strength of these algorithms is to take advantage of their advanced and well-optimized features (such as the internal management of the Schur complement, compression and out-of-core techniques and distributed-memory parallelism). While multi-solve and multi-factorization reduce the negative impact of the limitations in the state-of-the-art vanilla solver couplings (see Section 2.2), they cannot eliminate them completely and therefore remain suboptimal with respect to the ideal implementation (see Section 1.6.3).

In this chapter, we proposed a single-stage algorithm with the ambition to achieve the ideal approach. The main ideas of this design are to fully benefit from the optimal symmetry and sparsity of the FEM/BEM system, to avoid assembling the entire Schur complement matrix S in RAM and to improve the composability of the sparse and dense operations so as to allow for their asynchronous execution. While the implementation of the single-stage algorithm within fully-featured direct solvers can only be a long-term work due to their complex codebase, we proposed a proof-of-concept in a non fully-featured prototype. Despite missing numerical compression in the sparse parts of the system, proper out-of-core and distributed-memory parallelism, we validated the potential of the asynchronous execution and showed the competitivity of the single-stage approach with respect to the state-of-the-art as well as the two-stage algorithms. Although there is still a long way to go before the single-stage algorithm may represent a serious concurrency to the two-stage algorithms including all of the advanced features of the fully-featured solvers they rely on, this work is meant to serve as a proof-of-concept and an inspiration for future research.

In the short term, the next step would be to enable out-of-core computation of the Schur complement blocks and explore the potential of the approach to process even larger coupled sys-

tems through a memory-aware study. In the long term, it would be interesting to launch a discussion with the development teams of sparse direct solvers about a possible implementation of an analogous approach in fully-featured solvers.

Chapter 8

Reproducibility

Beyond the main contribution related to the solvers for coupled sparse/dense FEM/BEM linear systems, we dedicated a substantial part of this thesis to improving the reproducibility of our work. In computer science in general and in HPC in particular, reproducibility of a research study has always been a complex matter. On the one hand, rebuilding exactly the same software environment on various computing platforms and over extended periods of time may be long, tedious and sometimes virtually impossible to be done manually. On the other hand, while the experimental method is usually explained in research studies, the instructions required to reproduce the latter from A to Z should also be provided in a comprehensive manner.

The topic of reproducible research has been addressed from multiple points of view as well as in multiple scientific domains and applications [59, 88, 62, 123, 60]. In computational sciences, a dedicated peer-reviewed journal promotes and encourages the reproducibility [112]. Throughout this chapter, we address the challenges of ensuring a reproducible research study in computer science in general and in the context of this thesis in particular. We then present the strategy we adopt to face these challenges including working principles, software tools and their alternatives. To share the resulting guidelines, we provide a minimal working example of a reproducible research study on solvers for coupled FEM/BEM systems. Moreover, we introduce and reference examples of such studies from this thesis.

Concretely, in Section 8.1, we discuss the reproducibility of both software environments and numerical experiments as well as the issue of a long-term preservation of all the materials related to a research study, including the source code and reproducing instructions. In Section 8.2, we analyze approaches and tools we can rely on to improve the reproducibility of our scientific production. Then, in Section 8.3, we provide a minimal working example of a reproducible research study and in Section 8.4, we cite examples of such studies from this thesis. Finally, we conclude in Section 8.5.

8.1 Challenges

Regarding the reproducibility of a research study in computer science, we distinguish three main challenges to take on. When we design our working software environment on a given machine, using given revisions of software packages and configurations, it is important to be able to **recreate the exact same conditions on a different machine**, i.e. on a high-performance computing platform for performing experiments. This brings us to the next problem. We often need to perform the experiments more than once, using various software versions or

algorithmic options and maybe months or years later. We thus deem crucial to **maintain a complete documentation**, a sort of laboratory log book, on how to recreate the adequate software environment and perform all the experiments from scratch. The third issue resides in the **availability of all the materials related to a research study in the long term**. Here, we refer especially to the aforementioned documentation, experimental results and associated scientific publications. However, we also have to consider all the source code required to run the experiments, regardless of whether it is our own or an external dependency.

Ensuring software environment reproducibility in the context of this thesis is all the more difficult because of a complex software stack including numerous linear algebra libraries, sparse and dense direct solvers as well as post-processing tools. In addition to that, working with proprietary Airbus source code and external libraries available only in binary form limits our efforts for fully reproducible research from the very beginning. As of the experiments, due to a large spectrum of problem sizes, algorithmic choices and software revisions we want to explore, we face the challenge of maintaining a relatively easily customizable and reproducible benchmark campaign.

8.2 Strategy

To set up or reproduce a software environment, scientists often rely on modules and manual builds. The main issue of this approach is that modules as well as building instructions vary from one machine or system configuration to another. These may provide different package versions or not provide the required software at all. A more convenient solution would be to rely on a package manager, such as Spack [72], allowing for easily defining custom package versions, using different configurations and available compilers. Nevertheless, Spack relies on system-provided compilers and other low-level components of the software dependency tree. This still threatens the reproducibility of the resulting software environment on other machines and over the time. We can bypass this problem by resorting to an autonomous software bundle produced for example using Singularity [95, 63] or Docker [103, 2].

However, the container solution does not alleviate some of the most important concerns. For example, if we want to use a different version of one or more packages, we either have to modify the container interactively, which would make it even less reproducible, or re-build it from scratch, which can take a lot of time if performed regularly. Moreover, because containers are often based on an existing Linux distribution, we are limited to the versions of various core packages (compilers, parallel libraries, linear algebra kernels, solvers, ...) provided by that particular distribution in that particular release unless we want to re-build and re-configure an important portion of the software stack.

Our goal is to cope with these limitations and be able to generate fully controlled software environments that we can easily modify and reproduce. To this end, we propose to explore an approach resorting to an alternative package manager such as Guix [8] or Nix [64]. In this case, we choose to rely on Guix. The reason is twofold. One, there is an important effort to optimize Guix for reproducible scientific workflows in HPC known as Guix-HPC [9]. The second reason is our proximity to the main developers involved in the Guix project.

Guix [8] is a transactional package manager. Using the latter, we can provide a self-contained, executable description of the whole software environment used to run our experiment. This precision is a crucial building block for reproducible scientific workflows, yet it is something READMEs and containers do not even approximate.

What can be seen as a drawback of Guix is that it requires root privileges to be installed. While this is not a practical problem on a personal machine, it represents an important limitation

on high-performance computing clusters for example. To this day, these rarely provide Guix natively. However, in absence of a native Guix installation on the target machine, it is possible to produce a standalone tarball, a Singularity or a Docker image containing the entire environment on the development machine with root privileges, e.g. a personal computer, and transfer it to the target workstation. On computing clusters, Singularity and Docker are currently much more widespread in comparison to Guix. This alternative makes us suffer from some of the disadvantages related to containers, as discussed above, but it allows us to preserve the complete reproducibility of our software environments. Moreover, the container production process in Guix remains fully automated and transparent to the end-user.

When it comes to reproducing the experiments, we usually have to settle for comments in source code and README files to know how to use the scripts dedicated to run experiments and post-process results. Following [123, 107], we choose to write the source code of scripts and various configuration files allowing us to design and automatize numerical experiments in respect of the paradigm known as literate programming [93]. The idea of this approach is to associate source code or commands to execute with a narrative written in a natural language. There are numerous software tools designed for literate programming. We rely on Org mode for the Emacs text editor [65, 6] which defines the Org markup language allowing to combine formatted text, images and figures with source code. An Org document can then be exported to various output formats [16], such as LaTeX, Beamer or HTML, and shared with peers. Org is also suitable for composing final scientific publications. Source code blocks in an Org file may be either exported into a source file [17], e.g. to be compiled, or evaluated on-the-fly [15]. The evaluation may be triggered manually by the user or automatically on export and the result of execution, such as a figure or a return value of a math formula, can then be included in the surrounding formatted text.

Eventually, the scheme adopted for improving the reproducibility of the research studies within this thesis consists in building for each study a standalone version-controlled repository, i.e. a git repository. In the latter, the software environment is defined and handled by Guix and all the local source code, scripts as well as final publications such as articles and research reports are formatted in Org following the literate programming paradigm. Each repository also contains a set of guidelines for reproducing the software environment, running the experiments and producing the final publications.

As an alternative, we could consider the Maneage [32] project. In its philosophy, a reproducible research study is represented by a version-controlled repository. Whereas in our scheme the management of the software environment is delegated to Guix, in this case the study repository also contains the instructions on where to acquire, how to configure and build all the software stack including system libraries and compilers. The minimal requirement of Maneage is a Linux-like system distribution. Compared to Guix, there is no need for root privileges. However, while the build time of the software stack in Guix can be optimized thanks to pre-built binary substitutes of the major part of the software packages, in Maneage, the entire software stack has to be rebuilt on each target machine from scratch. Then, where we use Org mode for composing the experimental results post-processed using dynamically executable code blocks with a narrative to form the resulting scientific publications, Maneage relies on LaTeX documents with macros for dynamic inclusion of experimental results. This choice also makes it difficult to adopt the principles of literate programming, i.e. composing source code, which can be easily extracted, with natural language.

The last, but not least, subject we are concerned with is the long-term availability of our research studies and related source code, including external dependencies. It happens that version-controlled repositories, i.e. git or subversion, hosted on a code forge platform, e.g. Inria Forge or BitBucket, become suddenly unavailable due to the shutdown of the platform for example. This is where the Software Heritage project [21] comes into play. The goal of

the latter is to guarantee the availability of software or other type of source code in a long term by regularly archiving existing repositories and making them available through unique identifiers.

8.3 Minimal working example

The goal of this section is to provide a complete minimal working example of a research study following the principles of reproducibility and using the associated tools discussed in Section 8.2. Whereas in the context of this thesis we partially rely on proprietary source code and libraries, in the below example we build exclusively upon open-source software and test cases so as to allow anyone to reproduce the example research study.

This section is split into two major parts. Section 8.3.1 contains the example research study and Section 8.3.2 the companion document containing the literate description of all the local source code and scripts as well as the instructions for reproducing the entire study and the numerical experiments it features in the original software environment.

Note that the figures and results cited in the text of the experimental part of the example study are produced dynamically thanks to associated code blocks in the Org source of the study manuscript (see Section 8.2). For illustrative purposes, the complete listing of the latter is provided in Section B of the Appendix.

8.3.1 Example study

8.3.1.1 Introduction

This is an example experimental study relying on the `test_FEMBEM` solver test suite. Here, we are especially interested in solving coupled sparse/dense FEM/BEM linear systems arising in the domain of aeroacoustics. The idea is to evaluate the solvers available in the open-source version of `test_FEMBEM` for the solution of this kind of linear systems.

8.3.1.2 Experiments

The open-source version of `test_FEMBEM` [22] does not implement couplings of sparse and dense direct solvers which is the preferred method for solving sparse/dense FEM/BEM systems. Therefore, we process the systems as dense using either the HMAT-OSS or the Chameleon direct solver. HMAT-OSS [10] is an open-source and sequential version of the compressed hierarchical $\mathcal{H}$-Matrix dense direct solver HMAT [101] developed at Airbus. Chameleon [1] is a fully open-source dense direct solver without compression.

As of the test case, we consider a simplified *wide pipe* which is still close enough to real-life models (see Figure 8.1). Note that all the benchmarks were conducted on a single machine equipped with an octa-core Intel(R) Xeon(R) W3520 running at 2.661 GHz and 8 GiB of RAM.

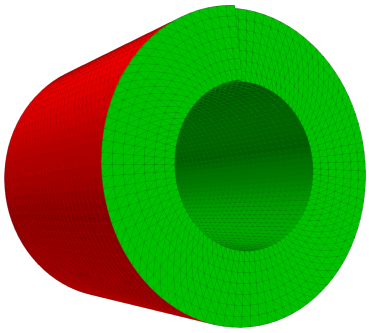


Figure 8.1: A *wide pipe* mesh counting 20,000 vertices.

At first, we want to know to which extent can data compression improve the computation time. For this, we compare sequential executions of both HMAT-OSS, the compressed solver, and Chameleon, the non-compressed solver, on coupled FEM/BEM systems of different sizes (see Figure 8.2). The results clearly show the advantage of using data compression, especially with increasing size of the target linear system.

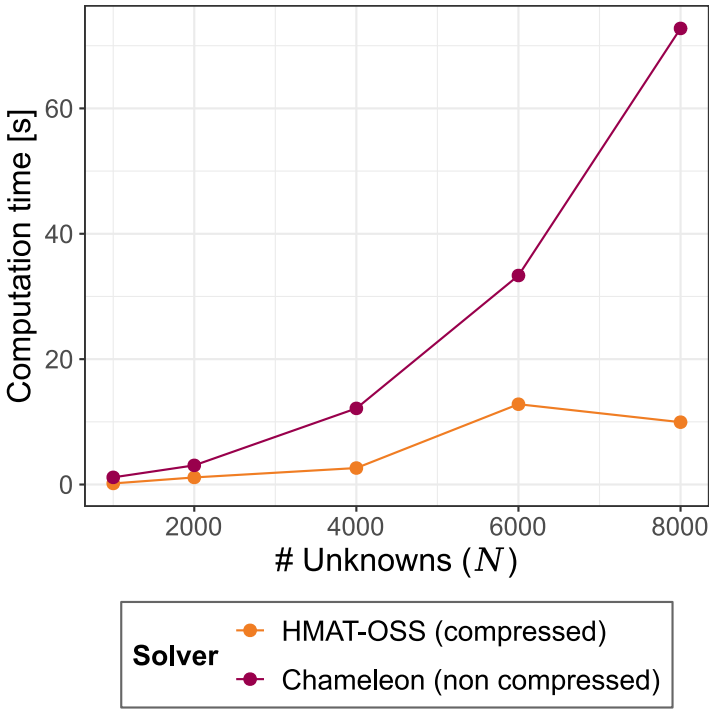


Figure 8.2: Computation times of sequential runs of HMAT-OSS and Chameleon on coupled sparse/dense FEM/BEM linear systems of varying size.

To study the impact of parallel execution on the time to solution, we limit ourselves to the Chameleon solver as HMAT-OSS is sequential-only. In Figure 8.3, we compare the computation times of Chameleon on coupled FEM/BEM systems of different sizes using either one or four threads. According to the results, we can observe a significant decrease in computation time in case of parallel executions. Moreover the parallel efficiency of the run on the largest linear system considered (8000 unknowns) is approximately 77%.

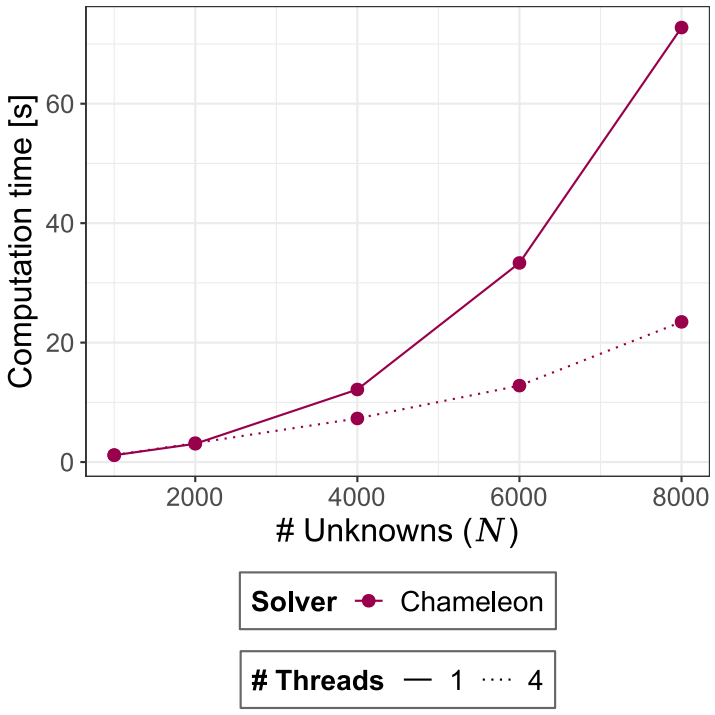


Figure 8.3: Computation times of sequential and parallel runs of Chameleon on coupled sparse/dense FEM/BEM linear systems of varying size.

8.3.1.3 Conclusion

We evaluated the performance of the solvers available in the `test_FEMBEM` test suite on coupled sparse/dense FEM/BEM linear systems. The solvers considered were HMAT-OSS, a sequential compressed dense direct solver and Chameleon, a multi-threaded non-compressed dense direct solver.

The comparison of sequential runs of HMAT-OSS and Chameleon showed an important positive impact of data compression on the time to solution. In addition, the comparison of sequential and parallel runs of Chameleon as well as the computed parallel efficiency showed a considerable speed-up of the parallel execution.

8.3.1.4 Notes on reproducibility

With the aim of keeping the experimental environment of the study reproducible, we manage the associated software framework with the GNU Guix transactional package manager [8]. Moreover, relying on the principles of literate programming [93], we provide a full documentation on the construction process of the experimental environment, the execution of benchmarks, the collection and the visualization of results as well as on producing the final manuscripts in a dedicated technical report associated with this study (see Section 8.3.2). A public companion contains all of the source code, guidelines and other material required for reproducing the study: `https://gitlab.inria.fr/thesis-mfelsoci/dissertation/example-fembem`, archived on `https://archive.softwareheritage.org/` under the identifier `swh:1:snp:d8cdf44424c392dc25564870cfacb9fec5872554`.

8.3.2 Reproducing guidelines

Within our research work, we put strong emphasis on ensuring the reproducibility of the experimental environment. On the one hand, we make use of the GNU Guix transactional package manager allowing for a complete reproducibility of software environments across different machines. On the other hand, we rely on the principles of literate programming in an effort to maintain an exhaustive, clear and accessible documentation of our experiments and the associated environment allowing others to reproduce the study.

The goal of this section is to provide such a documentation of our study entitled 'Solvers for coupled sparse/dense FEM/BEM linear systems' (see Section 8.3.1). Especially, we aim to describe and explain here all of the source code and procedures involved in construction of the experimental software environment, execution of benchmarks, gathering and post-processing of results as well as publishing of the study manuscript.

In Section 8.3.2.1, we provide detailed guidelines on how to reproduce the study and post-process the experimental results. In Section 8.3.2.2, we remind the concept of literate programming and the associated tools. In the beginning of Section 8.3.2.3, we present GNU Guix and how it can be used to build a reproducible software environment. Then in paragraphs Defining benchmarks and Running benchmarks of Section 8.3.2.3, we detail the design of our numerical experiments.

8.3.2.1 Guidelines for reproducing the study

We include here a set of guidelines for reproducing the experimental software environment we describe in this technical report, the experiments and graphical representations of results presented in Section 8.3.1 as well as the corresponding manuscript itself.

We provide instructions for the situation where native Guix is installed on the system. Note that if the target machine where the study should be reproduced misses Guix or the latter can not be installed there, e.g. a high-performance computing platform, it is possible to create a Singularity image, a Docker bundle or a relocatable standalone tarball containing the required software environment using Guix on compatible computer and transfer the result to the target machine [12].

All the materials necessary to reproduce the study reside in a dedicated Git repository at `https://gitlab.inria.fr/thesis-mfelsoci/dissertation/example-fembem`. In case of unavailability of the repository at the aforementioned address, it has been archived on the Software Heritage platform (`https://archive.softwareheritage.org/`) under the identifier `swh:1:dir:2f1ff4ffd940332c2e7480146ddb67d24be82cd2` with no time limit on its conservation.

At first, we clone the repository on the target machine and navigate to it. To clone from its original location, we use the standard Git command.

```
git clone https://gitlab.inria.fr/thesis-mfelsoci/dissertation/example-fembem
cd example-fembem
```

At first, we need to extract source code from Org files (see Section 8.3.2.2).

```
guix shell --pure git emacs emacs-org -- emacs --batch --no-init-file -l org \
     --eval '(progn (setq org-src-preserve-indentation t) (dolist (file
     ↪  (directory-files-recursively "." "\\.org$")) (org-babel-tangle-file
     ↪  file)))'
```

In order to reproduce the experimental software environment, we rely on a list of channels (see Channels in Section 8.3.2.3), a manifest file (see Environment specification in Section 8.3.2.3) and then the combination of the `guix time-machine` and the `guix shell` commands. This is followed by the command for launching the experiments which will be executed in that environment.

```
guix time-machine -C channels.scm -- shell --pure -m manifest.scm -- \
     ./benchmarks/run.sh -d ./benchmarks/definitions.csv -o ./benchmarks/results
```

Finally, we produce the PDF files corresponding to the study manuscript and the hereby guidelines. On export of the Org source of the study manuscript, the experimental results will be post-processed thanks to the R code blocks within the document.

```
guix time-machine -C channels.scm -- shell --pure -m manifest.scm -- \
     emacs --batch --no-init-file --load publish.el \
     --eval '(org-publish "example-fembem")'
```

8.3.2.2 Literate programming

We choose to write the source code of scripts and various configuration files allowing us to design and automatize numerical experiments in respect of the paradigm known as literate programming [93]. The idea of this approach is to associate source code with an explanation of its purpose written in a natural language.

There are numerous software tools designed for literate programming. We rely on Org mode for Emacs [65, 6] which defines the Org markup language allowing to combine formatted text, images and figures with traditional source code. Files containing documents written in Org mode should end with the `.org` extension.

Extracting a compilable or interpretable source code from an Org document is called tangling [17]. It is also possible to evaluate a particular source code block directly from the Emacs editor [15] while editing. For example, this is particularly useful for the visualization of experimental results.

Eventually, an Org document can be exported to various output formats [16] such as LaTeX or Beamer, HTML and so on. This is done using a custom Emacs Lisp publishing script.

Publishing script The Emacs Lisp script described in this section allows for transforming Org documents in the repository into LaTeX PDF documents.

At first, we load packages for:

- Org mode support,

  ```
  (require 'org)
  ```

- evaluation of R source code blocks,

  ```
  (require 'ess)
  ```

- publishing functions,

```
(require 'ox-publish)
```

- LaTeX publishing functions,

```
(require 'ox-latex)
```

- bibliography support.

```
(require 'org-ref)
```

We have to force publishing of unchanged files to make sure all the documents get exported. Otherwise, the files considered unmodified based on Org timestamps are not published even if they were previously deleted from the publishing directory.

```
(setq org-publish-use-timestamps-flag nil)
```

Also, we want to use a custom LaTeX publishing command.

```
(setq org-latex-pdf-process (list "latexmk --shell-escape -f -pdf %f"))
```

Then, we need to load languages for code block evaluation,

```
(org-babel-do-load-languages
 'org-babel-load-languages
 '((shell . t)
   (R . t)))
```

prevent the publishing back-end from prompting for code block evaluation on export

```
(setq org-confirm-babel-evaluate nil)
```

and enable code block evaluation.

```
(setq org-export-babel-evaluate t)
```

Regarding the formatting and the visual aspect of code blocks, we want to:

- preserve indentation on export and tangle

```
(setq org-src-preserve-indentation t)
```

- use TAB to indent code blocks,

```
(setq org-src-tab-acts-natively t)
```

- customize the appearance of code blocks.

```
(setq my-minted-options
      '(("linenos = false") ("mathescape") ("breaklines")
        ("bgcolor = yellow!15")))
(setq org-latex-packages-alist '())
(add-to-list 'org-latex-packages-alist '("" "minted"))
(setq org-latex-listings 'minted)
(setq org-latex-minted-options my-minted-options)
```

At the end, we configure PDF document publishing.

```
(setq org-publish-project-alist
      (list
       (list "study"
             :base-directory "."
             :exclude ".*"
             :include ["study.org"]
             :base-extension "org"
             :publishing-function '(org-latex-publish-to-pdf)
             :publishing-directory ".")
       (list "guidelines"
             :base-directory "."
             :exclude ".*"
             :include ["reproducing-guidelines.org"]
             :base-extension "org"
             :publishing-function '(org-latex-publish-to-pdf)
             :publishing-directory ".")
       (list "example-fembem"
             :components '("study" "guidelines"))))
(provide 'publish)
```

8.3.2.3 Building reproducible software environments

To keep our software environment bit-for-bit reproducible, we rely on the GNU Guix [8] package manager. Guix is a transactional package manager and a stand-alone GNU Linux distribution where each user can install its own packages without any impact on the others and with the possibility to switch between multiple system or package versions. An environment created using Guix is fully reproducible across different computing platforms. In other words, a package built on a computer with a given commit remains the same when rebuilt on another machine.

Channels Software packages in Guix are available through dedicated Git repositories, called channels, containing package definitions. The first and default channel is the system one providing Guix itself as well as the definitions of some commonly used packages such as system libraries, compilers, text editors and so on. Afterwards, we need to include additional channels using a custom channel file `channels.scm`. For each channel, we specify the commit of the associated repository to acquire. This way, we make sure to always build the environment using the exact same versions of every single package in the system and guarantee a better reproducibility. Following the system channel, we include also `guix-hpc` providing various scientific software and libraries.

```
(list
 (channel
  (name 'guix)
  (url "https://git.savannah.gnu.org/git/guix.git")
```

```
  (commit "eb34ff16cc9038880e87e1a58a93331fca37ad92"))
 (channel
  (name 'guix-hpc)
  (url "https://gitlab.inria.fr/guix-hpc/guix-hpc.git")
  (commit "7506a50557beca54903fea496f3185b86c354e35")))
```

Environment specification To enter a particular software environment using Guix, we use the `guix shell` command. Options of the latter allows us to specify the packages to include together with the desired version, commit or branch. To avoid typing long command lines anytime we want to enter the environment, we use a Guix manifest file [7] describing the packages to include into the target environment using a Scheme language expression.

```
(packages->manifest
  (list (specification->package "test_FEMBEM")
        (specification->package "openmpi")
        (specification->package "openssh")
        (specification->package "sed")
        (specification->package "which")
        (specification->package "grep")
        (specification->package "coreutils")
        (specification->package "bash")
        (specification->package "python-pygments")
        (specification->package "python@3")
        (specification->package "texlive")
        (specification->package "r")
        (specification->package "r-ggplot2")
        (specification->package "inkscape")
        (specification->package "emacs")
        (specification->package "emacs-org")
        (specification->package "emacs-biblio")
        (specification->package "emacs-org-ref")
        (specification->package "emacs-ess")
        (specification->package "sed")
        (specification->package "which")
        (specification->package "grep")))
```

Defining benchmarks Within the experimental part of the study, we want to run the benchmark cases specified in the comma-separated values `definitions.csv` file in Listing 1. The latter features six columns:

1. `system_type` indicating the type of linear system (`fem`, `bem` or `fembem`),
2. `solver` specifying the solver to use (`hmat` or `chameleon`),
3. `threads` giving the number of threads to use for computation,
4. `nbpts` giving the number of unknowns in the system, i.e. the size of the system,
5. `arith` defining the arithmetic and precision of matrix coefficients (`s` for simple real, `d` for double real, `c` for simple complex or `z` for double complex),
6. `sym` indicating the symmetry of the coefficient matrix (`1` for symmetric or `0` for non-symmetric).

We then pass the `definitions.csv` file to the `run.sh` shell script (see Running benchmarks) we use to execute all the benchmarks.

```
system_type,solver,threads,nbpts,arith,sym
fembem,hmat,1,1000,z,1
fembem,hmat,1,2000,z,1
fembem,hmat,1,4000,z,1
fembem,hmat,1,6000,z,1
fembem,hmat,1,8000,z,1
fembem,chameleon,1,1000,z,1
fembem,chameleon,1,2000,z,1
fembem,chameleon,1,4000,z,1
fembem,chameleon,1,6000,z,1
fembem,chameleon,1,8000,z,1
fembem,chameleon,4,1000,z,1
fembem,chameleon,4,2000,z,1
fembem,chameleon,4,4000,z,1
fembem,chameleon,4,6000,z,1
fembem,chameleon,4,8000,z,1
```

Listing 1: `definitions.csv` file defining benchmark cases to run within the experimental part of the study.

Running benchmarks To run the benchmarks from the definition file introduced in Defining benchmarks, we rely on a shell script named `run.sh` described below. The script begins with a help function that can be triggered using the `-h` option.

```
function help() {
  echo "Run test_FEMBEM benchmarks defined in FILE." >&2
  echo "Usage: $(basename $0) [options]" >&2
  echo >&2
  echo "Options:" >&2
  echo "  -h                Print this help message." >&2
  echo "  -d FILE           Run the benchmarks defined in FILE. This is a" \
       "mandatory option." >&2
  echo "  -e EXECUTABLE     Set test_FEMBEM executable to EXECUTABLE. The" \
       "default value is 'test_FEMBEM'." >&2
  echo "  -o FOLDER         Run the benchmarks, then store the logs and" \
       " results in FOLDER. The default value is '$(pwd)'." >&2
}
```

Then, we parse the arguments. `run.sh` recognizes four options:

```
DEFINITIONS=""
TEST_FEMBEM="test_FEMBEM"
OUTPUT="$(pwd)"

while getopts ":hd:e:o:" option;
do
  case $option in
```

1. `-d`, a mandatory option used to indicate the path to a benchmark definition file,

```
d)
  DEFINITIONS=$OPTARG

  if test ! -f $DEFINITIONS;
  then
    echo "Error: '$DEFINITIONS' is not a valid file!" >&2
```

```
    exit 1
  fi
  ;;
```

2. `-e` allowing one to specify an alternative path to the `test_FEMBEM` executable,

```
e)
  TEST_FEMBEM=$OPTARG

  if test ! -x $TEST_FEMBEM;
  then
    echo "Error: '$TEST_FEMBEM' is not a valid executable!" >&2
    exit 1
  fi
  ;;
```

3. `-h` which can be used to show a help message on how to use the script,

```
h)
  help
  exit 0
  ;;
```

4. `-o` used to specify an output folder to store benchmark results in.

```
o)
  OUTPUT=$OPTARG

  if test ! -d $OUTPUT;
  then
    echo "Error: '$OUTPUT' is not a valid folder!" >&2
    exit 1
  fi
  ;;
```

We also need to handle:

- the unknown option case,

```
\?) # Unknown option
  echo "Arguments mismatch! Invalid option '-$OPTARG'." >&2
  echo
  help
  exit 1
  ;;
```

- the missing option argument case and

```
:) # Missing option argument
  echo "Arguments mismatch! Option '-$OPTARG' expects an argument!" >&2
  echo
  help
  exit 1
  ;;
```

- any other situation.

```
    *)
      help
      exit 1
      ;;
  esac
done
```

Once the options have been parsed, we check if a benchmark definition file was specified.

```
if test "$DEFINITIONS" == "";
then
  echo "Error: No benchmark definition file specified!" >&2
  exit 1
fi
```

Before parsing the definition file, we need to determine the number of lines in the latter.

```
DEFINITIONS_NBLINES=$(wc -l $DEFINITIONS | cut -d ' ' -f 1)
```

If the file ends with a newline, we do not want to count that empty line.

```
if test $(tail -c 1 $DEFINITIONS | wc -l) -gt 0;
then
  DEFINITIONS_NBLINES=$(expr $DEFINITIONS_NBLINES - 1)
fi
```

Then, we finally read the definition file but without the header.

```
DEFINITIONS_WITHOUT_HEADER=$(cat $DEFINITIONS | tail -n $DEFINITIONS_NBLINES)
```

However, we extract and keep the header to reuse it later when creating an output results file.

```
HEADER=$(cat $DEFINITIONS | head -n 1)
```

We now create the output folder, if it does not exist yet, and navigate to it. Although, we store the current working directory path at first.

```
CWD=$(pwd)
mkdir -p $OUTPUT
cd $OUTPUT
```

Afterwards, we can prepare a `*.csv` file to store benchmark results in. We call it `results.csv`. It will have the same columns as the definition file plus the columns for computation times and relative error.

```
RESULTS="results.csv"
echo "$HEADER,tps_facto,tps_solve,error" > $RESULTS
```

Finally, we initialize a counter of failed benchmarks and iterate over all the lines read from the benchmark definition file to:

```
FAILURES=0

for line in $DEFINITIONS_WITHOUT_HEADER;
do
```

- extract benchmark parameters from the current line,

```
SYSTEM_TYPE=$(echo $line | cut -d ',' -f 1)
SOLVER=$(echo $line | cut -d ',' -f 2)
THREADS=$(echo $line | cut -d ',' -f 3)
NBPTS=$(echo $line | cut -d ',' -f 4)
ARITH=$(echo $line | cut -d ',' -f 5)
SYM=$(echo $line | cut -d ',' -f 6)
```

- choose a log file name based on the above values,

```
LOG="$SYSTEM_TYPE-$SOLVER-$THREADS-$NBPTS-$ARITH-$SYM.log"
```

- keep a unique benchmark name for later,

```
BENCHMARK="$SYSTEM_TYPE, $SOLVER (threads: $THREADS, unknowns: $NBPTS,"
BENCHMARK="$BENCHMARK arithmetic: $ARITH, symmetric: $SYM)"
```

- transform the parameters to `test_FEMBEM` options,

```
SYSTEM_TYPE="--$SYSTEM_TYPE"
SOLVER="-solve$SOLVER"

if test "$SOLVER" == "hmat";
then
  SOLVER="--hmat --hmat-eps-assemb 1e-3 --hmat-eps-recompr 1e-3 $SOLVER"
fi

NBPTS="-nbpts $NBPTS"

ARITH="-$ARITH"

if test $SYM -eq 1;
then
  SYM="--sym"
else
  SYM="--nosym"
fi
```

- run `test_FEMBEM` with the given parameters,

```
echo -n "Running $BENCHMARK..."
OMP_NUM_THREADS=1 MKL_NUM_THREADS=1 STARPU_NCPU=$THREADS $TEST_FEMBEM \
$SYSTEM_TYPE $SYM $ARITH $NBPTS $SOLVER > $LOG 2>&1
```

- parse computation times and relative error from the log file on success and append them to the results file or

```
if test $? -eq 0;
then
  TPS_FACTO=$(cat $LOG | grep -E "<PERFTESTS> TpsCpu.*Facto" | \
                cut -d '=' -f 2 | sed 's/[^0-9.]//g')
  TPS_SOLVE=$(cat $LOG | grep -E "<PERFTESTS> TpsCpuSolve" | \
                cut -d '=' -f 2 | sed 's/[^0-9.]//g')
  ERROR=$(cat $LOG | grep -E "<PERFTESTS> Error" | \
            cut -d '=' -f 2 | sed 's/[^-0-9e.+]//g')
  echo "$line,$TPS_FACTO,$TPS_SOLVE,$ERROR" >> $RESULTS
  echo "Done"
else
```

- show an error message and increase the failed benchmark counter on failure.

```
  echo "Failed"
    FAILURES=$(expr $FAILURES + 1)
  fi
done
```

At the end of benchmark execution we show global results,

```
echo
echo "Successful benchmarks: $(expr $DEFINITIONS_NBLINES - $FAILURES)"
echo "Failed benchmarks: $FAILURES"
```

perform some cleaning,

```
rm -f testHMAT_matrix.json
```

restore the initial working directory and exit.

```
cd $CWD
exit 0
```

8.4 Examples from this thesis

This section refers to two examples of research studies the results of which are parts of this thesis. The studies were conducted according to the principles of reproducibility and using the tools promoted in Section 8.2.

[29] was our first study. An accompanying technical report [30], similar to the example in Section 8.3.2, was released describing all the experimental software stack and providing all the guidelines necessary to reproduce the study provided that the user has access to the proprietary Airbus source code.

[31] is a more elaborate example. The version-controlled git repository, containing both the study and the reproducing instructions, is better structured and archived on Software Heritage under the identifier `swh:1:dir:c512a02f28164a47fee5af715b642b28b2f9f528`.

8.5 Conclusion

This manuscript itself has been composed in Org mode in a git repository. Besides the manuscript and the associated static files such as figures, the repository also contains the specification of the software environment using Guix which is required to compile the source Org file into the present document through LaTeX.

Since the beginning of this thesis and with reproducibility in mind, we experienced Guix, Org mode but also other tools mentioned in Section 8.2. The way we make use of Guix, Org mode and more recently of Software Heritage varies and evolves over time. In this chapter, we advocate our current methods and principles regarding the reproducibility of research studies in computer science. By sharing these, we hope to draw more attention to this topic and provide the scientific community with elements that may help to improve the reproducibility of scientific production.

Conclusion and perspectives

This thesis deals with direct methods for solving large coupled sparse/dense FEM/BEM linear systems of equations arising in an industrial context of aeroacoustic simulations. Due to their size, the systems cannot be processed through a straightforward coupling of the state-of-the-art sparse and dense direct solvers.

To tackle this problem, we proposed two new classes of algorithms, namely the two-stage methods multi-solve and multi-factorization, designed to bypass the limitations in vanilla sparse and dense direct solver couplings from the state of the art. These algorithms thus allow us to benefit from numerical compression techniques within the solver building blocks themselves as well as on the link between the sparse and the dense operations. To validate the algorithms, we performed an experimental study involving both academic and industrial aeroacoustic problems. The results show that thanks to numerical compression the algorithms make it possible to process significantly larger coupled FEM/BEM systems than vanilla coupling approaches allow for on a given shared-memory multi-core machine. We furthermore showed that the algorithms can take advantage of the available memory to increase their performance, in a memory-aware fashion.

Thanks to the introduction of out-of-core computation into the multi-solve and multi-factorization methods, we further reduced the memory requirements of the proposed algorithms. We are now able to process even larger coupled FEM/BEM systems on a single workstation. Furthermore, in the case of the multi-factorization method, this also allows for accelerating the computation. Finally, we extended the two-stage algorithms to distributed-memory parallelism.

In summary, compared to a reference state-of-the-art approach, the proposed algorithms make it possible to process coupled FEM/BEM systems up to 7× larger (14 millions of unknowns in total against 2 millions) on a single shared-memory multi-core machine and more than 6.5× larger (40 millions of unknowns in total against 6 millions) in a distributed-memory environment.

We also studied the energetic profile of the two-stage methods within a multi-metric study. Experimental results established that the usage of numerical compression techniques is worth from an energetic perspective too. The profiles of the processor and memory power consumption together with the memory usage and flop rate allowed us to better understand the behavior of the application. Eventually, the study allowed us to identify a major performance bottleneck in our implementation as well as a potential load balancing issue in the sparse direct solver.

Despite allowing us to process considerably larger coupled systems compared to the state-of-the-art vanilla couplings, the two-stage methods remain suboptimal. Also, the issues brought to light by the multi-metric study of the algorithms led the proposition of an alternative single-stage coupling scheme. We presented a proof-of-concept of such an implementation relying on task-based sparse and dense direct solvers using the same runtime which favors more efficient data sharing between the building blocks of the sparse and the dense solver as well

as asynchronous computations. A preliminary comparative experimental study validated our implementation by confirming that it can reach the target solution accuracy. Furthermore, we illustrated the potential benefit of an asynchronous execution and showed that even a proof-of-concept of this approach can compete with as well two-stage methods as state-of-the-art vanilla couplings.

This work offers multiple promising prospects. Regarding the two-stage methods, the next step would be to deploy the parallel distributed multi-solve and multi-factorization algorithms at Airbus with the aim to process even more demanding industrial simulations than before.

Regarding the single-stage approach, in the short term, we should focus on enabling the out-of-core computation of the Schur complement preventing us from processing significantly larger coupled systems in the proof-of-concept. In the long term, it would be interesting to launch a discussion with the development teams of sparse direct solvers about a possible implementation of an analogous approach in fully-featured solvers.

Then, the single-stage scheme may not be necessarily implemented using a coupling of two distinct solvers, i.e. a sparse and a dense solver. Another possible approach would be to consider a unique solver capable of applying either sparse or dense operations based on the partitioning of the input coupled FEM/BEM system. This would allow for the most straightforward interoperability between the sparse and the dense building blocks in terms of both data sharing and task execution. Recent developments in the HMAT solver [71] make the solver a good candidate for this kind of implementation.

Other perspectives for future research in this domain have more general context. In this thesis, we focused on symmetric coupled systems. However other physical models may lead to non-symmetric systems. Adjusting the proposed algorithms and exploring their performance on non-symmetric systems may enlarge their scope of application. Moreover, we could evaluate other finite element methods in the corresponding sparse part of the coupled linear system such as discontinued Galerkin [53, 113, 47] or high-order finite elements [121]. This would likely have an impact on the proportion of FEM-related unknowns in the system and could change the conclusions regarding the best-performing algorithms. Furthermore, if we do not limit ourselves only to direct solution methods, there is a large scale of iterative approaches to explore. One would then need to address the questions regarding the choice of the iterative solvers, sparse and dense preconditioners and so on.

Appendix

A FEM-only and BEM-only linear systems

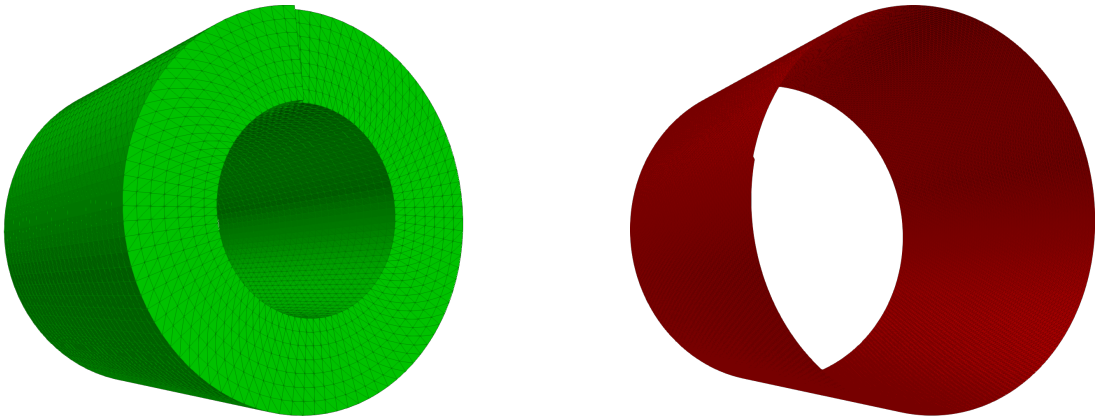
To implement the proposed algorithms for solving coupled sparse/dense FEM/BEM linear systems (see Section 1.3), we resort to different sparse and dense direct solvers (see Section 2.6.4). In this section, we present a preliminary experimental study the goal of which is to benchmark these solvers separately on purely sparse linear systems arising from FEM discretization (see Section A.1) and then on purely dense linear systems arising from BEM discretization (see Section A.2). This separated evaluation allow us to detect their performance specificities for a better understanding of their behavior when committed to solve coupled FEM/BEM linear systems. In other words, after isolating performance issues linked to the individual solution of FEM and BEM systems, we are more likely to identify the best performing coupled solver configurations while focusing only on the parameters specific to the latter.

For the purposes of this evaluation, we used test cases based on the *wide pipe* (see Section 1.4.1) but leading to either purely FEM or purely BEM linear systems (see Figure 8.4). Unlike in the case of coupled FEM/BEM systems (see Section 1.6.3), here, the solution process simply amounts to a *sparse factorization* step (FEM-only systems) or *dense factorization* step (BEM-only systems) followed by a *sparse solve* step or a *dense solve* step, respectively. We remind the reader that the test cases are designed so that we can compute the relative error of the computed solution instead of only estimating it (see Section 1.5.2). We consider complex coefficients in double precision accuracy. MUMPS and HMAT both provide low-rank compression and expose a precision parameter ϵ set to 10^{-3} or 10^{-6} when using the compression feature (see Section 1.5.2.3). Measurements of computation time and RAM usage are done as described in Section 3.3.2 of Chapter 3. Like RAM consumption, disk usage is monitored by the `rss.py` Python script but through the `du` command [3]. Computation time is reported separately for the factorization and the solve steps.

We have conducted our experiments on a single `miriel` node on the PlaFRIM platform [19]. A `miriel` node has a total of 24 processor cores (in 2 sockets and 4 NUMA subnodes) running each at 2.5 GHz and 128 GiB of RAM. It is the policy of the platform to deactivate Hyper-Threading and Turbo-Boost in order to improve the reproducibility of the experiments. The solver test suite is compiled with GNU C Compiler (gcc) 9.3.0, OpenMPI 4.0.3, Intel(R) MKL library 2019.1.144, and MUMPS 5.2.1.

On one hand, we vary a set of solver parameters considering a fixed parallel configuration and observe the evolution of the computation time, relative error E_{rel} of the solution approximation (see Section 1.5.2), RAM and disk usage peaks according to the unknown count and ϵ. For these tests, all the matrices are stored in memory, i.e. the out-of-core feature of the sparse and dense solvers (see Section 2.6.1), when available, was not used.

On the other hand, we consider linear systems having a fixed count of unknowns and evaluate the scalability of different solvers for various parallel configurations listed in Table 8.1.



(a) *wide pipe* considering only a volume mesh *v* and leading to FEM-only linear systems.

(b) *wide pipe* considering only a surface mesh *s* and leading to BEM-only linear systems.

Figure 8.4: *wide pipe* test cases for FEM-only and BEM-only benchmarks.

# MPI processes	Binding	# threads
1	node	1, 6, 12, 18 or 24
2	socket	1, 2, 4, 8 or 12
4	NUMA subnode	1, 2, 4 or 6
1, 6, 12, 18 or 24	core	1

Table 8.1: Complete list of parallel configurations considered for the study in Section A.

A.1 FEM systems

At first, we study the performance of the direct solvers MUMPS and HMAT (see sections 2.6.4.1 and 2.6.4.2) on sparse linear systems resulting from FEM discretization and counting from 250,000 up to 10,000,000 unknowns. Here, we thus rely on the *wide pipe* test case variant in Figure 8.4a.

The hierarchical matrix structure as well as its implementation in the HMAT solver were primarily intended for dense matrices. Indeed, according to the results featured in figures 8.5 and 8.6, MUMPS seems to perform better both in terms of computation time and memory consumption. For illustration, HMAT is unable to process systems with 4,000,000 unknowns and more due to memory limitations while MUMPS can go up to 10,000,000 unknowns provided that ϵ is set to 10^{-3}.

Unlike in the case of HMAT, different low-rank compression thresholds do not significantly impact the performance of MUMPS in terms of computation time. On the other hand and as expected, the more precision we ask for, the more memory the solver consumes. This is true both for MUMPS and HMAT. Although, the difference in memory consumption of MUMPS is more significant only when the compression is completely disabled (see Figure 8.6). Finally, the nearly non-existent disk space consumption of both MUMPS and HMAT confirms that the out-of-core computation has been disabled.

HMAT features an implementation prototype of the Nested Dissection (ND) algorithm (see Section 2.6.4.2) which is a heuristic reordering technique allowing to reduce matrix fill-in resulting from factorization (see Section 1.5.1.2) with the aim to reduce solver's memory requirements and improve its performance. Due to current implementation limitations, HMAT is able to apply the algorithm only on non-symmetric matrices. Moreover, a considerably higher memory consumption of the solver when using ND does not allow us to test cases with more than 250,000 unknowns on `miriel` nodes having 128 GiB of RAM. Note that more advanced

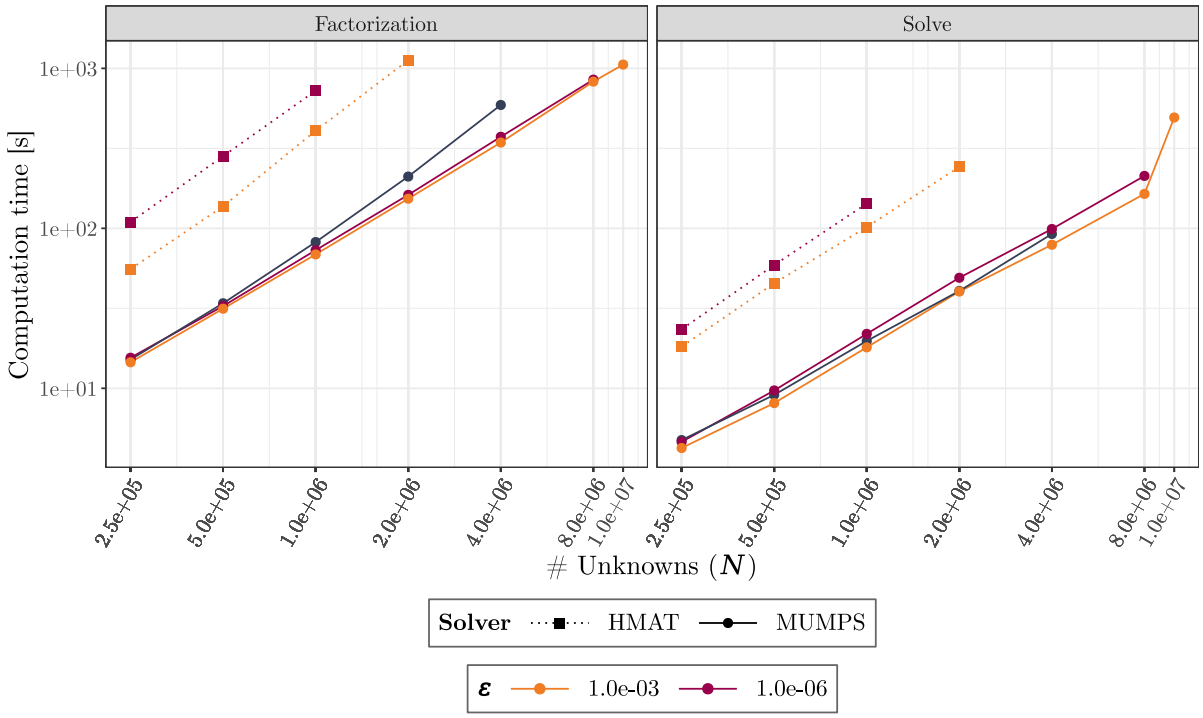


Figure 8.5: Computation time of MUMPS and HMAT on FEM systems run in parallel using 1 MPI process and 24 threads on a single `miriel` node for different low-rank compression thresholds ϵ and no ϵ for MUMPS when compression is disabled.

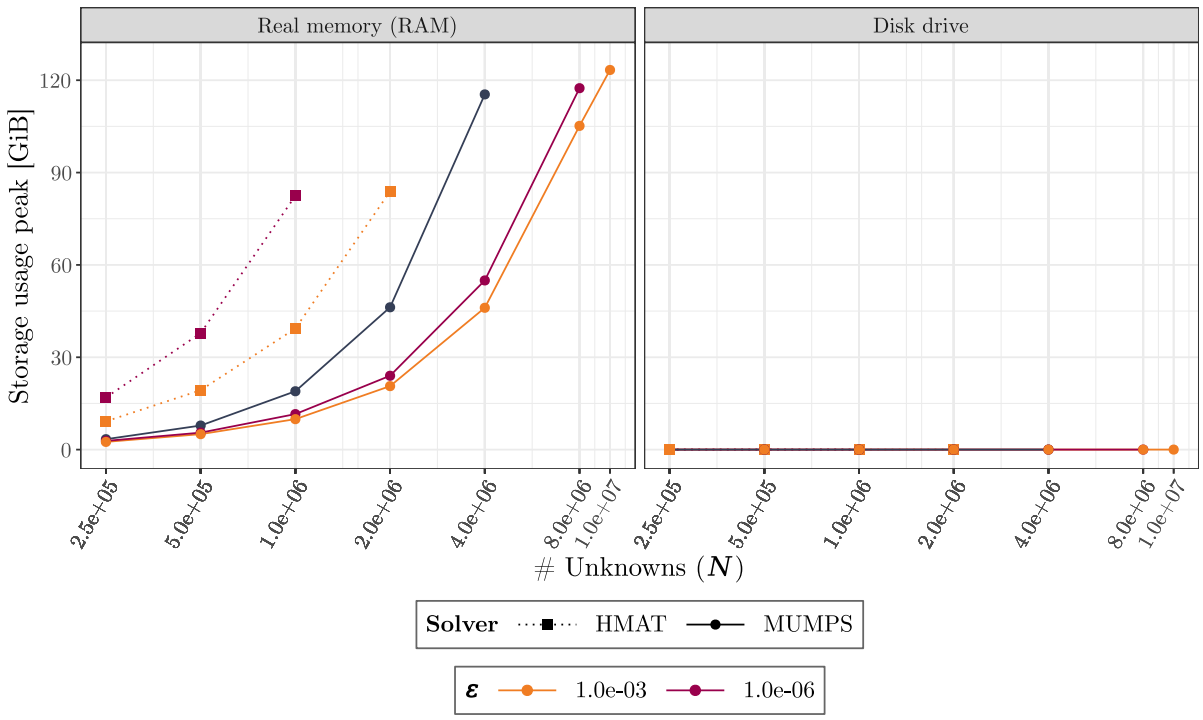


Figure 8.6: RAM and disk space usage peaks of MUMPS and HMAT on FEM systems run in parallel using 1 MPI process and 24 threads on a single `miriel` node for different low-rank compression thresholds ϵ and no ϵ for MUMPS when compression is disabled.

schemes for processing sparse matrices have been derived [71] but have not been integrated in the industrial framework and are thus not discussed in the present study.

In Figure 8.7, we compare HMAT with and without the use of ND on smaller sparse FEM systems using non-symmetric matrices and having from 25,000 up to 250,000 unknowns. We can not observe a significant difference in computation time among runs with different low-rank compression thresholds when using ND. Nevertheless, it appears to provide a clearly better performance in terms of computation time for cases with up to 100,000 unknowns. Although, starting from 200,000 unknowns, this trend seems to cease.

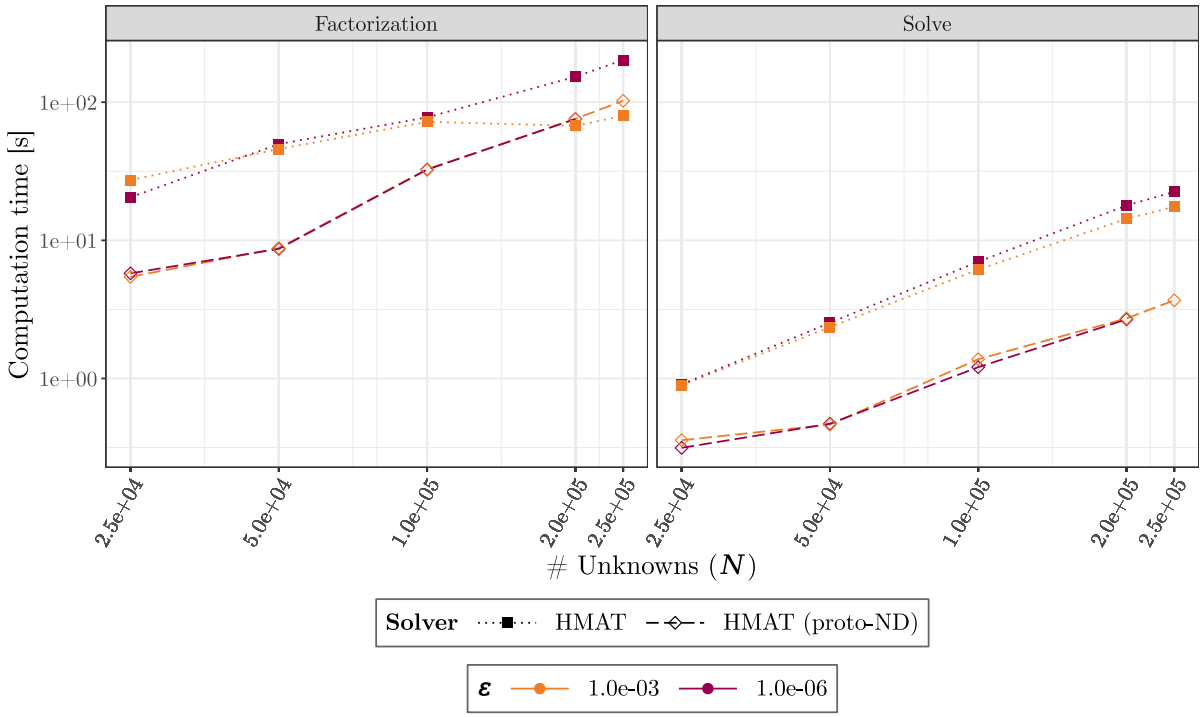


Figure 8.7: Computation time of HMAT on non-symmetric FEM systems with and without the Nested Dissection (ND) enabled, run in parallel using 1 MPI process and 24 threads on a single `miriel` node for different low-rank compression thresholds ϵ.

Notice that, HMAT was run with ND only to produce the results featured in Figure 8.7. In all the other cases, ND was disabled.

According to Figure 8.8, HMAT yields a better solution accuracy on sparse FEM than on dense BEM systems (see Section A.2). Although the relative error of the solution approximations computed by MUMPS when using compression is smaller than the corresponding low-rank compression thresholds, it is not as low as in the case of HMAT. When the compression is disabled, we naturally observe the relative error to approach the machine precision u (see Section 1.5.2.3). Eventually, the results also validates the stability of both solvers for given problem sizes.

To compare the scalability of MUMPS and HMAT on FEM systems while putting in action various parallel configurations (see Table 8.1), we consider systems having 2,000,000 unknowns for both MUMPS and HMAT. In the case of MUMPS, we consider systems with 4,000,000 unknowns as well. The more MPI processes are involved in the computation, the more memory consumption increases (see Figure 8.9). This also explains that, the case with 4,000,000 unknowns relying exclusively on MPI parallelism causes a memory overflow using 16 cores and more. Finally, the low-rank compression threshold ϵ (see Section 1.5.2.3) has been set to 10^{-3} for all the runs.

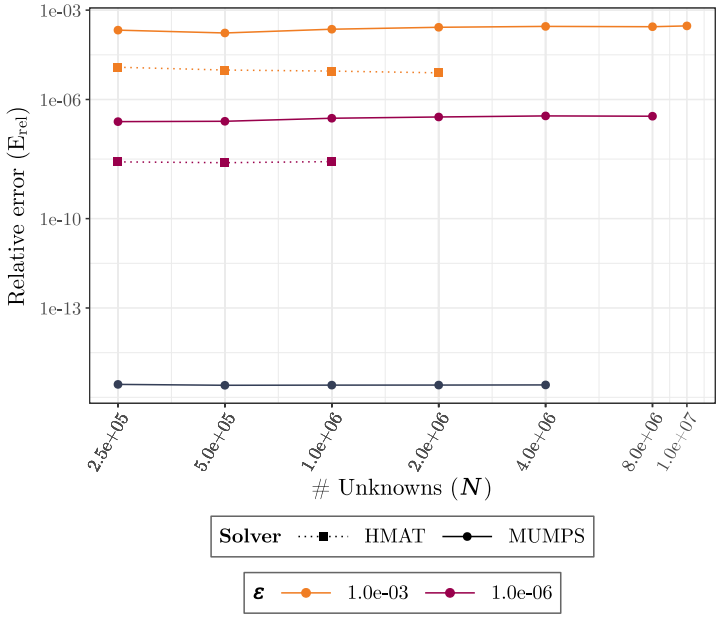


Figure 8.8: Relative solution error E_{rel} of MUMPS and HMAT on FEM systems run in parallel run using 1 MPI process and 24 threads on single `miriel` node for different low-rank compression thresholds ϵ and no ϵ set for MUMPS when compression is disabled.

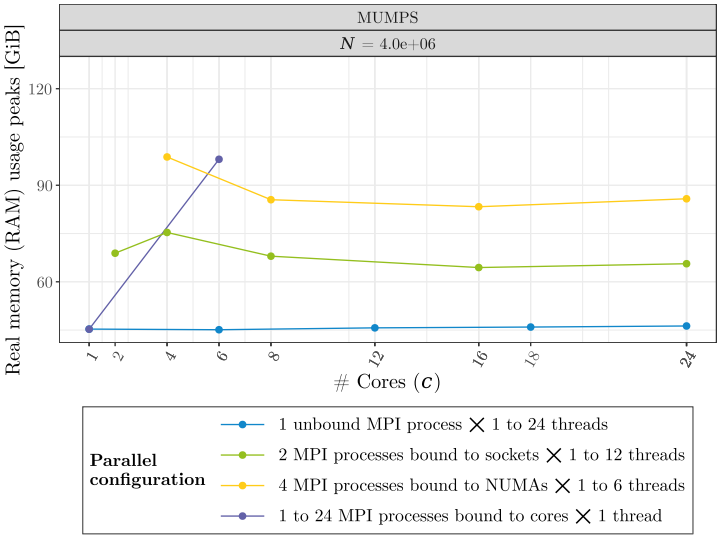


Figure 8.9: RAM usage peaks of MUMPS on FEM systems run in 4 different kinds of parallel configurations using only 1 MPI process without binding times 1 to 24 threads or 2 MPI processes bound to sockets times 1 to 12 threads or 4 MPI processes bound to NUMA subnodes times 1 to 6 threads or 1 to 24 MPI processes bound to cores times 1 thread on single `miriel` node with the low-rank compression threshold ϵ set to 10^{-3}.

In Figure 8.10 we show the computation times and in Figure 8.11 we show the parallel efficiency of factorization and solve phases of MUMPS and HMAT for different count of processor cores made available for the computation. Considered parallel configurations are listed in Table 8.1. MUMPS scales well for all of the assessed parallel configurations with no significant difference in computation time of the most demanding factorization phase.

Regarding the parallel efficiency, the best performing MUMPS configuration for both problem sizes and the factorization phase seems to be the one using 4 MPI processes times 1 to 6 threads yielding nearly 20% parallel efficiency on 24 cores. In the case of the solve phase, exclusively MPI parallelism yields the best results with 20% parallel efficiency on 24 cores considering a problem with 2,000,000 unknowns and nearly 30% on 12 cores considering a problem with 4,000,000 unknowns. However, this study was conducted using an older version of MUMPS, i.e. 5.2.1, and numerous optimizations have been implemented in the more recent versions of the solver [56].

HMAT scales well only if we rely exclusively on local thread parallelism. The parallel efficiency of factorization reaches approximately 40% on 24 cores. The MPI parallelization of HMAT is not optimized for computations on a single node. Therefore, we dropped the scalability tests of HMAT for the parallel configurations involving more than one MPI process as the results would not be representative. A preliminary investigation revealed that the significant decrease in performance, when MPI processes are involved, comes from a poorly balanced workload. In HMAT, it is difficult to determine an efficient static mapping for MPI processes. Yet in the explored model we rely on static mapping which explains the execution trace provided by StarPU and presented in Figure 8.12. The more MPI processes are involved in the computation, the more time threads spend in inactivity (sleeping).

A.2 BEM systems

In this section, we evaluate the performance of the direct solvers SPIDO and HMAT (see sections 2.6.4.2 and 2.6.4.2) on dense linear systems resulting from BEM discretization and counting from 25,000 up to 1,000,000 unknowns. Here, we thus rely on the *wide pipe* test case variant in Figure 8.4b.

According to Figure 8.13, the computation times of SPIDO are significantly higher than those measured in the case of HMAT. For example, the factorization time of SPIDO on a system with 100,000 unknowns is comparable to the factorization time of HMAT on a system having 1,000,000 unknowns. Also, without out-of-core and lacking any kind of data compression, SPIDO quickly approaches the 128 GiB memory limit of the `miriel` nodes leaving it unable to process linear systems with 200,000 or more unknowns (see Figure 8.14).

These results show the advantages the hierarchical matrix structure and the low-rank compression capabilities (see Section 1.5.1.1) implemented in HMAT (see Section 2.6.4.2) for the solution of dense linear systems. We have observed both better computation times as well as lower memory footprint allowing the HMAT solver to process systems with up to 1,000,000 of unknowns when the low-rank compression threshold ϵ is set to 10^{-3}. Nevertheless, after tightening up the threshold to 10^{-6}, the factorization time increases more rapidly and the memory limit is reached sooner too. Ultimately, this makes the runs on systems counting more than 400,000 unknowns (see figures 8.13 and 8.14) fail due to insufficient memory. The difference is naturally less noticeable for the solve phase having a considerably lower complexity compared to factorization.

Out-of-core being disabled, HMAT shows no disk space consumption (see Figure 8.14). One would expect the same for SPIDO, although the version of the solver used for the experiment stores an auxiliary matrix on disk regardless the out-of-core setting. This happens only when

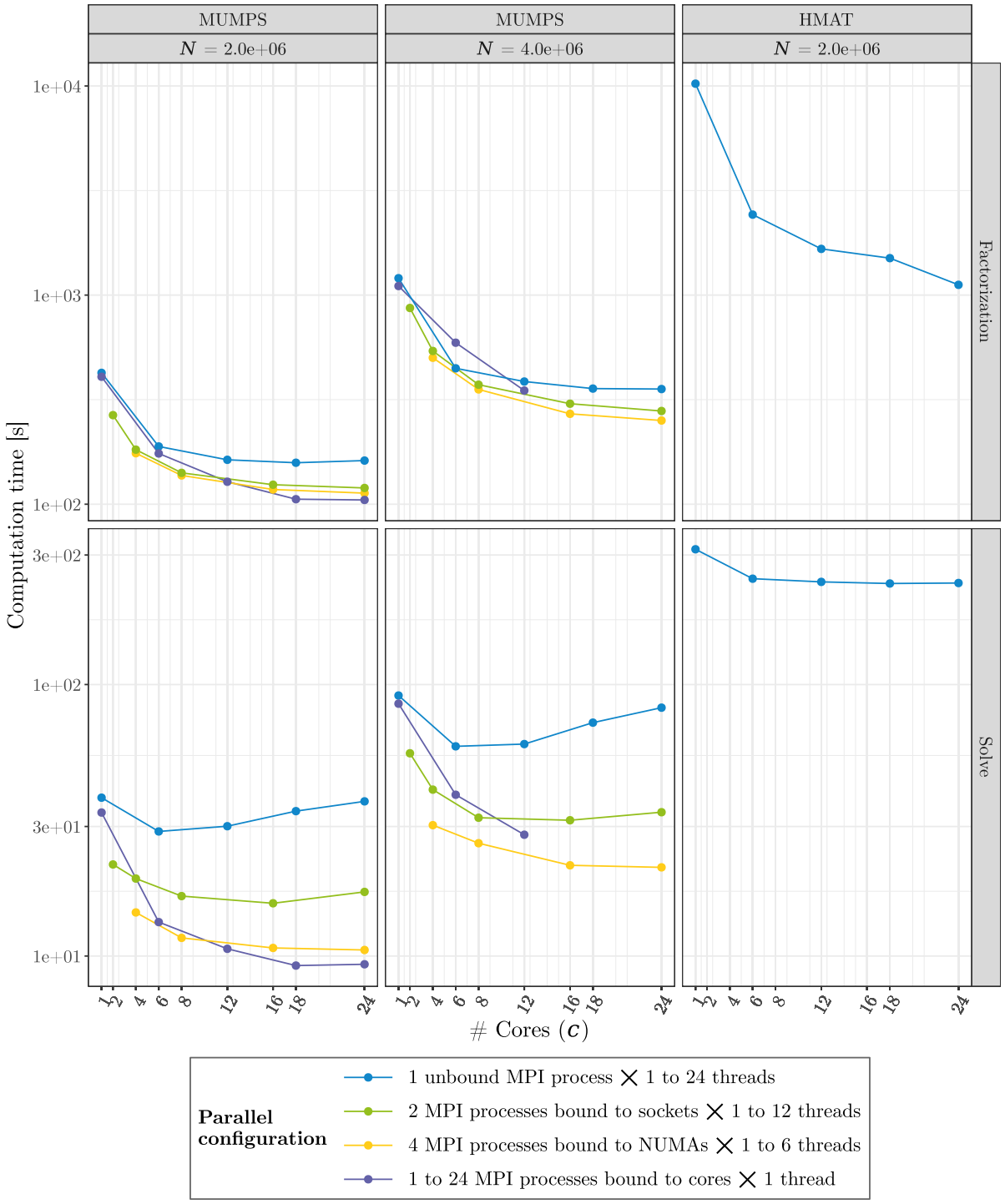


Figure 8.10: Scalability of MUMPS and HMAT on FEM systems run in 4 different kinds of parallel configurations using only 1 MPI process without binding times 1 to 24 threads or 2 MPI processes bound to sockets times 1 to 12 threads or 4 MPI processes bound to NUMA subnodes times 1 to 6 threads or 1 to 24 MPI processes bound to cores times 1 thread on single `miriel` node with the low-rank compression threshold ϵ set to 10^{-3}.

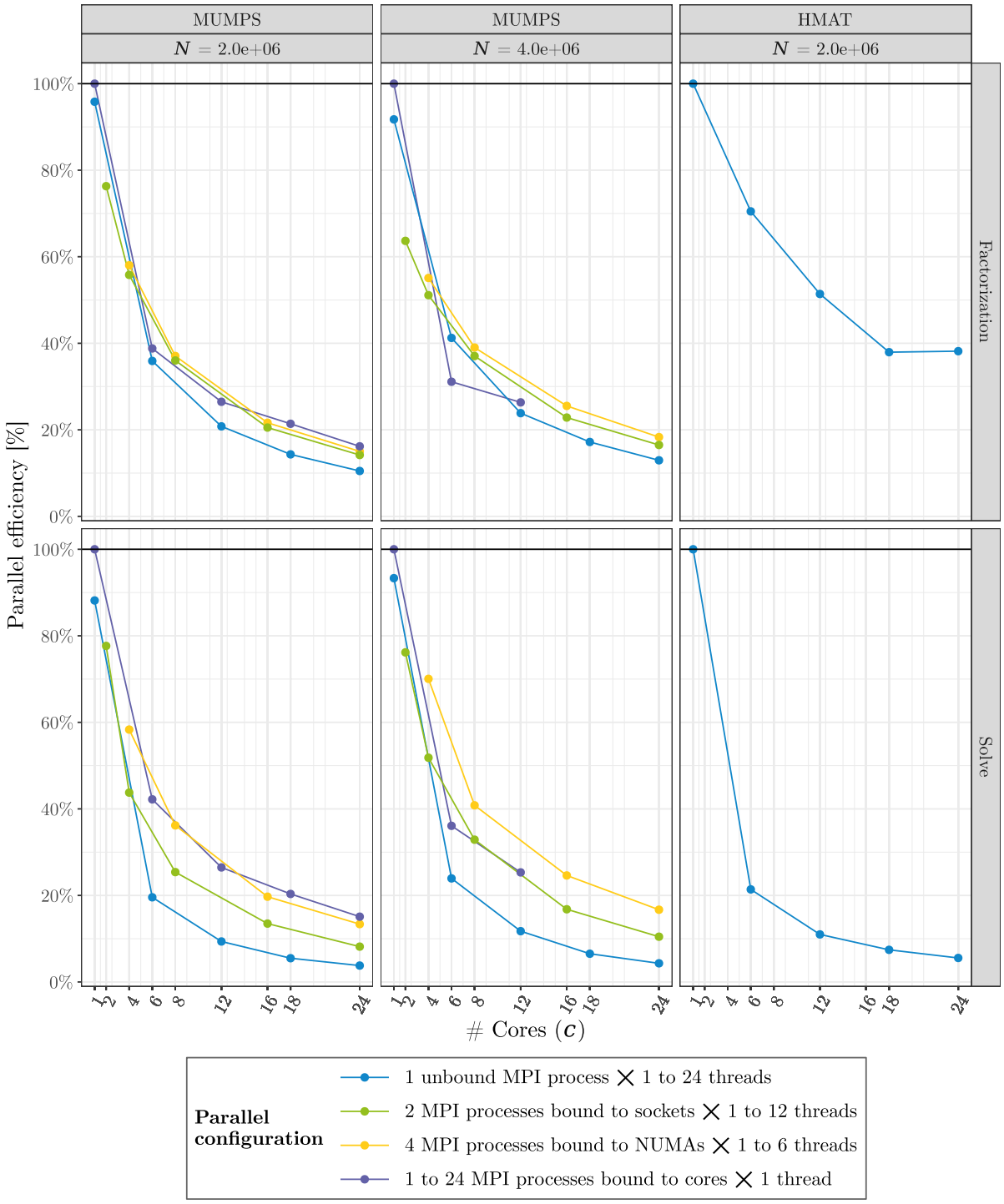


Figure 8.11: Parallel efficiency of MUMPS and HMAT on FEM systems run in 4 different kinds of parallel configurations using only 1 MPI process without binding times 1 to 24 threads or 2 MPI processes bound to sockets times 1 to 12 threads or 4 MPI processes bound to NUMA subnodes times 1 to 6 threads or 1 to 24 MPI processes bound to cores times 1 thread on a single `miriel` node with the low-rank compression threshold ϵ set to 10^{-3}.

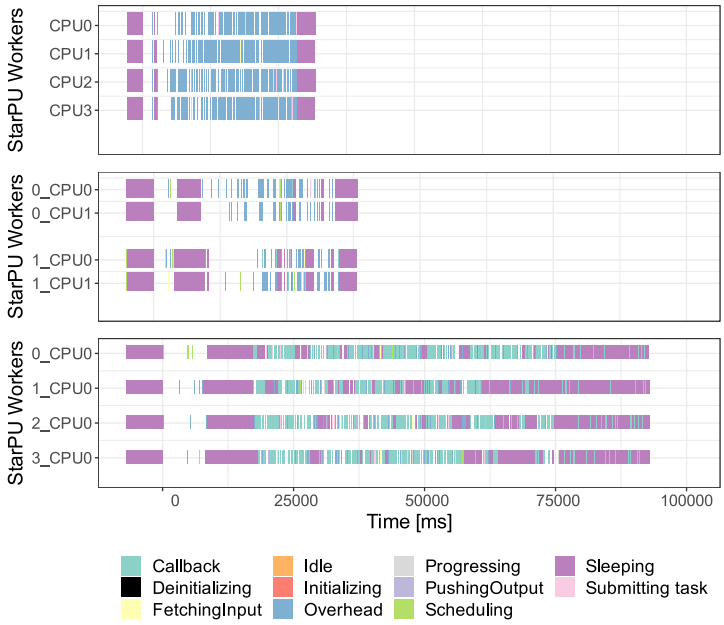


Figure 8.12: Graphical visualization of the execution traces provided by the StarPU runtime corresponding to the execution of HMAT on a FEM system having 25,000 unknowns using (from the top to the bottom of the figure) 1 MPI process times 4 StarPU workers, 2 MPI processes times 2 threads, 4 MPI processes with 1 thread per process on a single `miriel` node respectively. Blank spaces represent the time spent in computation. The dark violet color indicates sleeping.

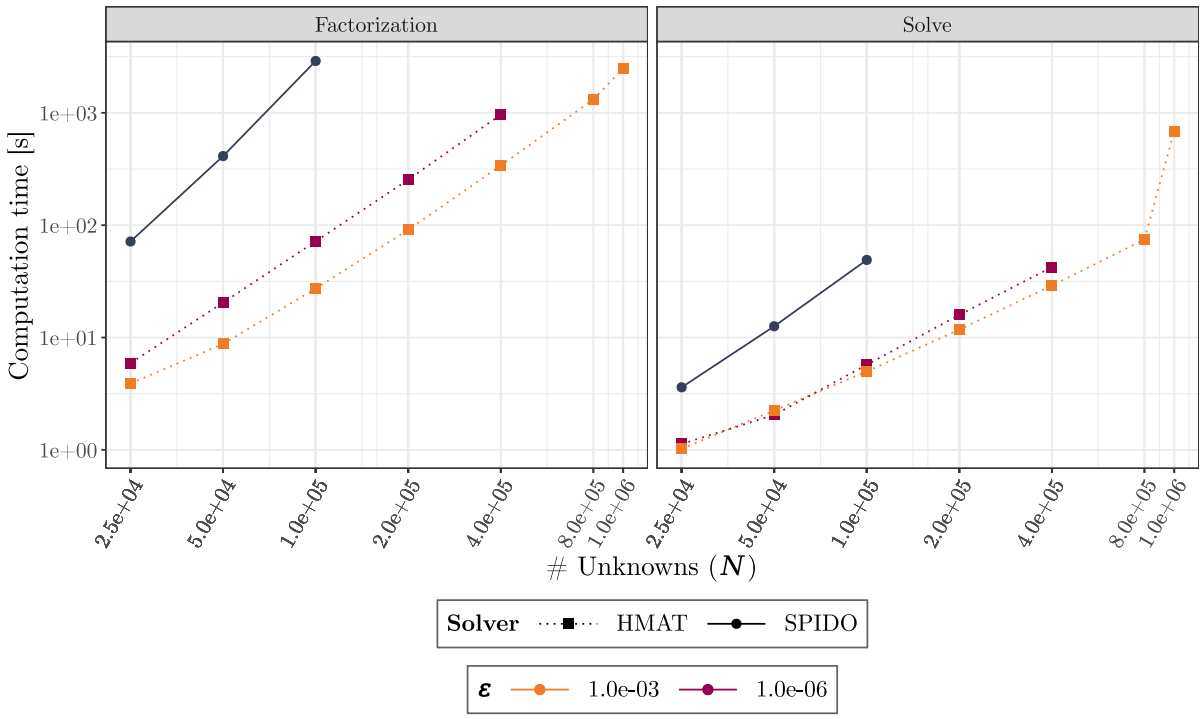


Figure 8.13: Computation times of SPIDO and HMAT on BEM systems run in parallel using 1 MPI process and 24 threads on a single `miriel` node for different low-rank compression thresholds ϵ in the case of HMAT and no ϵ set for SPIDO.

the LDL^T-factorization is used and it has been corrected by commit `e057b1d6` in the MPF package by the time of finishing this study.

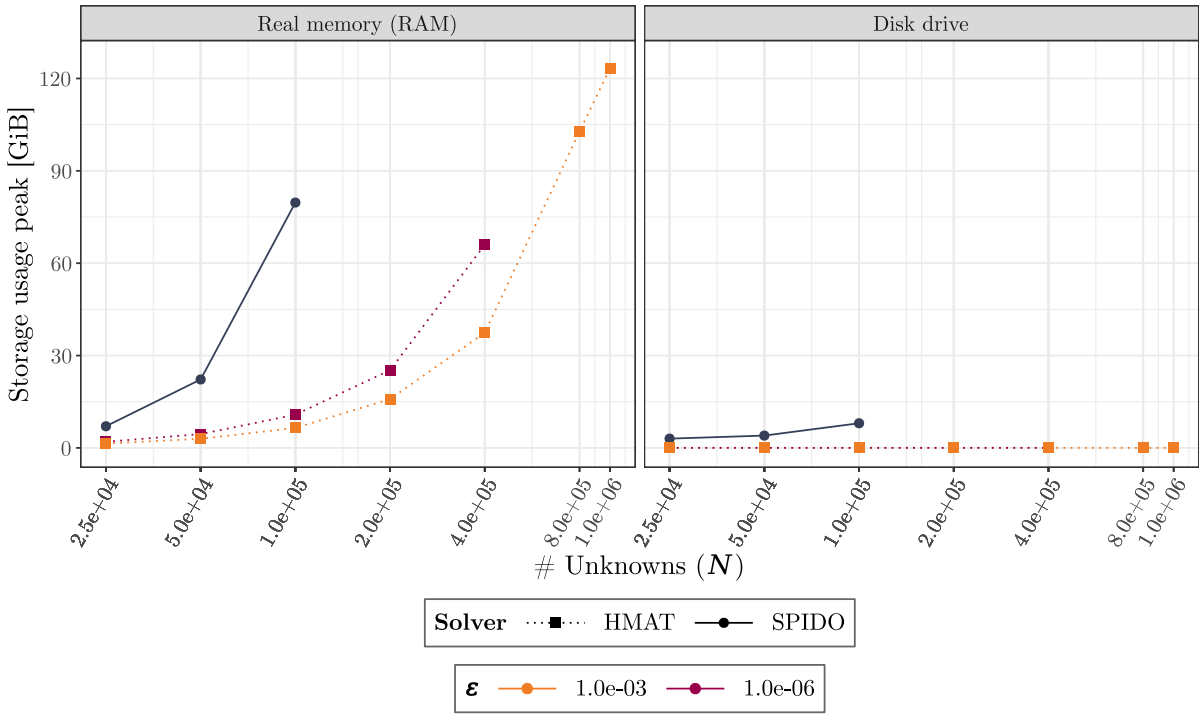


Figure 8.14: RAM and disk usage peaks of SPIDO and HMAT on BEM systems run in parallel using 1 MPI process and 24 threads on a single `miriel` node for different low-rank compression thresholds ϵ in the case of HMAT and no ϵ set for SPIDO.

Figure 8.15 validates the stability of the solvers for given problem sizes. The relative error of the solution approximations computed by HMAT exceeds the given low-rank compression thresholds ϵ (see Section 1.5.2). However, it is merely a small deviation and we consider it as non-significant in this case. Regarding SPIDO using no data compression, the relative error is naturally approaching the machine precision u (see Section 1.5.2.3).

When it comes to evaluate the scalability and the parallel efficiency of SPIDO and HMAT, we consider dense systems with 100,000 unknowns processed using multiple parallel configurations (see Table 8.1). In the case of HMAT, we consider also systems with 1,000,000 unknowns. Note that the low-rank compression threshold ϵ for HMAT has been set to 10^{-3}.

In Figure 8.16 we show the computation times and in Figure 8.17 we show the parallel efficiency of factorization and solve phases of SPIDO and HMAT for different counts of processor cores. SPIDO scales well for all of the assessed parallel configurations. Unlike in case of the solve phase, there is no significant difference in computation times of the factorization phase. In terms of parallel efficiency, the two best performing configurations regarding the factorization phase are the one using only MPI parallelism and the one combining 4 MPI processes times 1 to 6 threads yielding almost 90% parallel efficiency on 24 cores. The solve phase appears to scale the best when relying exclusively on MPI parallelism yielding nearly 80% parallel efficiency on 24 cores.

Unlike SPIDO, HMAT scales well when relying exclusively on thread parallelism (and a single MPI process) for both factorization and solve phases as it was the case for FEM systems (see Section A.1). The parallel efficiency of factorization reaches approximately 60% on 24 cores considering a system with 100,000 unknowns and approximately 50% with 1,000,000 unknowns. The solve phase represents only a small part in the overall computation time which explains its

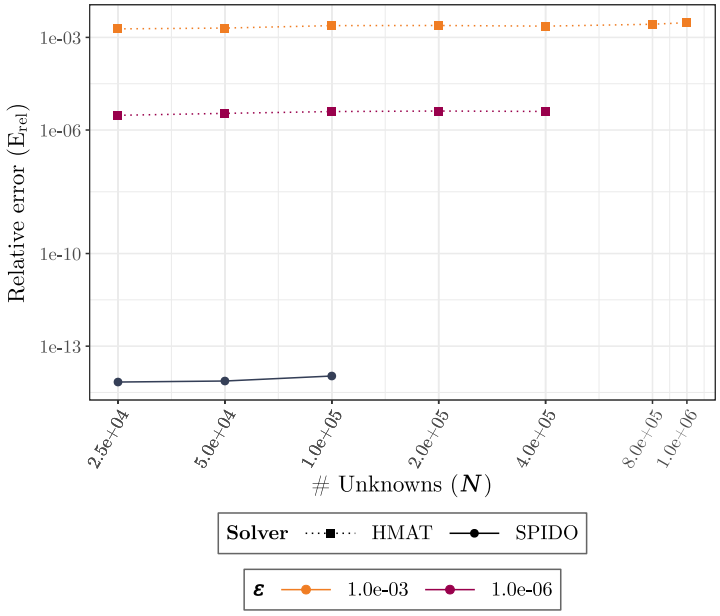


Figure 8.15: Relative error E_{rel} of SPIDO and HMAT on BEM systems run in parallel using 1 MPI process and 24 threads on a single `miriel` node for different low-rank compression thresholds ϵ in the case of HMAT and no ϵ set for SPIDO.

low efficiency in a multi-threaded environment. We confirm the poor performance of HMAT when MPI parallelism is involved on a single node for sparse systems too. According to the execution trace provided by StarPU and presented in Figure 8.18. The more MPI processes are involved in the computation, the more time threads spend in inactivity (sleeping).

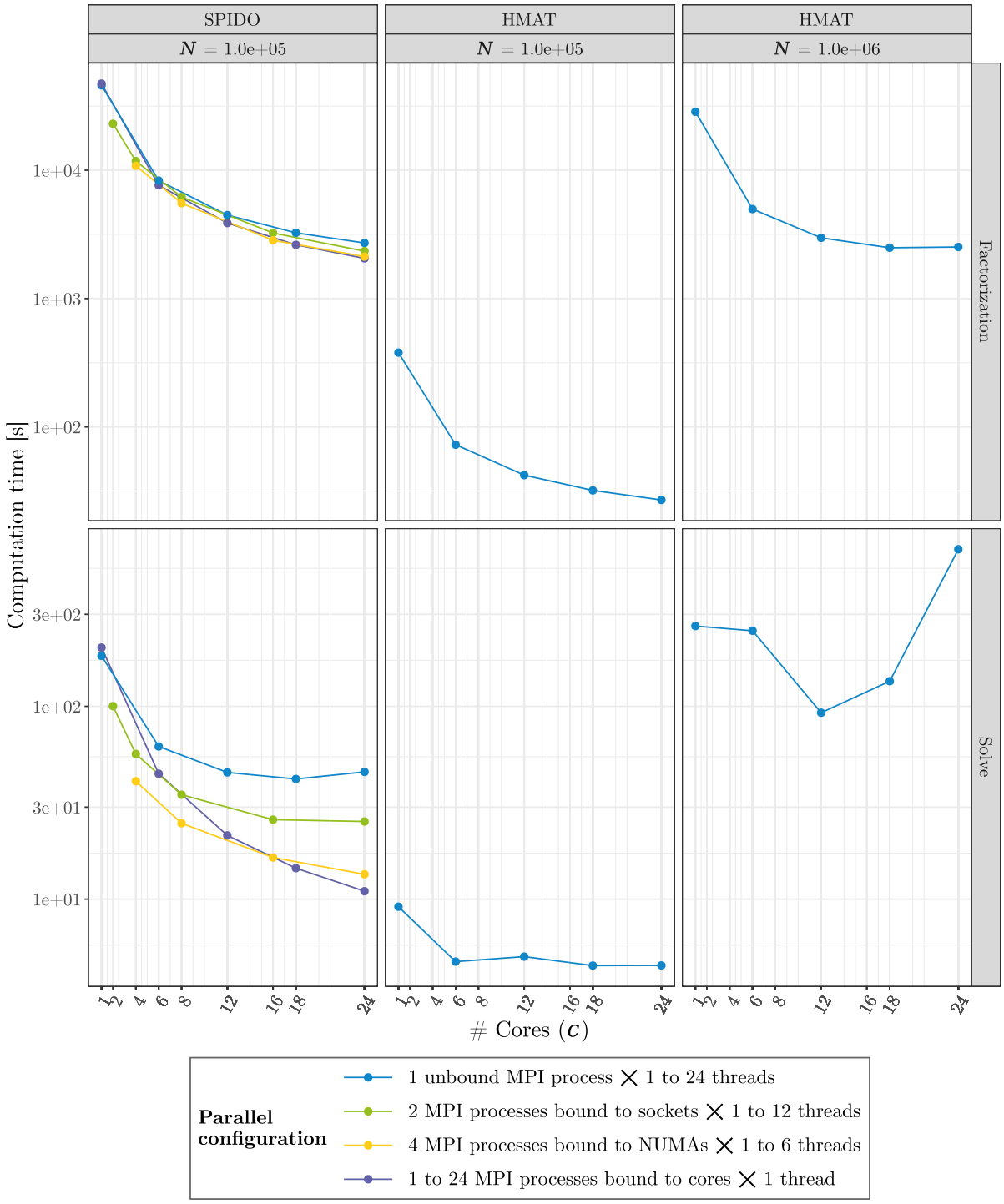


Figure 8.16: Scalability of SPIDO and HMAT on BEM systems run in 4 different kinds of parallel configurations using only 1 MPI process without binding times 1 to 24 threads or 2 MPI processes bound to sockets times 1 to 12 threads or 4 MPI processes bound to NUMA subnodes times 1 to 6 threads or 1 to 24 MPI processes bound to cores times 1 thread on single `miriel` node with the low-rank compression threshold ϵ set to 10^{-3} and no ϵ set for SPIDO.

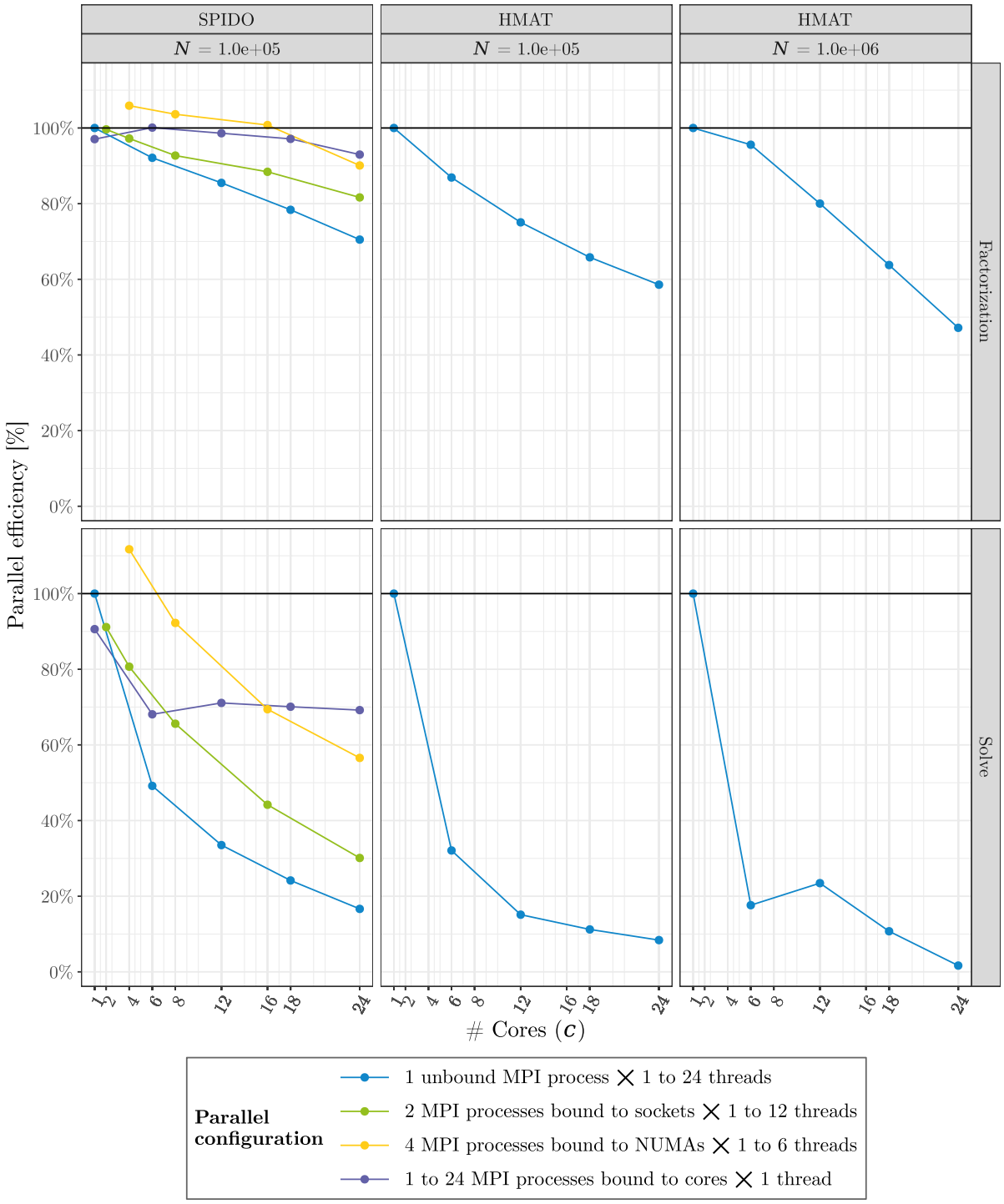


Figure 8.17: Parallel efficiency of SPIDO and HMAT on BEM systems run in 4 different kinds of parallel configurations using only 1 MPI process without binding times 1 to 24 threads or 2 MPI processes bound to sockets times 1 to 12 threads or 4 MPI processes bound to NUMA subnodes times 1 to 6 threads or 1 to 24 MPI processes bound to cores times 1 thread on a single `miriel` node with the low-rank compression threshold ϵ set to 10^{-3} and no ϵ set for SPIDO.

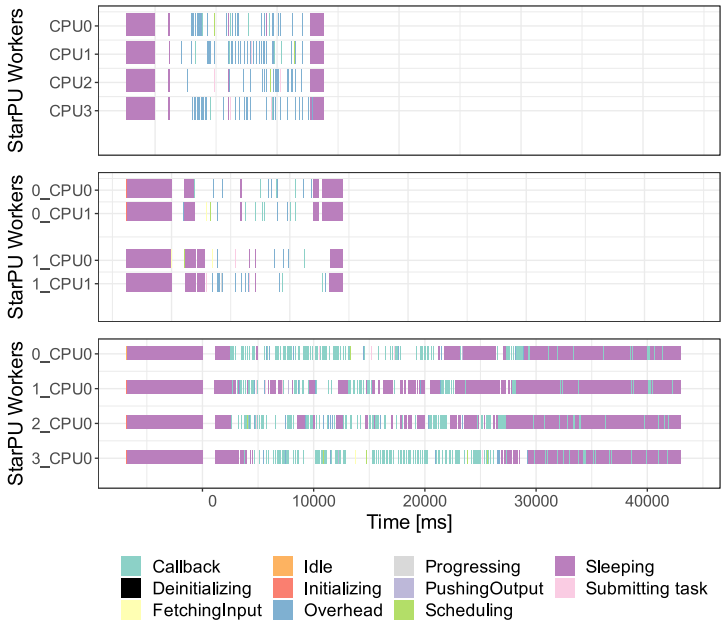


Figure 8.18: Graphical visualization of the execution traces provided by the StarPU runtime corresponding to the execution of HMAT on a BEM system having 25,000 unknowns using (from the top to the bottom of the figure) 1 MPI process times 4 StarPU workers, 2 MPI processes times 2 threads, 4 MPI processes with 1 thread per per process on a single `miriel` node respectively. Blank spaces represent the time spent in computation. The dark violet color indicates sleeping.

B Example study manuscript

This section represents the complete listing of the Org source of the example research study manuscript from Section 8.3.1.

The Org document begins by a header defining various properties, such as title and authors, simple replacement macros [18] similar to custom LaTeX commands. We then have the statements specific to LaTeX export, i.e. lines beginning with `#+LaTeX_CLASS` and `#+LaTeX_HEADER`.

```
#+TITLE: Solvers for coupled sparse/dense FEM/BEM linear systems
#+AUTHOR: Marek Felšöci
#+DATE: {{{time(%B %d\, %Y)}}}
#+OPTIONS: toc:nil
#+MACRO: test_FEMBEM ~test_FEMBEM~
#+MACRO: hmat HMAT
#+MACRO: hmat-oss HMAT-OSS
#+MACRO: chameleon Chameleon
#+MACRO: epsilon $10^{-3}$
#+MACRO: spipe /wide pipe/
#+MACRO: hmatrix \(\mathcal{H}\)-Matrix
#+MACRO: ggplot ~ggplot2~
#+MACRO: svglite ~svglite~
#+LaTeX_CLASS: article
#+LaTeX_CLASS_OPTIONS: [a4paper, 11pt, twoside, table]
#+LaTeX_HEADER: \usepackage[inkscapelatex = false]{svg}
#+LaTeX_HEADER: \usepackage[margin = 25mm, head = 10mm]{geometry}
#+LaTeX_HEADER: \setlength{\parindent}{0pt}
#+LaTeX_HEADER: \setlength{\parskip}{2mm}
#+LaTeX_HEADER: \PassOptionsToPackage{hyphens}{url}\usepackage{hyperref, float}
```

Follows the Org-formatted text corresponding to the section Introduction (see Section 8.3.1.1 and to the beginning of the section Experiments (see Section 8.3.1.2). We can notice macro expansions in between triple curly brackets, i.e. HMAT for the `hmat` macro.

```
* Introduction
:PROPERTIES:
:CUSTOM_ID: example-introduction
:END:

This is an example experimental study relying on the {{{test_FEMBEM}}} solver
test suite. Here, we are especially interested in solving coupled sparse/dense
FEM/BEM linear systems arising in the domain of aeroacoustics. The idea is to
evaluate the solvers available in the open-source version of {{{test_FEMBEM}}}
for the solution of this kind of linear systems.

* Experiments
:PROPERTIES:
:CUSTOM_ID: example-study
:END:

The open-source version of {{{test_FEMBEM}}} cite:testFEMBEM does not implement
couplings of sparse and dense direct solvers which is the preferred method for
solving sparse/dense FEM/BEM systems. Therefore, we process the systems as dense
using either the {{{hmat-oss}}} or the {{{chameleon}}} direct solver.
{{{hmat-oss}}} cite:hmat-oss is an open-source and sequential version of the
compressed hierarchical {{{hmatrix}}} dense direct solver {{{hmat}}} cite:Lize14
developed at Airbus. {{{chameleon}}} cite:chameleon is a fully open-source dense
direct solver without compression.

As of the test case, we consider a simplified {{{spipe}}} which is still close
enough to real-life models (see Figure [[figure:example-pipe]]). Note that all
the benchmarks were conducted on a single machine equipped with an octa-core
Intel(R) Xeon(R) W3520 running at 2.661 GHz and 8 GiB of RAM.

#+CAPTION: A {{{spipe}}} mesh counting 20,000 vertices.
#+NAME: figure:example-pipe
#+ATTR_LaTeX: :width .3\columnwidth :placement [H]
[[./figures/pipe-2.png]]

At first, we want to know to which extent can data compression improve the
computation time. For this, we compare sequential executions of both
{{{hmat-oss}}}, the compressed solver, and {{{chameleon}}}, the non-compressed
solver, on coupled FEM/BEM systems of different sizes (see Figure
[[figure:hmat-chameleon]]). The results clearly show the advantage of using data
compression, especially with increasing size of the target linear system.
```

The fragment below features the R code block responsible for generating Figure 8.2. The header of the code block specifies that the output is a graphics file and gives also its dimensions and resulting file name. When the code block is evaluated, the result is dynamically inserted into the surrounding text where the `#+RESULTS:` keyword followed by the `#+NAME` of the source code block is located.

```
#+HEADER: :results output graphics file :exports results :width 5 :height 5
#+HEADER: :file ./figures/hmat-chameleon.svg :eval yes
#+NAME: code:hmat-chameleon
#+BEGIN_SRC R
library(ggplot2)

data <- read.csv(file = "benchmarks/results/results.csv", header = TRUE)
```

```
ggplot(
  data = subset(x = data, threads == 1),
  mapping = aes(x = nbpts, y = tps_facto + tps_solve, color = solver)
) +
geom_line() +
geom_point(size = 2.5) +
scale_x_continuous(name = "# Unknowns (\U1D441)") +
scale_y_continuous(name = "Computation time [s]") +
labs(color = "Solver") +
scale_color_manual(
  values = c("hmat" = "#F07E26", "chameleon" = "#9B004F"),
  labels = c(
    "hmat" = "HMAT-OSS (compressed)", "chameleon" = "Chameleon (non compressed)"
  )
) +
theme_bw() +
theme(
  legend.background = element_rect(color = "gray40", size = 0.5),
  legend.box = "vertical",
  legend.position = "bottom",
  legend.text = element_text(size = 14),
  legend.title = element_text(size = 14, face = "bold"),
  text = element_text(size = 16)
) +
guides(
  color = guide_legend(
    order = 1,
    nrow = 2,
    byrow = TRUE
  )
)
#+END_SRC
```

In this case, it is in a figure environment directly after the code block. Here, the output graphic file is represented by a local file link. Note that the Emacs text editor can be set up to display the corresponding image itself instead of the link to it.

```
#+CAPTION: Computation times of sequential runs of {{{hmat-oss}}} and
#+CAPTION: {{{chameleon}}} on coupled sparse/dense FEM/BEM linear systems of
#+CAPTION: varying size.
#+NAME: figure:hmat-chameleon
#+ATTR_LaTeX: :width .6\columnwidth
#+RESULTS: code:hmat-chameleon
[[./figures/hmat-chameleon.svg]]
```

The next paragraph of text ends with a dynamically computed percentage value. The latter results from the evaluation of the R source code block below the paragraph.

```
To study the impact of parallel execution on the time to solution, we limit
ourselves to the {{{chameleon}}} solver as {{{hmat-oss}}} is sequential-only. In
Figure [[figure:chameleon]], we compare the computation times of {{{chameleon}}}
on coupled FEM/BEM systems of different sizes using either one or four threads.
According to the results, we can observe a significant decrease in computation
time in case of parallel executions. Moreover the parallel efficiency of the run
on the largest linear system considered (8000 unknowns) is approximately
call_efficiency(results="benchmarks/results/results.csv", size=8000, nt=4)%.

#+NAME: efficiency
#+HEADER: :var results="NA" size=-1 nt=-1 :exports none :results raw
```

```
#+BEGIN_SRC R
data <- read.csv(file = results, header = TRUE)
data <- subset(data, solver == "chameleon" & nbpts == size)
data$time <- data$tps_facto + data$tps_solve
Ts <- data[data$threads == 1, "time"]
Tp <- data[data$threads == nt, "time"]
E <- (Ts / (Tp * nt)) * 100
print(as.integer(E))
#+END_SRC
```

The following fragment corresponds to the R source code block for generating Figure 8.3.

```
#+HEADER: :results output graphics file :exports results :width 5 :height 5
#+HEADER: :file ./figures/chameleon.svg :eval yes
#+NAME: code:chameleon
#+BEGIN_SRC R
library(ggplot2)

data <- read.csv(file = "benchmarks/results/results.csv", header = TRUE)

ggplot(
  data = subset(x = data, solver == "chameleon"),
  mapping = aes(
    x = nbpts,
    y = tps_facto + tps_solve,
    color = solver,
    linetype = as.character(x = threads),
    shapes = as.character(x = threads)
  )
) +
geom_line() +
geom_point(size = 2.5) +
scale_x_continuous(name = "# Unknowns (\U1D441)") +
scale_y_continuous(name = "Computation time [s]") +
labs(color = "Solver", linetype = "# Threads", shape = "# Threads") +
scale_color_manual(
  values = c("chameleon" = "#9B004F"),
  labels = c("chameleon" = "Chameleon")
) +
scale_linetype_manual(
  values = c("1" = "solid", "4" = "dotted"),
) +
scale_shape_manual(
  values = c("1" = 15, "4" = 16),
) +
theme_bw() +
theme(
  legend.background = element_rect(color = "gray40", size = 0.5),
  legend.box = "vertical",
  legend.position = "bottom",
  legend.text = element_text(size = 14),
  legend.title = element_text(size = 14, face = "bold"),
  text = element_text(size = 16)
) +
guides(
  color = guide_legend(
    order = 1,
    nrow = 1,
    byrow = TRUE
  )
)
```

```
#+END_SRC
```

After Figure 8.3, we continue with the section Conclusion (see Section 8.3.1.3).

```
#+CAPTION: Computation times of sequential and parallel runs of {{{chameleon}}}
#+CAPTION: on coupled sparse/dense FEM/BEM linear systems of varying size.
#+NAME: figure:chameleon
#+ATTR_LaTeX: :width .6\columnwidth
#+RESULTS: code:chameleon
[[./figures/chameleon.svg]]

* Conclusion
:PROPERTIES:
:CUSTOM_ID: example-conclusion
:END:

We evaluated the performance of the solvers available in the {{{test_FEMBEM}}}
test suite on coupled sparse/dense FEM/BEM linear systems.  The solvers
considered were {{{hmat-oss}}}, a sequential compressed dense direct solver and
{{{chameleon}}}, a multi-threaded non-compressed dense direct solver.

The comparison of sequential runs of {{{hmat-oss}}} and {{{chameleon}}} showed
an important positive impact of data compression on the time to solution. In
addition, the comparison of sequential and parallel runs of {{{chameleon}}} as
well as the computed parallel efficiency showed a considerable speed-up of the
parallel execution.
```

We end with the section Notes on reproducibility (see Section 8.3.1.4) and the inclusion of bibliography.

```
* Notes on reproducibility
:PROPERTIES:
:CUSTOM_ID: example-reproducibility
:END:

With the aim of keeping the experimental environment of the study reproducible,
we manage the associated software framework with the GNU Guix transactional
package manager cite:guix. Moreover, relying on the principles of literate
programming cite:Knuth84, we provide a full documentation on the construction
process of the experimental environment, the execution of benchmarks, the
collection and the visualization of results as well as on producing the final
manuscripts in a dedicated technical report associated with this study
cite:RT-EXAMPLE. A public companion contains all of the source code, guidelines
and other material required for reproducing the study:
[[https://gitlab.inria.fr/thesis-mfelsoci/dissertation/example-fembem]],
archived on [[https://archive.softwareheritage.org/]] under the identifier
@@latex:\\@@ =swh:1:snp:d8cdf44424c392dc25564870cfacb9fec5872554=.

bibliography:references.bib
bibliographystyle:siam
```

Bibliography

[1] *Chameleon, a dense linear algebra software for heterogeneous architectures* . https://gitlab.inria.fr/solverstack/chameleon.

[2] *Docker: Accelerated, Containerized Application Development*. https://www.docker.com/.

[3] *du Linux manual page*. https://man7.org/linux/man-pages/man1/du.1.html.

[4] *Energy Scope website*. https://sed-bso.gitlabpages.inria.fr/datacenter/energy_scope.html.

[5] *First Airbus A350-900 XWB for Cathay Pacific - take off*.

[6] *GNU Emacs: An extensible, customizable, free/libre text editor — and more*. https://www.gnu.org/software/emacs/.

[7] *GNU Guix Cookbook: Basic setup with manifests*. https://guix.gnu.org/cookbook/en/html_node/Basic-setup-with-manifests.html.

[8] *GNU Guix software distribution and transactional package manager*. https://guix.gnu.org.

[9] *Guix-HPC - Reproducible software deployment for high-performance computing*. https://hpc.guix.info/.

[10] *hmat-oss*. https://github.com/jeromerobert/hmat-oss.

[11] *hmat-oss, a hierarchical matrix C/C++ library including a LU solver*. https://github.com/jeromerobert/hmat-oss.

[12] *Invoking `guix pack` (GNU Guix Reference Manual)*. https://guix.gnu.org/manual/en/html_node/Invoking-guix-pack.html.

[13] *OpenMP: The OpenMP API specification for parallel programming*. https://www.openmp.org/.

[14] *OpenMPI: A high performance message passing library*. https://www.open-mpi.org/.

[15] *Org mode documentation (Evaluating source code)*. https://orgmode.org/manual/Evaluating-Code-Blocks.html.

[16] *Org mode documentation (Exporting)*. https://orgmode.org/manual/Exporting.html.

[17] *Org mode documentation (Extracting source code)*. https://orgmode.org/manual/Extracting-Source-Code.html.

[18] *Org mode documentation (Macro Replacement)*. https://orgmode.org/manual/Macro-Replacement.html.

[19] *PlaFRIM: Plateforme fédérative pour la recherche en informatique et mathématiques*. https://plafrim.fr/.

[20] *qr_mumps, a software package for the solution of sparse, linear systems on multicore computers*. http://buttari.perso.enseeiht.fr/qr_mumps/.

[21] *Software Heritage*. https://www.softwareheritage.org/.

[22] *test_FEMBEM, a simple application for testing dense and sparse solvers with pseudo-FEM or pseudo-BEM matrices*. https://gitlab.inria.fr/solverstack/test_fembem.

[23] *TGCC: Très Grand Centre de calcul du CEA*. https://www-hpc.cea.fr/en/complexe/tgcc.htm.

[24] *Thermal Design Power (TDP) in Intel(R) Processors*. https://www.intel.com/content/www/us/en/support/articles/000055611/processors.html.

[25] S. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra, *Analysis and Design Techniques towards High-Performance and Energy-Efficient Dense Linear Solvers on GPUs*, IEEE Transactions on Parallel and Distributed Systems, 29 (2018), pp. 2700–2712.

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

[27] E. Agullo, A. Buttari, A. Guermouche, J. Herrmann, and A. Jego, *Task-Based Parallel Programming for Scalable Algorithms: application to Matrix Multiplication*, Research Report RR-9461, Inria Bordeaux - Sud-Ouest, Feb. 2022.

[28] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez, *Implementing multifrontal sparse solvers for multicore architectures with Sequential Task Flow runtime systems*, ACM Transactions on Mathematical Software, (2016).

[29] E. Agullo, M. Felšöci, and G. Sylvand, *A comparison of selected solvers for coupled FEM/BEM linear systems arising from discretization of aeroacoustic problems*, Research Report RR-9412, Inria Bordeaux Sud-Ouest, June 2021.

[30] ——, *A comparison of selected solvers for coupled FEM/BEM linear systems arising from discretization of aeroacoustic problems: literate and reproducible environment*, Technical Report RT-0513, Inria Bordeaux Sud-Ouest, June 2021.

[31] E. Agullo, M. Felšöci, and G. Sylvand, *Direct solution of larger coupled sparse/dense linear systems using low-rank compression on single-node multi-core machines in an industrial context* , in 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Los Alamitos, CA, USA, 2022, IEEE Computer Society, pp. 25–35.

[32] M. Akhlaghi, R. Infante-Sainz, B. F. Roukema, M. Khellat, D. Valls-Gabaud, and R. Baena-Galle, *Toward long-term and archivable reproducibility*, Computing in Science & Engineering, 23 (2021), pp. 82–91.

[33] J. I. Aliaga, H. Anzt, M. Castillo, J. C. Fernández, G. León, J. Pérez, and E. S. Quintana-Ortí, *Unveiling the Performance-Energy Trade-off in Iterative Linear System Solvers for Multithreaded Processors*, Concurrency and Computation : Practice and Experience, 27 (2015), p. 885–904.

[34] G. Alleon, *Résolution de grands problèmes d'électromagnétisme sur calculateurs parallèles*, PhD thesis, 2000. Thèse de doctorat dirigée par Petiton, Serge Informatique Paris 6 2000.

[35] P. Amestoy, J.-Y. L'Excellent, and G. Moreau, *On exploiting sparsity of multiple right-hand sides in sparse direct solvers*, SIAM Journal on Scientific Computing, 41 (2019), pp. A269–A291.

[36] P. R. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L'Excellent, and C. Weisbecker, *Improving multifrontal methods by means of block low-rank representations*, SIAM Journal on Scientific Computing, 37 (2015), pp. A1451–A1474.

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

[38] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent, *MUMPS multifrontal massively parallel solver version 2.0*, (1998).

[39] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammerling, A. McKenney, et al., *LAPACK Users' guide*, vol. 9, SIAM, 1999.

[40] H. Anzt, J. Dongarra, G. Flegar, N. J. Higham, and E. S. Quintana-Ortí, *Adaptive precision in block-Jacobi preconditioning for iterative sparse linear system solvers*, Concurrency and Computation: Practice and Experience, 31 (2019), p. e4460. e4460 cpe.4460.

[41] H. Anzt, S. Tomov, and J. Dongarra, *Energy Efficiency and Performance Frontiers for Sparse Computations on GPU Supercomputers*, in Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM '15, New York, NY, USA, 2015, Association for Computing Machinery, p. 1–10.

[42] H. Anzt, S. Tomov, and J. Dongarra, *On the performance and energy efficiency of sparse linear algebra on GPUs*, The International Journal of High Performance Computing Applications, 31 (2017), pp. 375–390.

[43] J. ao Vicente Ferreira Lima, I. Raïs, L. Lefèvre, and T. Gautier, *Performance and energy analysis of OpenMP runtime systems with dense linear algebra algorithms*, The International Journal of High Performance Computing Applications, 33 (2019), pp. 431–443.

[44] C. Augonnet, S. Thibault, and R. Namyst, *StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines*, Rapport de recherche RR-7240, INRIA, Mar. 2010.

[45] T. Banachiewicz, *Principes d'une nouvelle technique de la méthode des moindres carrés; Méthode de résolution numérique des équations linéaires, du calcul des déterminants et des inverses et de réduction des formes quadratiques*, Bull. Inter. Acad. Polon. Sci., Sér. A, (1938), pp. 393–404.

[46] P. K. Banerjee and R. Butterfield, *Boundary element methods in engineering science*, vol. 17, McGraw-Hill London, 1981.

[47] H. Barucq, N. Rouxelin, and S. Tordeux, *Low-order Prandtl-Glauert-Lorentz based Absorbing Boundary Conditions for solving the convected Helmholtz equation with Discontinuous Galerkin methods*, Journal of Computational Physics, 468 (2022).

[48] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*, Society for Industrial and Applied Mathematics, 1997.

[49] S. Brenner and R. Scott, *The mathematical theory of finite element methods*, vol. 15, Springer Science & Business Media, 2007.

[50] U. Brink, M. Kreienmeyer, and E. Stein, *Different Methodologies for Coupled BEM and FEM with Implementation on Parallel Computers*, in Boundary Element Topics, W. L. Wendland, ed., Berlin, Heidelberg, 1997, Springer Berlin Heidelberg, pp. 317–337.

[51] D. Brunner, M. Junge, and L. Gaul, *A comparison of FE–BE coupling schemes for large-scale problems with fluid–structure interaction*, International Journal for Numerical Methods in Engineering, 77 (2009), pp. 664–688.

[52] J. R. Bunch and L. Kaufman, *Some stable methods for calculating inertia and solving symmetric linear systems*, Mathematics of computation, (1977), pp. 163–179.

[53] N. Canouet, *Méthodes de Galerkin Discontinu pour la résolution du système de Maxwell sur des maillages localement raffinés non-conformes*, theses, Ecole des Ponts ParisTech, Dec. 2003.

[54] F. Casenave, *Aéroacoustique, couplage BEM-FEM 3D pour la simulation du bruit rayonné par un turboréacteur*, tech. rep., 2010.

[55] F. Casenave, A. Ern, and G. Sylvand, *Coupled BEM–FEM for the convected Helmholtz equation with non-uniform flow in a bounded domain*, Journal of Computational Physics, 257 (2014), pp. 627–644.

[56] CERFACS, ENS Lyon, INPT(ENSEEIHT)-IRIT, Inria, Mumps Technologies, Université de Bordeaux, *MUltifrontal Massively Parallel Solver (MUMPS) User's guide*, 2020.

[57] J. Charles, W. Sawyer, M. F. Dolz, and S. Catalán, *Evaluating the Performance and Energy Efficiency of the COSMO-ART Model System*, Comput. Sci., 30 (2015), p. 177–186.

[58] A.-L. Cholesky, *Note sur une méthode de résolution des équations normales provenant de l'application de la méthode des moindres carrés à un syteme d'équations linéaires en nombre inférieur à celui des inconnues*, Bull. Géodésique, 2 (1924), pp. 67–77.

[59] S. Cohen-Boulakia, K. Belhajjame, O. Collin, J. Chopard, C. Froidevaux, A. Gaignard, K. Hinsen, P. Larmande, Y. Le Bras, F. Lemoine, F. Mareuil, H. Ménager, C. Pradal, and C. Blanchet, *Scientific workflows for computational reproducibility in the life sciences: Status, challenges and opportunities*, Future Generation Computer Systems, 75 (2017), pp. 284–298.

[60] L. Courtès and R. Wurmus, *Reproducible and User-Controlled Software Environments in HPC with Guix*, in 2nd International Workshop on Reproducibility in Parallel Computing (RepPar), Vienne, Austria, Aug. 2015.

[61] A. De Coninck, D. Kourounis, F. Verbosio, O. Schenk, B. De Baets, S. Maenhout, and J. Fostier, *Towards Parallel Large-Scale Genomic Prediction by Coupling Sparse and Dense Matrix Algebra*, in 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2015, pp. 747–750.

[62] L. L. Desquilbet, S. Granger, B. Hejblum, A. Legrand, P. Pernot, N. P. Rougier, E. de Castro Guerra, M. Courbin-Coulaud, L. Duvaux, P. Gravier, G. Le Campion, S. Roux, and F. Santos, *Vers une recherche reproductible*, Unité régionale de formation à l'information scientifique et technique de Bordeaux, May 2019.

[63] S. Developers, *Singularityce*.

[64] E. Dolstra, M. De Jonge, E. Visser, et al., *Nix: A safe and policy-free system for software deployment.*, in LISA, vol. 4, 2004, pp. 79–92.

[65] C. Dominik, *The Org Mode 9.1 Reference Manual*, 12th Media Services, 2018.

[66] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct methods for sparse matrices*, Oxford University Press, 2017.

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

[68] ——, *The multifrontal solution of unsymmetric sets of linear equations*, SIAM Journal on Scientific and Statistical Computing, 5 (1984), pp. 633–641.

[69] A. Ern and J.-L. Guermond, *Theory and practice of finite elements*, vol. 159, Springer Science & Business Media, 2013.

[70] L. Euler, *Vollständige Anleitung zur Algebra: von den verschiedenen Rechnungsarten, Verhältnissen und Proportionen. Zweyter Theil*, Auf Kosten C. F. Schiermann, 1771.

[71] A. Falco, *Combler l'écart entre H-Matrices et méthodes directes creuses pour la résolution de systèmes linéaires de grandes tailles*, thèse de doctorat, Université de Bordeaux, June 2019.

[72] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral, *The Spack Package Manager: Bringing Order to HPC Software Chaos*, Supercomputing 2015 (SC'15), Austin, Texas, USA, November 15-20 2015. LLNL-CONF-669890.

[73] M. Ganesh and C. Morgenstern, *High-order FEM–BEM computer models for wave propagation in unbounded and heterogeneous media: Application to time-harmonic acoustic horn problem*, Journal of Computational and Applied Mathematics, 307 (2016), pp. 183–203. 1st Annual Meeting of SIAM Central States Section, April 11–12, 2015.

[74] M. C. Genes, *Parallel application on high performance computing platforms of 3D BEM/FEM based coupling model for dynamic analysis of SSI problems*, CIMNE, 2013, p. 205–216.

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

[76] A. George, J. W. Liu, and E. Ng, *Computer solution of sparse linear systems*, Academic, Orlando, (1994).

[77] D. Goldberg, *What every computer scientist should know about floating-point arithmetic*, ACM Comput. Surv., 23 (1991), pp. 5–48.

[78] J. F. Grcar, *Mathematicians of Gaussian elimination*, Notices of the AMS, 58 (2011), pp. 782–792.

[79] W. Gropp, *MPI: The Complete Reference*, no. v. 2 in Scientific and Engineering Computation, MIT Press, 1998.

[80] T. Gruber, J. Eitzinger, G. Hager, and G. Wellein, *LIKWID*. https://doi.org/10.5281/zenodo.5752537, 12 2021. This research has been partially funded by grants: BMBF 01IH13009 and BMBF 01IH16012C.

[81] W. Hackbusch, *A Sparse Matrix Arithmetic Based on H-Matrices. Part I: Introduction to H-Matrices*, Computing, 62 (1999), pp. 89–108. 10.1007/s006070050015.

[82] W. Hackbusch, *Hierarchical matrices: Algorithms and analysis*, vol. 49, Springer, 2015.

[83] B. Heise, *Parallel Solvers for coupled FEM-BEM equations with applications to non-linear magnetic field problems*, Vieweg+Teubner Verlag, Wiesbaden, 1995, pp. 73–85.

[84] B. Heise and M. Kuhn, *Parallel solvers for linear and nonlinear exterior magnetic field problems based upon coupled FE/BE formulations*, Computing, 56 (1996), pp. 237–258.

[85] P. Hénon, P. Ramet, and J. Roman, *PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems*, Parallel Computing, 28 (2002), pp. 301–321.

[86] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, no. 48, SIAM, 2002.

[87] N. J. Higham and T. Mary, *Solving block low-rank linear systems by LU factorization is numerically stable*, IMA Journal of Numerical Analysis, 42 (2021), pp. 951–980.

[88] K. Hinsen, *Computation in Science (Second Edition)*, IOP Publishing, Sept. 2020.

[89] Inria, *Parallel Sparse matriX package (PaStiX) Handbook*.

[90] H. Jagode, A. Danalis, H. Anzt, and J. Dongarra, *PAPI software-defined events for in-depth performance analysis*, The International Journal of High Performance Computing Applications, 33 (2019), pp. 1113–1127.

[91] N. Kamiya and H. Iwase, *BEM and FEM combination parallel analysis using conjugate gradient and condensation*, Engineering Analysis with Boundary Elements, 20 (1997), pp. 319–326.

[92] N. Kamiya, H. Iwase, and E. Kita, *Parallel computing for the combination method of BEM and FEM*, Engineering Analysis with Boundary Elements, 18 (1996), pp. 223–229. Parallel BEM and Related Methods and Algorithms.

[93] D. E. Knuth, *Literate Programming*, Comput. J., 27 (1984), p. 97–111.

[94] M. Kuhn, *FEM-BEM coupling and parallel multigrid solvers for 3D magnetic field problems*, in European Congress on Computational Methods in Applied Sciences and Engineering ECCOMAS Barcelona, 2000, pp. 1–12.

[95] G. M. Kurtzer, V. Sochat, and M. W. Bauer, *Singularity: Scientific containers for mobility of compute*, PLOS ONE, 12 (2017), pp. 1–20.

[96] X. Lacoste, *Scheduling and memory optimizations for sparse direct solver on multi-core/multi-GPU duster systems*, PhD thesis, Bordeaux, 2015.

[97] J.-Y. L'Excellent, *Multifrontal Methods: Parallelism, Memory Usage and Numerical Aspects*, habilitation à diriger des recherches, École normale supérieure de Lyon - ENS LYON, Sept. 2012.

[98] J. W. Liu, *Modification of the minimum-degree algorithm by multiple elimination*, ACM Transactions on Mathematical Software (TOMS), 11 (1985), pp. 141–153.

[99] ——, *The multifrontal method for sparse matrix solution: Theory and practice*, SIAM review, 34 (1992), pp. 82–109.

[100] B. Lizé, *Solveur direct haute performance*, tech. rep., EADS IW/École centrale Paris, 2009.

[101] B. Lizé, *Résolution Directe Rapide pour les Éléments Finis de Frontière en Électromagnétisme et Acoustique : $\mathcal{H}$-Matrices. Parallélisme et Applications Industrielles.*, PhD thesis, Université Paris 13, 2014.

[102] L. Marchal, *Memory and data aware scheduling*, habilitation à diriger des recherches, École Normale Supérieure de Lyon, Mar. 2018.

[103] D. Merkel, *Docker: Lightweight Linux Containers for Consistent Development and Deployment* , Linux Journal, 2014 (2014).

[104] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 4.0*, June 2021.

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

[106] R. Perera and E. Alarcon, *Parallel algorithms for FE/BE coupling*, 1998.

[107] L. Poirel, *Algebraic domain decomposition methods for hybrid (iterative/direct) solvers*, theses, Université de Bordeaux, Nov. 2018.

[108] P. Raviart and J. Thomas, *A mixed finite element method for 2-nd order elliptic problems*, in Mathematical Aspects of Finite Element Methods, I. Galligani and E. Magenes, eds., vol. 606 of Lecture Notes in Mathematics, Springer Berlin Heidelberg, 1977, pp. 292–315.

[109] V. Rischmuller, M. Haas, S. Kurz, and W. Rucker, *3D transient analysis of electromechanical devices using parallel BEM coupled to FEM*, IEEE Transactions on Magnetics, 36 (2000), pp. 1360–1363.

[110] E. Rothberg and S. C. Eisenstat, *Node selection strategies for bottom-up sparse matrix ordering*, SIAM Journal on Matrix Analysis and Applications, 19 (1998), pp. 682–695.

[111] F.-H. Rouet, *Memory and performance issues in parallel multifrontal factorizations and triangular solutions with sparse right-hand sides*, theses, Institut National Polytechnique de Toulouse - INPT, Oct. 2012.

[112] N. P. Rougier, K. Hinsen, F. Alexandre, T. Arildsen, L. Barba, F. C. Y. Benureau, C. T. Brown, P. de Buyl, O. Caglayan, A. P. Davison, M. A. Delsuc, G. Detorakis, A. K. Diem, D. Drix, P. Enel, B. Girard, O. Guest, M. G. Hall, R. N. Henriques, X. Hinaut, K. S. Jaron, M. Khamassi, A. Klein, T. Manninen, P. Marchesi, D. Mcglinn, C. Metzner, O. L. Petchey, H. E. Plesser, T. Poisot, K. Ram, Y. Ram, E. Roesch, C. Rossant, V. Rostami, A. Shifman, J. Stachelek, M. Stimberg, F. Stollmeier, F. Vaggi, G. Viejo, J. Vitay, A. Vostinar, R. Yurchak, and T. Zito, *Sustainable computational science: the ReScience initiative*, PeerJ Computer Science, 3 (2017), p. e142. 8 pages, 1 figure.

[113] N. Rouxelin, H. Barucq, and S. Tordeux, *Low-order Absorbing Boundary Conditions in HDG discretization of the convected Helmholtz equation*, in Waves 2022 15th International Conference on Mathematical and Numerical Aspects of Wave Propagation, Jul 2022, Palaiseau, France, Palaiseau, France, July 2022.

[114] Y. Saad, *Iterative methods for sparse linear systems*, SIAM, 2003.

[115] S. A. Sauter and C. Schwab, *Boundary Element Methods*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 183–287.

[116] M. Schauer, J. E. Roman, E. S. Quintana-Ortí, and S. Langer, *Parallel Computation of 3-D Soil-Structure Interaction in Time Domain with a Coupled FEM/SBFEM Approach*, Journal of Scientific Computing, 52 (2012), pp. 446–467.

[117] O. Schenk and K. Gärtner, *Parallel Sparse Direct and Multi-recursive iterative linear solvers (PARDISO): User's Guide*. `https://pardiso-project.org/manual/manual.pdf`.

[118] ——, *Solving unsymmetric sparse systems of linear equations with PARDISO*, Future Generation Computer Systems, 20 (2004), pp. 475–487.

[119] A. Schwarzenberg-Czerny, *On matrix factorization and efficient least squares solution*, Astronomy and Astrophysics Supplement Series, 110 (1995), p. 405.

[120] M. W. Sid Lakhdar, *Scaling the solution of large sparse linear systems using multifrontal methods on hybrid shared-distributed memory architectures*, PhD thesis, 2014. Thèse de doctorat dirigée par L'Excellent, Jean-Yves Informatique Lyon, École normale supérieure 2014.

[121] P. Solin, K. Segeth, and I. Dolezel, *Higher-Order Finite Element Methods*, Studies in Advanced Mathematics, CRC Press, 2003.

[122] L. Solis-Vasquez, D. Santos-Martins, A. Koch, and S. Forli, *Evaluating the Energy Efficiency of OpenCL-accelerated AutoDock Molecular Docking*, in 2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2020, pp. 162–166.

[123] L. Stanisic, A. Legrand, and V. Danjean, *An Effective Git And Org-Mode Based Workflow For Reproducible Research*, Operating Systems Review, 49 (2015), pp. 61–70.

[124] T. Steinmetz, N. Godel, G. Wimmer, M. Clemens, S. Kurz, and M. Bebendorf, *Efficient Symmetric FEM-BEM Coupled Simulations of Electro-Quasistatic Fields*, IEEE Transactions on Magnetics, 44 (2008), pp. 1346–1349.

[125] L. Trefethen and D. Bau, *Numerical linear algebra*, Miscellaneous Bks, Society for Industrial and Applied Mathematics, 1997.

[126] A. Wang, N. Vlahopoulos, and K. Wu, *Development of an energy boundary element formulation for computing high-frequency sound radiation from incoherent intensity boundary conditions*, Journal of Sound and Vibration, 278 (2004), pp. 413–436.

[127] M. Yannakakis, *Computing the minimum fill-in is NP-complete*, SIAM Journal on Algebraic Discrete Methods, 2 (1981), pp. 77–79.

[128] F. Zhang, *The Schur complement and its applications*, vol. 4, Springer Science & Business Media, 2006.

[129] P. Zhang, T. Wu, and R. Finkel, *Parallel computation for acoustic radiation in a subsonic nonuniform flow with a coupled FEM/BEM formulation*, Engineering Analysis with Boundary Elements, 23 (1999), pp. 139–153.

[130] O. Zienkiewicz and R. Taylor, *The finite element method*, vol. 3, McGraw-hill London, 1977.