On the parallel scalability of hybrid linear solvers for large 3D problems

Azzam Haidar

N ^{o} D'ORDRE 2623

THÈSE

présenté en vue de l'obtention du titre de

DOCTEUR DE L'INSTITUT NATIONAL POLYTECHNIQUE DE TOULOUSE

Spécialité: Mathématiques, Informatique et Télécommunications

par

Azzam HAIDAR

CERFACS

Sur l'extensibilité parallèle de solveurs linéaires hybrides pour des problèmes tridimensionels de grandes tailles

On the parallel scalability of hybrid linear solvers for large 3D problems

---

Thèse présentée le 23 Juin 2008 à Toulouse devant le jury composé de:

Fréderic Nataf	Directeur de Recherche CNRS, Lab Jacques-Louis Lions	France	Rapporteur
Ray Tuminaro	Chercheur senior, Sandia National Laboratories	USA	Rapporteur
Iain Duff	Directeur de Recherche RAL et CERFACS	Royaume-Uni	Examinateur
Luc Giraud	Professeur, INPT-ENSEEIHT	France	Directeur
Stéphane Lanteri	Directeur de Recherche INRIA	France	Examinateur
Gérard Meurant	Directeur de Recherche CEA	France	Examinateur
Jean Roman	Professeur, ENSEIRB-INRIA	France	Examinateur

Thèse préparée au CERFACS, Report Ref:TH/PA/08/93

Résumé

La résolution de très grands systèmes linéaires creux est une composante de base algorithmique fondamentale dans de nombreuses applications scientifiques en calcul intensif. La résolution performante de ces systèmes passe par la conception, le développement et l'utilisation d'algorithmes parallèles performants. Dans nos travaux, nous nous intéressons au développement et l'évaluation d'une méthode hybride (directe/itérative) basée sur des techniques de décomposition de domaine sans recouvrement. La stratégie de développement est axée sur l'utilisation des machines massivement parallèles à plusieurs milliers de processeurs. L'étude systématique de l'extensibilité et l'efficacité parallèle de différents préconditionneurs algébriques est réalisée aussi bien d'un point de vue informatique que numérique. Nous avons comparé leurs performances sur des systèmes de plusieurs millions ou dizaines de millions d'inconnues pour des problèmes réels $3D$.

Mots-clés: Décomposition de domaines, Méthodes itératives, Méthodes directes, Méthodes hybrides, Complément de Schur, Systèmes linéaires denses et creux, Méthodes de Krylov, GMRES, Flexible GMRES, CG, Calcul haute performace, Deux niveaux de parallèlisme, Calcul parallèle distribué, Calcul sientifique, Simulation numériques de grande taille, Techniques de préconditionnement, Préconditionneur de type Schwarz additive.

Abstract

Large-scale scientific applications and industrial simulations are nowadays fully integrated in many engineering areas. They involve the solution of large sparse linear systems. The use of large high performance computers is mandatory to solve these problems. The main topic of this research work was the study of a numerical technique that had attractive features for an efficient solution of large scale linear systems on large massively parallel platforms. The goal is to develop a high performance hybrid direct/iterative approach for solving large 3D problems. We focus specifically on the associated domain decomposition techniques for the parallel solution of large linear systems. We have investigated several algebraic preconditioning techniques, discussed their numerical behaviours, their parallel implementations and scalabilities. We have compared their performances on a set of 3D grand challenge problems.

Keywords: Domain decomposition, Iterative methods, Direct methods, Hybrid methods, Schur complements Linear systems, Krylov methods, GMRES, flexible GMRES, CG, High performance computing, Two levels of parallelism, Distributed computing, Scientific computing, Large scale numerical simulations, Preconditioning techniques, Additive Schwarz preconditioner.

Remerciements

C'est une habitude saine que de rendre mérite, avec mon enthousiasme le plus vif et le plus sincère à tous ceux qui à leur manière ont contribué mener ce travail à bien. Je désire alors exprimer ma profonde gratitude :

Envers Luc GIRAUD, pour avoir accepté de me diriger patiemment, mais aussi spécialement pour m'avoir accordé sa confiance en me laissant toute liberté dans mes initiatives, tout en étant suffisamment attentif pour que je ne m'égare pas sur des pistes peu prometteuses. Mais également pour sa disponibilité et sa générosité exceptionnelles, il s'est montré très disponible pour toutes mes questions et problèmes à résoudre. Pour sa gentillesse, il a pris le temps de lire, relire et corriger soigneusement cette thèse. J'ai pu profiter des ses compétences scientifiques, de ses conseils pertinents et précieux. Par son charisme, son dynamisme et sa passion exceptionnelle pour la recherche, il m'a beaucoup appris pour évoluer dans le monde de la recherche et de l'industrie.

Envers le directeur de l'équipe algorithme parallèle Iain DUFF, je tiens à lui exprimer ma très vive reconnaissance. Monsieur j'ai eu l'honneur de travailler ces trois années au sein de votre équipe. Cette expérience professionnelle me sera très bénéfique en ce qui concerne mes projets d'avenir. Je tiens à vous remercier pour toutes vos remarques toujours pertinentes et vos idées attentives.

A Serge GRATTON, et Xavier VASSEUR, je tiens à vous adresser mes sincères remerciements pour vos encouragements généreux et vos suggestions judicieuses : ils m'ont été précieux. Vous vous êtes montrés très disponibles pour toutes mes discussions vous m'avez aidé avec grande gentillesse ce qui m'a permis de m'ouvrir à d'autres horizons.

A l'assistance de plusieurs personnes de l'équipe MUMPS. Particulièrement je tiens à remercier Jean-Yves L'EXCELLENT et Patrick AMESTOY pour leur support et leurs aides très précieuses de tous moments. Vous avez pris le temps de développer et de débuguer de nouvelles fonctionnalités qui m'ont été très bénéfiques.

Aux membres du Jury qui m'ont honoré en acceptant d'évaluer mon travail et d'être présent ici aujourd'hui. Chacun d'eux mérite un remerciement particulier pour m'avoir accordé son attention. Un sincère remerciement à Fréderic NATAF et Ray TUMINARO qui m'ont fait l'honneur d'accepté la charge d'être rapporteurs. Je leur suis reconnaissant pour le temps qu'ils ont consacré à la lecture de ce manuscrit et pour l'intérêt qu'ils ont montré pour mon travail. J'aimerais remercier sincèrement Iain DUFF qui m'a fait l'honneur de présider mon jury, qui m'a beaucoup encouragé et inspiré. Aussi bien également Luc GIRAUD qui m'a motivé et qui m'a considérablement soutenu durant toutes mes recherches de thèse. Je désire aussi remercier vivement Stéphane LANTERI pour ces conseils amicaux ainsi que ses discussions importantes et son accueil passionnant lors de mon séjour dans son équipe à l'INRIA-Sophia-Antipolis. Je désire aussi remercier très vivement Gérard MEURANT pour toutes ses remarques scientifiques, ses suggestions et ses conseils judicieux qu'il m'a souvent transmis grâce à son caractère chaleureux : ils m'ont été très précieux. Il n'a jamais manqué une occasion pour m'encourager. Enfin je tiens à remercier profondément Jean ROMAN pour avoir accepté de participer à ce jury, ainsi que pour ses conversations enrichissantes lors de mes visites à l'INRIA Bordeaux. Egalement je tiens à remercier toutes les personnes qui ont assisté à cette soutenance de thèse.

C'est un grand privilège d'effectuer sa thèse au CERFACS, j'exprime ma très vive reconnaissance à Jean-Claude ANDRE le directeur du laboratoire. Monsieur, le CERFACS est un bon endroit de convivialité et j'ai eu l'honneur d'effectuer ma thèse au sein de votre laboratoire. Heureuse-

ment que l'équipe CSG du CERFACS était là pour m'aider à me dépatouiller avec les expériences et pour venir me sauver lorsque j'étais perdu au fin fond des tracas informatiques. Merci pour votre délicatesse et votre attention. Merci également au travail et à la gentillesse de l'administration, Brigitte, Chantal, Dominique, Lydia, Michèle, et Nicole.

Egalement une pensée pour tous ceux avec qui j'ai partagé les moments qui font la vie d'un étudiant, les discussions dans les couloirs. Un remerciement particulier aux membres de l'équipe ALGO avec qui j'ai partagé de très beaux moments chaleureux, les déjeuners (départ à 12h30 tapante !), les pauses café/thé, les sudokus, les affaires de logiques du "le Monde", les sorties, je glisse un remerciement amical à cette joyeuse bande.

To the Samcef project, especially to Stéphane PRALET, who provided us with the structural mechanics problems support, for the help he gave me among this work, and for kindly developing special functionality allowing me to use the samcef code, for his advice and numerous suggestions.

To all the members of the Consortium Seiscope project with whom I had fruitful discussion and who, provides me the seismic applications that enables me to progress in my work: Florent SOURBIER, Jean VIRIEUX, Romain BROSSIER and StÂťephane OPERTO.

To the INRIA-NACHOS team. I must thank Stéphane LANTERI who opened me the door of an enriching collaboration, who interest in my work and who gave me constructive advice.

To Masha SOSONKINA and Layne WATSON whose deserve grateful thanks for providing me with an huge amount of simulations hours on the Virginia Tech supercomputer, and for many helpful discussions and advices.

Un grand grand Merci également a toutes celles et ceux qui ont participé directement ou indirectement à ce travail. Tous celles et ceux qui m'ont témoigné leur amitié, qui m'ont apporté leur aide, qui m'ont accompagné pendant cette aventure et que je ne peux citer ici, vous êtes nombreux...

Cette aventure de m'est pas propre, enfin, c'est quand même l'aboutissement de toute une scolarité, je voudrais remercier chaleureusement ma petite maman à qui je dois beaucoup et qui a toujours été une fervente supportrice avec mes deux sœurs et mon frère. Kamil, un grand merci du cœur a toi mon père, pour m'avoir donné l'envie d'être heureux et pour m'avoir toujours soutenu dans mes choix. Et tout le reste de ma famille, Merci pour votre soutien, vos encouragements et votre présence dans les moments difficiles. Je vous témoigne ici toute ma reconnaissance et tout mon amour.

Contents

I

Part I

Solving large linear systems on large parallel distributed platforms

Introduction

La résolution de très grands systèmes linéaires creux est une composante de base algorithmique fondamentale dans de nombreuses applications scientifiques de calcul intensif. Il s'agit souvent l'étape la plus consommatrice aussi bien en temps CPU qu'en espace mémoire. La taille des systèmes utilisés pour les grandes simulations complexes fait que le calcul à hautes performances est aujourd'hui incontournable (on doit résoudre aujourd'hui des systèmes de plusieurs millions ou dizaines de millions d'inconnues pour des problèmes réels 3D). La résolution performante de ces systèmes passe par la conception, le développement et l'utilisation d'algorithmes parallèles performants qui possèdent des propriétés d'extensibilité pour permettre le passage à l'échelle et une exploitation efficace de plateformes de calcul avec un grand nombre de processeurs.

Dans nos travaux, nous nous intéressons au développement et l'évaluation d'une méthode hybride (directe/itérative) basée sur des techniques de décomposition de domaine sans recouvrement. Il s'agit d'une méthodologie générale qui permet en particulier de résoudre des systèmes linéaires issus de la discrétisation d'équations aux dérivées partielles. La stratégie de développement est axée sur l'utilisation des machines massivement parallèles de plusieurs milliers de processeurs.
De ces travaux se dégagent trois contributions principales qui structurent le manuscrit de thèse ; celui-ci comporte :

- La formulation et l'analyse de différents préconditionneurs de type Schwarz additif algébrique pour des problèmes 3D (Chapitre 2-3). Afin de réduire les coÃżts informatiques de leurs mises en œuvre nous proposons des variantes qui exploitent des algorithmes en précision mixte (32-bits et 64-bits) ainsi que des approximations creuses. Pour des implantations sur très grand nombre de processeurs nous considérons soit des corrections par un préconditionnement de deuxième niveau (correction grille grossière) soit des mises en œuvre algorithmiques exploitant deux niveaux de parallélisme. Le Chapitre 4 est dédié à la description de l'implantation parallèle de ces approches.

- L'étude systématique de l'extensibilité et l'efficacité parallèle de ces préconditionneurs est réalisée aussi bien d'un point de vue informatique que numérique. Dans, ce contexte des problèmes modèles académiques de type équations de diffusions $3D$ (Chapitre 5) ou de convection-diffusion (Chapitre 6) sont considérés. L'étude est menée sur des machines jusqu'à 2048 processeurs pour résoudre des problèmes $3D$ à plus de 50 millions d'inconnues. Ces études ont été menées sur des machines telles que le SystemX de Virginia Tech ou l'IBM Blue-Gene du CERFACS.

- La validation de notre approche sur des cas réels est enfin réalisée sur des problèmes de mécanique des structures en maillages non-structurés (en collaboration avec la société SAMTECH-Chapitre 7) et en imagerie sismique (en collaboration avec le consortium SEISCOPE - Chapitre 8). Dans ce dernier cas, on s'intéresse à la résolution des équations de Helmholtz en régime fréquentiel. Plusieurs simulations sur des cas réels $2D$ et $3D$ ont été réalisés. Dans ces chapitres, nous présentons notamment les détails des performances parallèles de notre approche exploitant deux nivaux de parallélisme.

Ce manuscrit débute par un exposé du cadre mathématique de notre étude et se termine par un bilan de cette étude, ainsi qu'une discussion sur quelques pistes de travaux futurs avec des applications possibles sur les équations de Maxwell en régime harmonique (collaboration avec l'équipe NACHOS, INRIA-Sophia-Antipolis).

Part I: résumé

Les méthodes de décomposition de domaines ont été développées pour résoudre des problèmes complexes ou de grande taille, et plus récemment pour traiter des maillages non conformes, ou coupler différents modèles. Nous introduisons dans ce chapitre les bases mathématiques des méthodes de décomposition de domaines avec et sans recouvrement appliquées à la résolution de grands systèmes linéaires $Ax = b$. Les avantages de telles méthodes hybrides directes/itératives qu'elles sont des approches:

- plus robustes que les méthodes itératives classiques et moins coûteuses en mémoire et en calcul que les méthodes directes;
- bien adaptées aux calculateurs parallèles;
- qui permettent la réutilisation de codes séquentiels existants au niveau des calculs locaux. Ces techniques constituent une approche modulaire du parallélisme.

Méthodes de décomposition de domaines avec recouvrement

Elles consistent à découper le domaine de calcul en sous-domaines qui se recouvrent comme le montre la Figure 1. Elles permettent une actualisation simple des solutions locales sur la base de conditions aux limites de type Dirichlet ou Neumann ou une combinaison des deux aux frontières artificielles introduites par la décomposition.

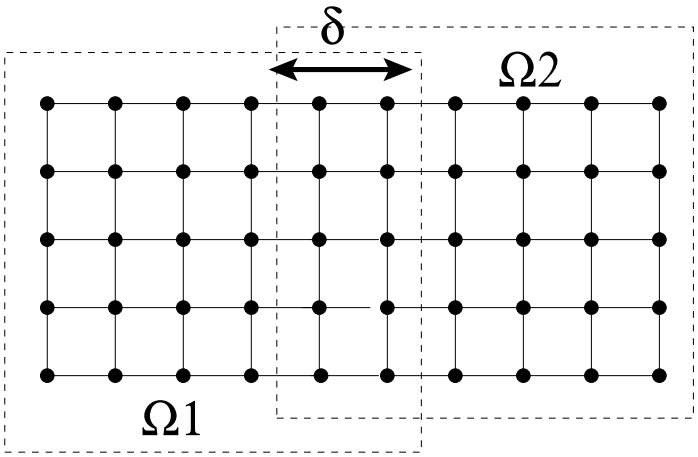


Figure 1: Décomposition de domain avec recouvrement.

Dans une classe d'approches basées sur cette approche, la résolution du système linéaire $Ax = b$ se fait par une méthode itérative appliquée en combinaison avec un préconditionneur. Cette méthode se caractérise par une complexité arithmétique accrue du fait de la replication des degrés de liberté (ddls) dans les roues du recouvrement. Leur inconvénient principal est donc de compliquer quelque peu la mise en œuvre numérique, surtout lors de la résolution de problèmes 3D sur des géométries complexes.

Méthodes de décomposition de domaines sans recouvrement

Elles consistent à découper le domaine de calcul en sous-domaines sans recouvrement comme le montre la Figure 2; et à reformuler le problème en un problème équivalent restreint à l'interface

uniquement. Une classe de ces méthodes de décompositions de domaines sans recouvrement ou méthodes du complément de Schur s'apparentent aux méthodes d'élimination de Gauss par blocs. Elles consistent à ramener la résolution d'un problème global posé sur l'ensemble des degrés de liberté issus de la discrétisation du domaine de calcul, à la résolution d'un problème de taille moindre posé sur les nœuds situés sur les interfaces entre les sous-domaines voisins dans la partition du domaine de calcul.

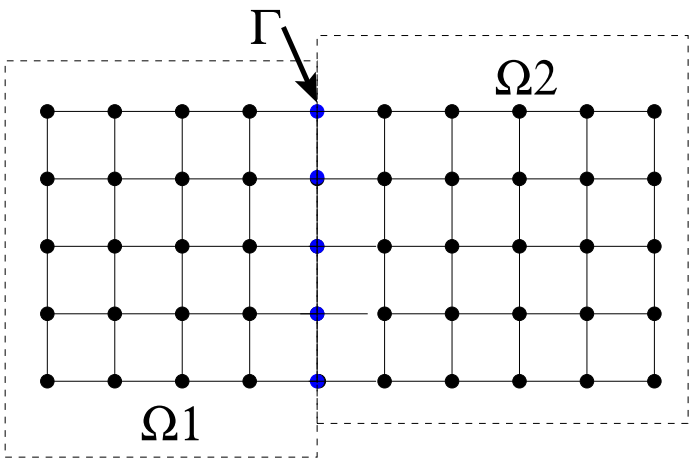


Figure 2: Décomposition de domain sans recouvrement, méthode du complément du Schur.

Soit A la matrice associée à la discrétisation du domaine Ω. Soit Γ l'ensemble des indices des nœuds appartenant à l'interface entre les sous-domaines voisins. Groupant les indices correspondant à Γ et ceux correspondant aux intérieurs des sous-domaines; le système linéaire (1) s'écrit sous la forme (2). En appliquant une élimination de Gauss par bloc on aboutit au système (3). l'opérateur associé au problème d'interface et résultant du processus d'élimination de Gauss par bloc est appelé complément de Schur $\mathcal{S}$. Du point de vue du problème continu, cela revient à la définition d'un nouvel opérateur, l'opérateur de Steklov-Poincaré, qui agit sur les variables d'interface et dont la discrétisation donne le complément du Schur de l'opérateur du problème global;

$$\mathcal{A} \qquad x \quad = \quad b \quad , \tag{1}$$

$$\begin{pmatrix} \mathcal{A}_{1,1} & 0 & \mathcal{A}_{1,\Gamma} \\ 0 & \mathcal{A}_{2,2} & \mathcal{A}_{2,\Gamma} \\ \mathcal{A}_{\Gamma,1} & \mathcal{A}_{\Gamma,2} & \mathcal{A}_{\Gamma,\Gamma} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_\Gamma \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_\Gamma \end{pmatrix}, \tag{2}$$

$$\begin{pmatrix} \mathcal{A}_{1,1} & 0 & \mathcal{A}_{1,\Gamma} \\ 0 & \mathcal{A}_{2,2} & \mathcal{A}_{2,\Gamma} \\ 0 & 0 & \mathcal{S} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_\Gamma \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_\Gamma - \sum_{i=1}^{2} \mathcal{A}_{\Gamma,i}\mathcal{A}_{i,i}^{-1} \quad b_i \end{pmatrix}. \tag{3}$$

Ainsi résoudre le système linéaire $Ax = b$ revient à résoudre le système reduit ou le complément du Schur $\mathcal{S}x_\Gamma = f$ ensuite résoudre simultanement les problèmes intérieurs $\mathcal{A}_i x_i = b_i - \mathcal{A}_{i,\Gamma} x_\Gamma$. Soit $\mathcal{S}_k$ le complément de Schur local en mémoire distribuée, donc c'est une matrice pleine obtenue en éliminant les degrés de liberté internes au sous-domaine k. Alors l'operateur $\mathcal{S}$ s'obtient en

sommant les contributions des différents sous-domaines, et s'écrit dans le cas géneral sous la forme suivante:

$$\mathcal{S} = \sum_{i=1}^{N} \mathcal{R}_{\Gamma_i}^T \mathcal{S}^{(i)} \mathcal{R}_{\Gamma_i}. \tag{4}$$

Le problème d'interface ainsi posé est alors résolu par une méthode itérative adaptée, la plupart des fois une méthode de Krylov. Généralement, ces méthodes sont employées en combiniaison avec un préconditionnement. Ainsi, nous introduisons notre préconditionneur parallèle qui est nommé Schwarz additif algébrique.

Pour être réellement efficaces, non seulement il faut avoir une bonne vitesse de convergence mais, de plus, il faut que la méthode choisie soit utilisable éfficacement sur des calculateurs parallèles massivement parallèles. Nous illustrons une étude sur les méthodes de Krylov et leur parallélisation. Il existe plusieurs approches basées sur les espaces de Krylov. Une première classe composée des méthodes de résidu minimal. Elles minimisent le résidu $r_k = b - Ax_k$, ce qui conduit à résoudre un problème de moindre carrés de petite taille. Des exemples de ces méthodes sont MINRES pour les matrices symétriques indéfinies et GMRES pour les matrices quelconques. Une deuxième classe est basée sur des principes d'orthogonalisation dans lesquelles on impose que les résidus soient orthogonaux à l'espace de Krylov. Un exemple de ces méthodes est la méthode du Gradient Conjugué pour les matrices symétriques définies positives. Dans le Chapitre 2, nous décrivons trois de ces méthodes; GMRES, flexible GMRES et CG qui sont les plus utilisées.

Préconditionneur Schwarz additif algébrique

Dans la littérature, il existe différents types de préconditionneurs associés à la méthode de complément de Schur dont par exemple le préconditionneur de type tels BPS, Neumann-Dirichlet ou encore Neumann-Neumann. On définit notre préconditionneur de type Schwarz additif pour le complément de Schur de manière algébrique, ce qui le rend adapté à des situations souvent rencontrer en industrie où le maillage est non-structuré. Pour chaque sous-domaine Ω_k , on appelle $\bar{\mathcal{S}}$ le complément de Schur local assemblé. Ce complément de Schur assemblé est formé par la somme des contributions des compléments de Schur des sous-domaines voisins au sous-domaine k . Autrement dit, les blocs diagonaux de $\bar{\mathcal{S}}_k$ seront augmentés par les contributions du bloc diagonal correspondant de chacun du Schur des sous-domaines voisins au sous-domaine k .

En écriture matricielle, le complement de Schur globale $\mathcal{S}$ pour une décomposition en tranches s'écrit sous la forme suivante:

$$\mathcal{S} = \begin{pmatrix} \ddots & & & & \\ & \mathcal{S}_{kk} & \mathcal{S}_{k\ell} & & \\ & \mathcal{S}_{\ell k} & \mathcal{S}_{\ell\ell} & \mathcal{S}_{\ell m} & \\ & & \mathcal{S}_{m\ell} & \mathcal{S}_{mm} & \mathcal{S}_{mn} \\ & & & \mathcal{S}_{nm} & \mathcal{S}_{nn} \end{pmatrix}. \tag{5}$$

On définit le préconditionneur Schwarz additif algébrique de la façon suivante (6) qui n'est autre que la somme des inverses du complément de Schur local assemblé comme le montre l'équation (7)

$$\mathcal{M} = \begin{pmatrix} \ddots & & & & \\ & \boxed{\begin{matrix} \mathcal{S}_{kk} & \mathcal{S}_{k\ell} \\ \mathcal{S}_{\ell k} & \mathcal{S}_{\ell\ell} \end{matrix}}^{-1} & & & \\ & & \boxed{\begin{matrix} \mathcal{S}_{\ell\ell} & \mathcal{S}_{\ell m} \\ \mathcal{S}_{m\ell} & \mathcal{S}_{mm} \end{matrix}}^{-1} & & \\ & & & \boxed{\begin{matrix} \mathcal{S}_{mm} & \mathcal{S}_{mn} \\ \mathcal{S}_{nm} & \mathcal{S}_{nn} \end{matrix}}^{-1} & \end{pmatrix}, \tag{6}$$

$$\mathcal{M}_{\mathrm{d-64}} = \sum_{i=1}^{N} \mathcal{R}_{\Gamma_i}^T (\bar{\mathcal{S}}^{(i)})^{-1} \mathcal{R}_{\Gamma_i}. \tag{7}$$

On note que dans le cas des problèmes $3D$, les matrices locales du complément de Schur deviennent très grandes et leur utilisation comme préconditionneur local devient coûteuse aussi bien en calcul qu'en stockage. Il est donc nécessaire de construire une forme allégée du préconditionneur, ce qui nous a amené a considérer:

- le préconditionneur Schwarz additif algébrique creux $M_{\mathrm{sp-64}}$,
- le préconditionneur Schwarz additif algébrique en précision mixte $M_{\mathrm{d-mix}}$,
- ou encore une combinaison des deux, c.à.d. préconditionneur algébrique additif Schwarz creux en précision mixte $M_{\mathrm{sp-mix}}$.

Deux niveaux de parallélisme

Le but de cette section est d'expliquer comment exploiter efficacement le parallélisme pour l'implémentation d'applications massivement parallèles. La plupart des algorithmes et des codes existant sont implémentés avec un seul niveau de parallélisme (*1-level*). Cependant suivant notre expérience surtout dans les domaines applicatifs on s'est rendu compte que parfois un seul niveau de parallélisme n'est pas suffisant et qu'on pourrait améliorer la robustesse et l'efficacité de la méthode hybride par l'utilisation d'un deuxième niveau de parallélisme (*2-levels*). On cite deux de ces motivations de base qui sont á l'origine du développement d'un algorithme à deux niveaux de parallélisme :

- Dans plusieurs applications industrielles, augmenter le nombre de sous-domaines fait accroître le nombre d'itérations nécessaire au processus itératif pour converger; cela notamment dans des applications de mécanique de structures (Chapitre 7) et dans des applications séismique (Chapitre 8). Pour cela nous avons proposé l'utilisation de deux niveaux de parallélisme comme une amélioration du comportement numérique de la méthode. Autrement dit, au lieu d'augmenter le nombre de sous-domaines, on alloue plusieurs processeurs par sous-domaine et on garde petit le nombre de sous-domaines, de cette façon on garde les propriétés numériques du système et on réalise une simulation plus efficacement.
- Les grandes simulations industrielles nécessitent un très grand espace de stockage par sous-domaine. La plupart des fois cet espace n'est pas disponible sur un processeur. Alors sur des machines parallèles de type SMP, les codes standard (*1-level*) utilisent un seul processeur du nœud de la machine laissant les autres processeurs du nœud en état "inactif" exécutant une simulation avec un pourcentage inacceptable de la puissance crête du nœud. L'idée ici est de ne pas perdre l'efficacité de ces processeurs "inactif" et d'exploiter cette ressource de calcul en les allouant au sous-domaine correspondant. Ici l'exploitation de deux niveaux de parallélisme est vu comme une amélioration de la performance parallèle d'un algorithme.

L'exploitation de deux niveaux de parallélisme est illustrée par des exemples pratiques dans les Chapitre 7 et 8.

Chapter 1

Introduction

The solution of many large sparse linear systems depends on the simplicity, flexibility and availability of appropriate solvers. The goal of this thesis is to develop a high performance hybrid direct/iterative approach for solving large $3D$ problems. We focus specifically on the associated domain decomposition techniques for the parallel solution of large linear systems. We consider a standard finite element discretization of Partial Differential Equations (PDE's) that are used to model the behaviour of some physical phenomena. The use of large high performance computers is mandatory to solve these problems. In order to solve large sparse linear systems either direct or iterative solvers can be solved.

Direct methods, are widely used and are the solvers of choice in many applications especially when robustness is the primary concern. It is now possible to solve $3D$ problems with a couple of million equations in a robust way with the direct solvers fully exploiting the parallel algorithmic of blockwise solvers optimized for modern parallel supercomputers. Unfortunately, solving large sparse linear systems by these methods has often been quite unsatisfactory, especially when dealing with practical "industrial" problems ($3D$ problems can lead to systems with millions or hundreds of millions of unknowns). They scale poorly with the problem size in terms of computing times and memory requirements, especially on problems arising from the discretization of large PDE's in three dimensions of space.

On the other hand, iterative solvers are commonly used in many engineering applications. They require less storage and often require fewer operations than direct methods, especially when an approximate solution of relatively low accuracy is sought. Unfortunately, the performance of these latter techniques depends strongly on the spectral properties of the linear system. Both the efficiency and the robustness can be improved by using an efficient preconditioner. It is widely recognized that preconditioning is the most critical ingredient in the development of efficient solvers for challenging problems in scientific computation.

The preconditioner is an operator that transforms the original linear system into another one having the same solution but better properties with respect to the convergence features of the iterative solver used. Generally speaking, the preconditioner attempts to improve the spectral properties of the matrix associated with the linear system. For symmetric positive definite problems (SPD), an upper bound of the rate of convergence of the conjugate gradient method, depends on the distribution of the eigenvalues (in other terms of the condition number of the system). Hopefully, the preconditioned system will have a smaller condition number. For unsymmetric problems the situation is more complicated, we do not have any convergence bound based on the distribution of the eigenvalues for iterative solvers like GMRES, but some arguments exist for diagonalizable matrices [39]. Nevertheless, a clustered spectrum away from zero often results in faster convergence.

In general, a good preconditioner must satisfy many constraints. It must be inexpensive to compute and to apply in terms of both computational time and memory storage. Since we are interested in parallel applications, the construction and application of the preconditioner of the

system should also be parallelizable and scalable. That is the preconditioned iterations should converge rapidly, and the performance should not be degraded when the number of processors increases. Notice that these two requirements are often in competition with each other, it is necessary to strike a compromise between the two needs.

There are two classes of preconditioners, one is to design specialized algorithms that are close to optimal for a narrow type of problems, whereas the second is a general-purpose algebraic method. The formers can be very successful, but require a complete knowledge of the problem which may not always be feasible. Furthermore, these problem specific approaches are generally very sensitive to the details of the problem, and even small changes in the problem parameters can penalize the efficiency of the solver. On the other hand, the algebraic methods use only information contained in the coefficient of the matrices. Though these techniques are not optimal for any particular problem, they achieve reasonable efficiency on a wide range of problems. In general, they are easy to compute and to apply and are well suited for irregular problems. Furthermore, one important aspect of such approaches is that they can be adapted and tuned to exploit specific applications.

Thus one of the interesting and powerful framework that reduces the complexity of the solvers for solving large $3D$ linear system in a massively parallel environment is to use hybrid approaches that combine iterative and direct methods. The focus of this thesis is on developing effective parallel algebraic preconditioners, that are suitable and scalable for high performance computation. They are based on the substructuring domain decomposition approach. Furthermore, we investigate work on multi-level parallel approaches to be able to exploit large number of processors with reasonable efficiency.

This manuscript is organized as follows.

Chapter 2 outlines the basic ingredients that are involved in the hybrid linear solvers. The main developments in the area of domain decomposition methods and preconditioning techniques from a historical perspective are presented. Furthermore, these methods are most often used to accelerate Krylov subspace methods. In that respect, we briefly present the Krylov subspace solvers we have considered for our numerical experiments. Finally, we introduce some basic concepts of the backward error analysis that enables us to make fair comparisons between the various considered techniques.

The availability of preconditioners is essential for a successful use of iterative methods; consequently the research on preconditioners has moved to center stage in the recent years. **Chapter 3** is mainly devoted to addressing the properties of the algebraic additive Schwarz preconditioners studied in this thesis. Section 3.2 is dedicated to the algebraic description of the additive Schwarz preconditioner. The main lines of the Schur complement approach are presented, and the main aspects of parallel preconditioning are introduced. We propose and study variants of the preconditioner. In Section 3.3 we intend to reduce the storage and the computational cost by using sparse approximation. We propose mixed precision computation to enhance performance, by using a combination of 32-bit and 64-bit arithmetics. Thus, we present in Section 3.4 a mixed precision variant of the additive Schwarz preconditioner and we motivate this idea by the fact that many recent processors exhibit 32-bit precision computing performance that is significantly higher than 64-bit calculation. Moreover, we study in Section 3.5 a two-level preconditioner based on algebraic constructions of the coarse space component. This coarse space ingredient aims at capturing the global coupling amongst the subdomains.

In **Chapter 4**, we discuss the parallel implementations of these techniques on distributed parallel machines. Another line of research that we propose, is to develop a *2-level parallel* algorithm that attempt to express parallelism between the subproblems but also in the treatment of each subproblem. Using this latter method, we introduce a new level of task and data parallelism that allows us to achieve high performance and to provide an efficient parallel algorithm for massively

parallel platforms. An efficient implementation in this framework requires a careful analysis of all steps of the hybrid method. A distributed data structure, capable of handling in a efficient way the main operations required by the method, is defined. Furthermore, several implementations issues are addressed, ranging from the way to implement parallel local solvers to the data exchange within the multi-level parallel iterative kernels.

The parallel performance and the numerical scalability of the proposed preconditioners are presented on a set of $3D$ academic model problems. This study is divided into two chapters.

Chapter 5 focuses on the symmetric positive definite systems arising from diffusion equations. We analyze in Section 5.3, the numerical behaviours of the sparsified and the mixed arithmetic variants and we compare them with the classical additive Schwarz preconditioner. Then in Section 5.4, the numerical scalability and the parallel efficiency obtained on massively distributed memory supercomputers using MPI as message library illustrate the scalability of the proposed preconditioners.

Chapter 6 is devoted to the convection-diffusion equations that leads to unsymmetric problems. We quantify the numerical behaviours of the proposed variants of the additive Schwarz preconditioner in Section 6.3. A crucial characteristic of a preconditioner is the way its response to disturbance changes when the system parameters change. For that, we intend to evaluate the sensitivity of the preconditioners to heterogeneous discontinuities with or without anisotropies in the diffusion coefficients, and to the convection dominated term. Results on parallel performance and numerical scalability on massively parallel platforms are presented in Section 6.4.

Large-scale scientific applications and industrial numerical simulations are nowadays fully integrated in many engineering areas such as aeronautical modeling, structural mechanics, geophysics, seismic imaging, electrical simulation and so on. Hence it is important to study the suitability and the performance of the proposed methods for such real application problems. In that respect, we investigate these aspects in two different application areas presented in two chapters.

In **Chapter 7** we focus on a specific engineering area, structural mechanics, where large problems have to be solved. Our purpose it to evaluate the robustness and possibly the performance of our preconditioner for the solution of the challenging linear systems that are often solved using direct solvers. We consider two different classes of problems. The first one is related to the solution of the linear elasticity equations. The second class of problems, probably more realistic in term of engineering applications, is still related to linear elasticity with constraints such as rigid bodies and cyclic conditions. We report on the numerical study and the parallel performance analysis using our favorite preconditioners. Moreover, we discuss how the parallel efficiency can be improved by using the *2-level parallel* approach. Analysis and experiments show that when using *2-levels* of parallelism the algorithm runs close to the aggregate performance of the available computing resources.

In **Chapter 8**, we investigate work in seismic modeling based on the frequency-domain full-waveform tomography approaches. We analyze the accuracy of the hybrid approach by comparing the results to those obtained from an analytical solution. Then a parallel performance study for respectively large $2D$ and $3D$ models arising in geophysical applications are reported. Finally, we evaluate the parallel performance of the *2-level parallel* algorithm. A preliminary investigation carried out on a set of numerical experiments confirm that the *2-level parallel* method allows us to attain a high level of performance and parallel efficiency.

The development of efficient and reliable preconditioned hybrid solvers is the key for successful solution of many large-scale problems. We have attempted to highlight some of the studies and developments that have taken place in the course of the three years of the thesis. There are

many further important problems and ideas that we have not been addressed. We discuss a few perspectives in a **Conclusion** part at the end of this manuscript.

Chapter 2

Some basics on hybrid linear solvers

In this chapter we briefly describe the basic ingredients that are involved in the hybrid linear solvers considered in this manuscript. The approach described in this work borrows some ideas to some classical domain decomposition techniques that are presented in Section 2.1. In this section some popular and well-known domain decomposition preconditioners are described from an algebraic perspective. Numerical techniques that rely on decomposition with overlap are described in Section 2.1.2 and some approaches with non-overlapping domains are discussed in Section 2.1.3. Furthermore, these methods are most often used to accelerate Krylov subspace methods. In that respect, we briefly present the Krylov subspace solvers we have considered for our numerical experiments. Both symmetric positive definite (SPD) problems and unsymmetric problems are encountered that are solved using the conjugate gradient method [60], described in Section 2.2.3, or variants of the GMRES technique [84, 87], described in Section 2.2.2. Because we investigate various variants of the preconditioners and intend to compare their numerical behaviours a particular attention should be paid to the stopping criterion. It should be independent from the preconditioner while ensuring that the computed solutions have similar quality in some metric. Consequently, in Section 2.2.4 we introduce some basic concepts of the backward error analysis that enables us to make this fair comparison.

2.1 Some roots in domain decomposition methods

2.1.1 Introduction

As pointed in [53], the term *domain decomposition* covers a fairly large range of computing techniques for the numerical solution of partial differential equations (PDE's) in time and space. Generally speaking, it refers to the splitting of the computational domain into subdomains with or without overlap. The splitting strategy is generally governed by various constraints/objectives. It might be related to

- some PDE features to, for instance, couple different models such as the Euler and Navier-Stokes equations in computational fluid dynamics;
- some mesh generator/CAD constraints to, for instance, merge a set of grids meshed independently (using possible different mesh generators) into one complex mesh covering an entire simulation domain;
- some parallel computing objective where the overall mesh is split into sub-meshes of approximately equal size to comply with load balancing constraints.

In this chapter we consider this latter situation and focus specifically on the associated domain decomposition techniques for the parallel solution of large linear systems, $Ax = b$, arising from

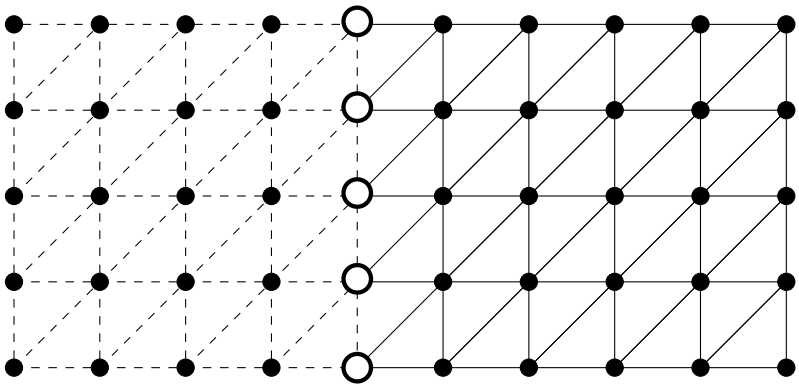


Figure 2.1: Partition of the domain based on a element splitting. Shared vertices are indicated by a circle.

PDE discretizations. Some of the presented techniques can be used as stationary iterative schemes that converge to the linear system solution by properly tuning their governing parameters to ensure that the spectral radius of the iteration matrix is less than one. However, domain decomposition schemes are most effective and require less tuning when they are employed as a preconditioner to accelerate the convergence of a Krylov subspace method [55, 86].

In the next sections an overview of some popular domain decomposition preconditioners is given from an algebraic perspective. We mainly focus on the popular finite element practice of only partially assembling matrices on the interfaces. That is, in a parallel computing environment, each processor is restricted so that it assembles matrix contributions coming only from finite elements owned by the processor. In this case, the domain decomposition techniques correspond to a splitting of the underlying mesh as opposed to splitting the matrix.

Consider a finite element mesh covering the computational domain Ω. For simplicity assume that piecewise linear elements F_k are used such that solution unknowns are associated with mesh vertices. Further, define an associated connectivity graph $G_\Omega = (W_\Omega, E_\Omega)$. The graph vertices $W_\Omega = \{1, \ldots, n_e\}$ correspond to elements in the finite element mesh. The graph edges correspond to element pairs that share at least one mesh vertex. That is, $E_\Omega = \{(i,j)\ s.t.\ F_i \cap F_j \neq \emptyset\}$. Assume that the connectivity graph has been partitioned resulting in N non-overlapping subsets Ω_i^0 whose union is W_Ω. These subsets are referred to as subdomains and are also often referred to as substructures. The Ω_i^0 can be generalized to overlapping subsets of graph vertices. In particular, construct Ω_i^1, the one-overlap decomposition of Ω, by taking Ω_i^0 and including all graph vertices corresponding to immediate neighbours of the vertices in Ω_i^0. By recursively applying this definition, the δ-layer overlap of W_Ω is constructed and the subdomains are denoted Ω_i^δ.

Corresponding to each subdomain Ω_i^0 we define a rectangular extension matrix $\mathcal{R}_i^{0^T}$ whose action extends by zero a vector of values defined at *mesh* vertices associated with the finite elements contained in Ω_i^0. The entries of $\mathcal{R}_i^{0^T}$ are zeros and ones. For simplicity, we omit the *0* superscripts and define $\mathcal{R}_i = \mathcal{R}_i^0$ and $\Omega_i = \Omega_i^0$. Notice that the columns of a given $\mathcal{R}_k$ are orthogonal, but that between the different $\mathcal{R}_i$'s some columns are no longer orthogonal. This is due to the fact that some mesh vertices overlap even though the graph vertices defined by Ω_i are non-overlapping (shared mesh vertices see Figure 2.1). Let Γ_i be the set of all mesh vertices belonging to the interface of Ω_i; that is mesh vertices lying on $\partial\Omega_i \backslash \partial\Omega$. Similarly, let $\mathcal{I}_i$ be the set of all remaining mesh vertices within the subdomain Ω_i (i.e. interior vertices). Considering only the discrete matrix contributions arising from finite elements in Ω_i gives rise to the following local discretization

matrix :

$$\mathcal{A}_i = \begin{pmatrix} \mathcal{A}_{\mathcal{I}_i\mathcal{I}_i} & \mathcal{A}_{\mathcal{I}_i\Gamma_i} \\ A_{\Gamma_i\mathcal{I}_i} & A_{\Gamma_i\Gamma_i} \end{pmatrix}, \tag{2.1}$$

where interior vertices have been ordered first. The matrix $\mathcal{A}_i$ corresponds to the discretization of the PDE on Ω_i with Neumann boundary condition on Γ_i and the one-one block $\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}$ corresponds to the discretization with homogeneous Dirichlet conditions on Γ_i. The completely assembled discretization matrix is obtained by summing the contributions over the substructures/subdomains :

$$A = \sum_{i=1}^{N} \mathcal{R}_i^T \mathcal{A}_i \mathcal{R}_i. \tag{2.2}$$

In a parallel distributed environment each subdomain is assigned to one processor and typically processor i stores $\mathcal{A}_i$. A matrix-vector product is performed in two steps. First a local matrix-vector product involving $\mathcal{A}_i$ is performed followed by a communication step to assemble the results along the interface Γ_i.

For the δ-overlap partition we can define a corresponding restriction operator $\mathcal{R}_i^\delta$ which maps mesh vertices in Ω to the subset of mesh vertices associated with finite elements contained in Ω_i^δ. Corresponding definitions of Γ_i^δ and $\mathcal{I}_i^\delta$ follow naturally as the boundary and interior mesh vertices associated with finite elements in Ω_i^δ. The discretization matrix on Ω_i^δ has a similar structure to the one given by (2.1) and is written as

$$\mathcal{A}_i^\delta = \begin{pmatrix} \mathcal{A}_{\mathcal{I}_i^\delta\mathcal{I}_i^\delta} & \mathcal{A}_{\mathcal{I}_i^\delta\Gamma_i^\delta} \\ \mathcal{A}_{\Gamma_i^\delta\mathcal{I}_i^\delta} & \mathcal{A}_{\Gamma_i^\delta\Gamma_i^\delta} \end{pmatrix}. \tag{2.3}$$

2.1.2 A brief overview on domain decomposition techniques with overlapping domains

The domain decomposition methods based on overlapping subdomains are most often referred to as Schwarz methods due to the pioneering work of Schwarz in 1870 [90]. This work was not intended as a numerical algorithm but was instead developed to show the existence of the elliptic problem solution on a complex geometry formed by overlapping two simple geometries where solutions are known. With the advent of parallel computing this basic technique, known as the alternating Schwarz method, has motivated considerable research activity. In this section, we do not intend to give an exhaustive presentation of all work devoted to Schwarz methods. Only additive variants that are well-suited to straightforward parallel implementation are considered. Within additive variants, computations on all subdomains are performed simultaneously while multiplicative variants require some subdomain calculations to wait for results from other subdomains. The multiplicative versions often have connections to block Gauss-Seidel methods while the additive variants correspond more closely to block Jacobi methods. We do not further pursue this description but refer the interested reader to [94].

2.1.2.1 Additive Schwarz preconditioners

With these notations the additive Schwarz preconditioner is given by

$$\mathcal{M}_{AS}^\delta = \sum_{i=1}^{N} \left(\mathcal{R}_i^{\delta-1}\right)^T \left(\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}^\delta\right)^{-1} \mathcal{R}_i^{\delta-1}. \tag{2.4}$$

Here the δ-overlap is defined in terms of finite element decompositions. The preconditioner and the $\mathcal{R}_i^{\delta-1}$ operators, however, act on mesh vertices corresponding to the sub-meshes associated with the finite element decomposition. The preconditioner is symmetric (or symmetric positive definite) if the original system, A, is symmetric (or symmetric positive definite).

Parallel implementation of this preconditioner requires a factorization of a Dirichlet problem on each processor in the setup phase. Each invocation of the preconditioner requires two neighbour to neighbour communications. The first corresponds to obtaining values within overlapping regions associated with the restriction operator. The second corresponds to summing the results of the backward/forward substitution via the extension operator.

In general, larger overlap usually leads to faster convergence up to a certain point where increasing the overlap does not further improve the convergence rate. Unfortunately, larger overlap implies greater communication and computation requirements.

2.1.2.2 Restricted additive Schwarz preconditioner

A variant of the classical additive Schwarz method is introduced in [23] which avoids one communication step when applying the preconditioner. This variant is referred to as Restricted Additive Schwarz (RAS). This variant does not have a natural counterpart in a mesh partitioning framework that by construction has overlapping sets of vertices. Consequently, the closest mesh partitioning counterpart solves a Dirichlet problem on a large subdomain but considers the solution only within the substructure. That is,

$$\mathcal{M}^{\delta}_{RAS} = \sum_{i=1}^{N} \mathcal{R}_i^T \left(\mathcal{A}^{\delta}_{\mathcal{I}_i\mathcal{I}_i}\right)^{-1} \mathcal{R}_i^{\delta-1}. \tag{2.5}$$

Surprisingly, $\mathcal{M}^{\delta}_{RAS}$ often converges faster than $\mathcal{M}^{\delta}_{AS}$ and only requires half the communication making it frequently superior on parallel distributed computers. Of course it might not be suitable for symmetric positive definite problems as it requires a non-symmetric Krylov solver.

All of the above techniques make use of a matrix inverse (i.e. a direct solver or an exact factorization) of a local Dirichlet matrix. In practice, it is common to replace this with an incomplete factorization [86, 85] or an approximate inverse [13, 14, 28, 57, 63]. This is particularly important for three-dimensional problems where exact factorizations are often expensive in terms of both memory and floating-point operations. While this usually slightly deteriorates the convergence rate, it can lead to a faster method due to the fact that each iteration is less expensive. Finally, we mention that these techniques based on Schwarz variants are available in several large parallel software libraries see for instance [10, 58, 59, 68, 99].

2.1.3 A brief overview on domain decomposition techniques with non-overlapping domains

In this section, methods based on non-overlapping regions are described. Such domain decomposition algorithms are often referred to as sub-structuring schemes. This terminology comes from the structural mechanics discipline where non-overlapping ideas were first developed. In this early work the primary focus was on direct solvers. Associating one frontal matrix with each subdomain allows for coarse grain multiple front direct solvers [36]. Motivated by parallel distributed computing and the potential for coarse grain parallelism, considerable research activity developed around iterative domain decomposition schemes. A very large number of methods have been proposed and we cannot cover all of them. Therefore, the main highlights are surveyed.

The governing idea behind sub-structuring or Schur complement methods is to split the unknowns in two subsets. This induces the following block reordered linear system :

$$\begin{pmatrix} \mathcal{A}_{II} & \mathcal{A}_{I\Gamma} \\ \mathcal{A}_{\Gamma I} & \mathcal{A}_{\Gamma\Gamma} \end{pmatrix} \begin{pmatrix} x_I \\ x_\Gamma \end{pmatrix} = \begin{pmatrix} b_I \\ b_\Gamma \end{pmatrix}, \tag{2.6}$$

where x_Γ contains all unknowns associated with subdomain interfaces and x_I contains the remaining unknowns associated with subdomain interiors. The matrix $\mathcal{A}_{II}$ is block diagonal where each block corresponds to a subdomain interior. Eliminating x_I from the second block row of

equation (2.6) leads to the reduced system

$$\mathcal{S}x_\Gamma = b_\Gamma - \mathcal{A}_{\Gamma I}\mathcal{A}_{II}^{-1}b_I, \text{ where } \mathcal{S} = \mathcal{A}_{\Gamma\Gamma} - \mathcal{A}_{\Gamma I}\mathcal{A}_{II}^{-1}\mathcal{A}_{I\Gamma} \tag{2.7}$$

and $\mathcal{S}$ is referred to as the *Schur complement matrix*. This reformulation leads to a general strategy for solving (2.6). Specifically, an iterative method can be applied to (2.7). Once x_Γ is determined, x_I can be computed with one additional solve on the subdomain interiors. Further, when $\mathcal{A}$ is symmetric positive definite (SPD), the matrix $\mathcal{S}$ inherits this property and so a conjugate gradient method can be employed.

Not surprisingly, the structural analysis finite element community has been heavily involved with these techniques. Not only is their definition fairly natural in a finite element framework but their implementation can preserve data structures and concepts already present in large engineering software packages.

Let Γ denote the entire interface defined by $\Gamma = \cup\,\Gamma_i$ where $\Gamma_i = \partial\Omega_i \backslash \partial\Omega$. As interior unknowns are no longer considered, new restriction operators must be defined as follows. Let $\mathcal{R}_{\Gamma_i} : \Gamma \rightarrow \Gamma_i$ be the canonical point-wise restriction which maps full vectors defined on Γ into vectors defined on Γ_i. Analogous to (2.2), the Schur complement matrix (2.7) can be written as the sum of elementary matrices

$$\mathcal{S} = \sum_{i=1}^{N} \mathcal{R}_{\Gamma_i}^T \mathcal{S}_i \mathcal{R}_{\Gamma_i}, \tag{2.8}$$

where

$$\mathcal{S}_i = \mathcal{A}_{\Gamma_i\Gamma_i} - \mathcal{A}_{\Gamma_i\mathcal{I}_i}\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}^{-1}\mathcal{A}_{\mathcal{I}_i\Gamma_i} \tag{2.9}$$

is a local Schur complement and is defined in terms of sub-matrices from the local Neumann matrix $\mathcal{A}_i$ given by (2.1). Notice that this form of the Schur complement has only one layer of interface unknowns between subdomains and allows for a straight-forward parallel implementation.

While the Schur complement system is significantly better conditioned than the original matrix A, it is important to consider further preconditioning when employing a Krylov method. It is well-known, for example, that $\kappa(A) = \mathcal{O}(h^{-2})$ when A corresponds to a standard discretization (e.g. piecewise linear finite elements) of the Laplace operator on a mesh with spacing h between the grid points. Using two non-overlapping subdomains effectively reduces the condition number of the Schur complement matrix to $\kappa(S) = \mathcal{O}(h^{-1})$. While improved, preconditioning can significantly lower this condition number further.

2.1.3.1 The Neumann-Dirichlet preconditioner

When a symmetric constant coefficient problem is sub-divided into two non-overlapping domains such that the subdomains are exact mirror images, it follows that the Schur complement contribution from both the left and right domains is identical. That is, $\mathcal{S}_1 = \mathcal{S}_2$. Consequently, the inverse of either $\mathcal{S}_1$ or $\mathcal{S}_2$ are ideal preconditioners as the preconditioned linear system is well-conditioned, e.g. $\mathcal{S}\mathcal{S}_1^{-1} = 2I$. A factorization can be applied to the local Neumann problem (2.1) on Ω_1 :

$$\mathcal{A}_1 = \begin{pmatrix} Id_{\mathcal{I}_1} & 0 \\ \mathcal{A}_{\mathcal{I}_1\Gamma_1}A_{\mathcal{I}_1\mathcal{I}_1}^{-1} & Id_{\Gamma_1} \end{pmatrix} \begin{pmatrix} A_{\mathcal{I}_1\mathcal{I}_1} & 0 \\ 0 & \mathcal{S}_1 \end{pmatrix} \begin{pmatrix} Id_{\mathcal{I}_1} & A_{\mathcal{I}_1\mathcal{I}_1}A_{\Gamma_1\mathcal{I}_1} \\ 0 & Id_{\Gamma_1} \end{pmatrix}$$

to obtain

$$\mathcal{S}_1^{-1} = \begin{pmatrix} 0 & Id_{\Gamma_1} \end{pmatrix} (\mathcal{A}_1)^{-1} \begin{pmatrix} 0 \\ Id_{\Gamma_1} \end{pmatrix}.$$

In general, most problems will not have mirror image subdomains and so $\mathcal{S}_1 \neq \mathcal{S}_2$. However, if the underlying system within the two subdomains is similar, the inverse of $\mathcal{S}_1$ should make an excellent preconditioner. The corresponding linear system is

$$\left(I + \mathcal{S}_1^{-1}\mathcal{S}_2\right) x_{\Gamma_1} = \left(\mathcal{S}_1\right)^{-1} b_{\Gamma_1}$$

so that each Krylov iteration solves a Dirichlet problem on Ω_2 (to apply $\mathcal{S}_2$) followed by a Neumann problem on Ω_1 to invert $\mathcal{S}_1$. The Neumann-Dirichlet preconditioner was introduced in [16].

Generalization of the Neumann-Dirichlet preconditioner to multiple domains can be done easily when a red-black coloring of subdomains is possible such that subdomains of the same color do not share an interface. In this case, the preconditioner is just the sum of the inverses corresponding to the black subdomains:

$$S = \sum_{i \in B} \mathcal{R}_{\Gamma_i}^T (\mathcal{S}_i)^{-1} \mathcal{R}_{\Gamma_i} \tag{2.10}$$

where B corresponds to the set of all black subdomains.

2.1.3.2 The Neumann-Neumann preconditioner

Similar to the Neumann-Dirichlet method, the Neumann-Neumann preconditioner implicitly relies on the similarity of the Schur complement contribution from different subdomains. In the Neumann-Neumann approach the preconditioner is simply the weighted sum of the inverse of the $\mathcal{S}_i$. In the two mirror image subdomains case,

$$S^{-1} = \frac{1}{2}\left(\mathcal{S}_1^{-1} + \mathcal{S}_2^{-1}\right).$$

This motivates using the following preconditioner with multiple subdomains :

$$M_{NN} = \sum_{i=1}^{N} \mathcal{R}_{\Gamma_i}^T D_i \mathcal{S}_i^{-1} D_i \mathcal{R}_{\Gamma_i} \tag{2.11}$$

where the D_i are diagonal weighting matrices corresponding to a partition of unity. That is,

$$\sum_{i=1}^{N} \mathcal{R}_{\Gamma_i}^T D_i \mathcal{R}_{\Gamma_i} = Id_{\Gamma}.$$

The simplest choice for D_i is the diagonal matrix with entries equal to the inverse of the number of subdomains to which an unknown belongs. The Neumann-Neumann preconditioner was first discussed in [19] and further studied in [98] where different choices for weight matrices are discussed. It should be noted that the matrices $\mathcal{S}_i$ can be singular for internal subdomains because they correspond to pure Neumann problems. The Moore-Penrose pseudo-inverse is often used for the inverse local Schur complements in (2.11) but other choices are possible such as inverting $\mathcal{A}_i + \epsilon I$ where ϵ is a small shift.

The Neumann-Neumann preconditioner is very attractive from a a parallel implementation point of view. In particular, all interface unknowns are treated similarly and no distinction is required to differentiate between unknowns on faces, edges, or cross points as it might be the case in other approaches.

2.2 Some background on Krylov subspace methods

2.2.1 Introduction

Among the possible iterative techniques for solving a linear system of equations the approaches based on Krylov subspaces are very efficient and widely used. Let A be a square nonsingular $n \times n$ matrix, and b be a vector of length n, defining the linear system

$$Ax = b \tag{2.12}$$

to be solved. Let $x_0 \in \mathbb{C}^n$ be an initial guess for this linear system and $r_0 = b - Ax_0$ be its corresponding residual.

The Krylov subspace linear solvers construct an approximation of the solution in the affine space $x_0 + \mathcal{K}_m$, where $\mathcal{K}_m$ is the Krylov space of dimension m defined by

$$\mathcal{K}_m = \text{span}\left\{r_0, Ar_0, \ldots, A^{m-1}r_0\right\}.$$

The various Krylov solvers differ in the constraints or optimality conditions associated with the computed solution. In the sequel, we describe in some detail the GMRES method [87] where the solution selected in the Krylov space corresponds to the vector that minimizes the Euclidean norm of the residual. This method is well-suited for unsymmetric problems. We also briefly present the oldest Krylov techniques that is the Conjugate Gradient method, where the solution in the Krylov space is chosen so that the associated residual is orthogonal to the space.

Many other techniques exist that we will not describe in this section; we rather refer the reader to the books [55, 86].

In many cases, such methods converge slowly, or even diverge. The convergence of iterative methods may be improved by transforming the system (2.12) into another system which is easier to solve. A preconditioner is a matrix that realizes such a transformation. If M is a non-singular matrix which approximates A^{-1}, then the transformed linear system:

$$MAx = Mb, \tag{2.13}$$

might be solved faster. The system (2.13) is preconditioned from the left, but one can also precondition from the right:

$$AMt = b. \tag{2.14}$$

Once the solution t is obtained, the solution of the system (2.12) is recovered by $x = Mt$.

2.2.2 The unsymmetric problems

The Generalized Minimum RESidual (GMRES) method was proposed by Saad and Schultz in 1986 [87] for the solution of large non hermitian linear systems.

For the sake of generality, we describe this method for linear systems whose entries are complex, everything also extends to real arithmetic.

Let $x_0 \in \mathbb{C}^n$ be an initial guess for the linear system (2.12) and $r_0 = b - Ax_0$ be its corresponding residual. At step k, the GMRES algorithm builds an approximation of the solution of (2.12) under the form

$$x_k = x_0 + V_k y_k, \tag{2.15}$$

where $y_k \in \mathbb{C}^k$ and $V_k = [v_1, \cdots, v_k]$ is an orthonormal basis for the Krylov space of dimension k defined by

$$\mathcal{K}(A, r_0, k) = \text{span}\left\{r_0, Ar_0, \ldots, A^{k-1}r_0\right\}.$$

The vector y_k is determined so that the 2–norm of the residual $r_k = b - Ax_k$ is minimized over $x_0 + \mathcal{K}(A, r_0, k)$. The basis V_k for the Krylov subspace $\mathcal{K}(A, r_0, k)$ is obtained via the well-known Arnoldi process [7]. The orthogonal projection of A onto $\mathcal{K}(A, r_0, k)$ results in an upper Hessenberg matrix $H_k = V_k^H A V_k$ of order k. The Arnoldi process satisfies the relationship

$$AV_k = V_k H_k + h_{k+1,k} v_{k+1} e_k^T, \tag{2.16}$$

where e_k is the k^{th} canonical basis vector. Equation (2.16) can be rewritten in a matrix form as

$$AV_k = V_{k+1}\bar{H}_k,$$

where

$$\bar{H}_k = \begin{bmatrix} H_k \\ 0 \cdots 0 \; h_{k+1,k} \end{bmatrix}$$

is an $(k+1) \times k$ matrix.

Let $v_1 = r_0/\beta$ where $\beta = \|r_0\|_2$. The residual r_k associated with the approximate solution x_k defined by (2.15) satisfies

$$\begin{aligned} r_k &= b - Ax_k = b - A(x_0 + V_k y_k) \\ &= r_0 - AV_k y_k = r_0 - V_{k+1}\bar{H}_k y_k \\ &= \beta v_1 - V_{k+1}\bar{H}_k y_k \\ &= V_{k+1}(\beta e_1 - \bar{H}_k y_k). \end{aligned}$$

Because V_{k+1} is a matrix with orthonormal columns, the residual norm $\|r_k\|_2 = \|\beta e_1 - \bar{H}_k y_k\|_2$ is minimized when y_k solves the linear least-squares problem

$$\min_{y \in \mathbb{C}^k} \|\beta e_1 - \bar{H}_k y\|_2. \tag{2.17}$$

We denote by y_k the solution of (2.17). Therefore, $x_k = x_0 + V_k y_k$ is an approximate solution of (2.12) for which the residual is minimized over $x_0 + \mathcal{K}(A, r_0, k)$. The GMRES method owes its name to this minimization property that is its key feature as it ensures the decrease of the residual norm associated with the sequence of iterates.

In exact arithmetic, GMRES converges in at most n steps. However, in practice, n can be very large and the storage of the orthonormal basis V_k may become prohibitive. On top of that, the orthogonalization of v_k with respect to the previous vectors $v_1, \cdots, v_{k-1}$ requires $4nk$ flops, for large k, the computational cost of the orthogonalization scheme may become very expensive. The restarted GMRES method is designed to cope with these two drawbacks. Given a fixed m, the restarted GMRES method computes a sequence of approximate solutions x_k until x_k is acceptable or $k = m$. If the solution was not found, then a new starting vector is chosen on which GMRES is applied again. Often, GMRES is restarted from the last computed approximation, i.e., $x_0 = x_m$ to comply with the monotonicity property of the norm decrease even when restarting. The process is iterated until a good enough approximation is found. We denote by GMRES(m) the restarted GMRES algorithm for a projection size of at most m. A detailed description of the restarted GMRES with right preconditioner and modified Gram-Schmidt algorithm as orthogonalization scheme is given in Algorithm 1.

We now briefly describe GMRES with right preconditioner and its flexible variant that should be preferred when the preconditioner varies from on step to the next. Let M be a square nonsingular $n \times n$ complex matrix, we define the right preconditioned linear system

$$AMt = b, \tag{2.18}$$

where $x = Mt$ is the solution of the unpreconditioned linear system. Let $t_0 \in \mathbb{C}^n$ be an initial guess for this linear system and $r_0 = b - AMt_0$ be its corresponding residual.

The GMRES algorithm builds an approximation of the solution of (2.18) of the form

$$t_k = t_0 + V_k y_k \tag{2.19}$$

where the columns of V_k form an orthonormal basis for the Krylov space of dimension m defined by

$$\mathcal{K}_k = \text{span}\left\{r_0, AMr_0, \ldots, (AM)^{k-1} r_0\right\},$$

and where y_k belongs to $\mathbb{C}^k$. The vector y_k is determined so that the 2-norm of the residual $r_k = b - AMt_k$ is minimal over $\mathcal{K}_k$.

The basis V_k for the Krylov subspace $\mathcal{K}_k$ is obtained via the well-known Arnoldi process. The orthogonal projection of A onto $\mathcal{K}_k$ results in an upper Hessenberg matrix $H_k = V_k^H A V_k$ of order k. The Arnoldi process satisfies the relationship

$$A[Mv_1, \cdots, Mv_k] = AMV_k = V_k H_k + h_{k+1,k} v_{k+1} e_k^H, \tag{2.20}$$

where e_k is the k^{th} canonical basis vector. Equation (2.20) can be rewritten as

$$AMV_k = V_{k+1}\bar{H}_k$$

where

$$\bar{H}_k = \begin{bmatrix} H_k \\ 0 \cdots 0 \; h_{k+1,k} \end{bmatrix}$$

is an $(k+1) \times k$ matrix.

Let $v_1 = r_0/\beta$ where $\beta = \|r_0\|_2$. The residual r_k associated with the approximate solution defined by Equation (2.19) verifies

$$\begin{aligned} r_k &= b - AMt_k = b - AM(t_0 + V_k y_k) \\ &= r_0 - AMV_k y_k = r_0 - V_{k+1}\bar{H}_k y_k \\ &= \beta v_1 - V_{k+1}\bar{H}_k y_k \\ &= V_{k+1}(\beta e_1 - \bar{H}_k y_k). \end{aligned} \tag{2.21}$$

Since V_{k+1} is a matrix with orthonormal columns, the residual norm $\|r_k\|_2 = \left\|\beta e_1 - \bar{H}_k y_k\right\|_2$ is minimal when y_k solves the linear least-squares problem (2.17). We will denote by y_k the solution of (2.17). Therefore, $t_k = t_0 + V_k y_k$ is an approximate solution of (2.18) for which the residual is minimal over $\mathcal{K}_k$. We depict in Algorithm 1 the sketch of the Modified Gram-Schmidt (MGS) variant of the GMRES method with right preconditioner.

Algorithm 1 Right preconditioned GMRES

1: Choose a convergence threshold ε
2: Choose an initial guess t_0
3: $r_0 = b - AMt_0 = b$; $\beta = \|r_0\|$
4: $v_1 = r_0/\|r_0\|$;
5: **for** $k = 1, 2, \ldots$ **do**
6: $w = AMv_k$;
7: **for** $i = 1$ `to` k **do**
8: $h_{i,k} = v_i^H w$
9: $w = w - h_{i,k} v_i$
10: **end for**
11: $h_{k+1,k} = \|w\|$
12: $v_{k+1} = w/h_{k+1,k}$
13: Solve the least-squares problem $\min \|\beta e_1 - \bar{H}_k y\|$ for y
14: Exit if convergence is detected
15: **end for**
16: Set $x_m = M(t_0 + V_m y)$

If the preconditioner involved at step 6 in Algorithm 1 varies at each step, we can still write an equality similar to (2.20) as:

$$\begin{aligned} A[M_1 v_1, \cdots, M_k v_k] &= A[z_1, \cdots, z_k] \\ &= AZ_k \\ &= V_k H_k + h_{k+1,k} v_{k+1} e_k^H \\ &= V_k \bar{H}_k, \end{aligned}$$

which enables us to get a relation similar to (2.21). Using $x_k = x_0 + Z_k y_k$ we have

$$\begin{aligned} r_k &= b - Ax_k = b - A(x_0 + Z_k y_k) \\ &= r_0 - AZ_k y_k = r_0 - V_{k+1}\bar{H}_k y_k \\ &= \beta v_1 - V_{k+1}\bar{H}_k y_k \\ &= V_{k+1}(\beta e_1 - \bar{H}_k y_k), \end{aligned}$$

where y_k is the solution of a least-squares problem similar to (2.17). Because this GMRES variant allows for flexible preconditioners it is referred to as Flexible GMRES. From an implementation point of view the main difference between right preconditioned GMRES and FGMRES is the memory requirement. In that latter algorithm, both V_k and Z_k need to be stored. We remind that only happy breakdowns might occur in GMRES (i.e., at step 11 of Algorithm 1 if $h_{k+1,k}$ is zero, the algorithm would breakdown but it does not care because it also means that it has found the solution [87]). This is no longer true for FGMRES that can break at step 12 before it has computed the solution. We describe the MGS variant of this method in Algorithm 2 and refer to [84] for a complete description of the convergence theory.

Algorithm 2 Flexible GMRES

1: Choose a convergence threshold ε
2: Choose an initial guess x_0
3: $r_0 = b - Ax_0 = b$; $\beta = \|r_0\|$
4: $v_1 = r_0/\|r_0\|$;
5: **for** $k = 1, 2, \ldots$ **do**
6: $z_k = M_k v_k$; % M_k is the preconditioner used at step k
7: $w = Az_k$;
8: **for** $i = 1$ `to` k **do**
9: $h_{i,k} = v_i^H w$
10: $w = w - h_{i,k} v_i$
11: **end for**
12: $h_{k+1k} = \|w\|$
13: $v_{k+1} = w/h_{k+1,k}$
14: Solve the least-squares problem $\min \|\beta e_1 - \bar{H}_k y\|$ for y
15: Exit if convergence is detected
16: **end for**
17: Set $x_m = x_0 + Z_m y$

There are numerical situations where the preconditioner varies from one step to the next of the construction of the space. In that framework, the FGMRES (Flexible Generalized Minimum Residual) method [84] is among the most widely used Krylov solvers for the iterative solution of general large linear systems when variable preconditioning is considered.

Implementations of the GMRES and FGMRES algorithms for real and complex, single and double precision arithmetics suitable for serial, shared memory and distributed memory computers is available from the Web at the following URL:

http://www.cerfacs.fr/algor/Softs/

The implementation is based on the reverse communication mechanism for the matrix-vector product, the preconditioning and the dot product computations. We used these packages in our experiments.

2.2.3 The symmetric positive definite problems

The Conjugate Gradient method was proposed in different versions in the early 50s in separate contributions by Lanczos [65] and Hestenes and Stiefel [60]. It becomes the method of choice for

the solution of large sparse hermitian positive definite linear system and is the starting point of the extensive work on the Krylov methods [88].

Let $A = A^H$ (where A^H denotes the conjugate transpose of A) be a square nonsingular $n \times n$ complex hermitian positive definite matrix, and b be a complex vector of length n, defining the linear system

$$Ax = b \tag{2.22}$$

to be solved.

Let $x_0 \in \mathbb{C}^n$ be an initial guess for this linear system, $r_0 = b - Ax_0$ be its corresponding residual and M^{-1} be the preconditioner. The preconditioned conjugate gradient algorithm is classically described as depicted in Algorithm 3

Algorithm 3 Preconditioned Conjugate Gradient

$k = 0$
$r_0 = b - Ax_0$
for $k = 0, 1, 2, \ldots$ **do**
  Solve $Mz_k = r_k$
  if $k = 1$ **then**
    $p_1 = z_0$
  else
    $\beta_{k-1} = z_{(k-1)}^H r_{k-1} / z_{k-2}^H r_{k-2}$
    $p_k = z_{k-1} + \beta_{k-1} p_{k-1}$
  end if
  $q_k = Ap_k$
  $\alpha_k = z_{k-1}^H r_{k-1} / p_k^H q_k$
  $x_k = x_{k-1} + \alpha_k p_k$
  $r_k = r_{k-1} - \alpha_k q_k$
  Exit if convergence is detected
end for

The conjugate gradient algorithm constructs the solution that makes its associated residual orthogonal to the Krylov space. A consequence of this geometric property is that it is also the minimum error solution in A-norm over the Krylov space $\mathcal{K}_k = \text{span}\left\{r_0, Ar_0, \ldots, A^{k-1}r_0\right\}$. It exists a rich literature dedicated to this method: for more details we, non-exhaustively, refer to [9, 54, 72, 86] and the references therein.

We simply mention that the preconditioned conjugate gradient method can be written as depicted in Algorithm 3 that enables us to still have short recurrence on the unpreconditioned solution.

2.2.4 Stopping criterion: a central component

The backward error analysis, introduced by Givens and Wilkinson [101], is a powerful concept for analyzing the quality of an approximate solution:

1. it is independent of the details of round-off propagation: the errors introduced during the computation are interpreted in terms of perturbations of the initial data, and the computed solution is considered as exact for the perturbed problem;

2. because round-off errors are seen as data perturbations, they can be compared with errors due to numerical approximations (consistency of numerical schemes) or to physical measurements (uncertainties on data coming from experiments for instance).

The backward error defined by (2.23) measures the distance between the data of the initial problem and those of a perturbed problem. Dealing with such a distance both requires to choose the data

that are perturbed and a norm to quantify the perturbations. For the first choice, the matrix and the right-hand side of the linear systems are natural candidates. In the context of linear systems, classical choices are the normwise and the componentwise perturbations [27, 61]. These choices lead to explicit formulas for the backward error (often a normalized residual) which is then easily evaluated. For iterative methods, it is generally admitted that the normwise model of perturbation is appropriate [11].

Let x_k be an approximation to the solution $x = A^{-1}b$. The quantity

$$\begin{aligned} \eta_{A,b}(x_k) &= \min_{\Delta A, \Delta b} \{\tau > 0 : \|\Delta A\| \leq \tau \|A\|, \ \|\Delta b\| \leq \tau \|b\| \\ &\qquad \text{and } (A + \Delta A)x_k = b + \Delta b\} \\ &= \frac{\|Ax_k - b\|}{\|A\|\|x_k\| + \|b\|}, \end{aligned} \tag{2.23}$$

is called the *normwise backward error* associated with x_k. It measures the norm of the smallest perturbations ΔA on A and Δb on b such that x_k is the exact solution of $(A+\Delta A)x_k = b+\Delta b$. The best one can require from an algorithm is a backward error of the order of the machine precision. In practice, the approximation of the solution is acceptable when its backward error is lower than the uncertainty of the data. Therefore, there is no gain in iterating after the backward error has reached machine precision (or data accuracy).

In many situations it might be difficult to compute (even approximatively) $\|A\|$. Consequently, another backward error criterion can be considered that is simpler to evaluate and implement in practice. It is defined by

$$\begin{aligned} \eta_b(x_k) &= \min_{\Delta b} \{\tau > 0 : \|\Delta b\| \leq \tau \|b\| \text{ and } Ax_k = b + \Delta b\} \\ &= \frac{\|Ax_k - b\|}{\|b\|}. \end{aligned} \tag{2.24}$$

This latter criterion measures the norm of the smallest perturbations Δb on b (assuming that they are no perturbations on A) such that x_k is the exact solution of $Ax_k = b + \Delta b$. Clearly we have $\eta_{A,b}(x_k) < \eta_b(x_k)$. It has been shown [34, 77] that GMRES with robust orthogonalization schemes is backward stable with respect to a backward error similar to (2.23) with a different choice for the norms.

We mention that $\eta_{A,b}$ and η_b are recommended in [11] when the concern related to the stopping criterion is discussed; the stopping criteria of the Krylov solvers we used for our numerical experiments are based on them.

For preconditioned GMRES, these criteria read differently depending on the location of the preconditioners. In that context, using a preconditioner means running GMRES on the linear systems:

1. $MAx = Mb$ for left preconditioning,
2. $AMy = b$ for right preconditioning,
3. $M_2AM_1y = M_2b$ for split preconditioning.

Consequently, the backward stability property holds for those preconditioned systems where the corresponding stopping criteria are depicted in Table 2.1. In particular, it can be seen that for all but the right preconditioning and $\eta_{M,b}$, the backward error depends on the preconditioner. For right preconditioning, the backward error η_b is the same for the preconditioned and unpreconditioned system because $\|AMt - b\| = \|Ax - b\|$. This is the main reason why for all our numerical experiments with GMRES we selected right preconditioning. A stopping criterion based on η_b enables a fair comparison among the tested approaches as the iterations are stopped once the approximations have all the same quality with respect to this backward error criterion.

	Left precond.	Right precond.	Split precond.
$\eta_{M,A,b}$	$\frac{\|MAx-Mb\|}{\|MA\|\|x\|+\|Mb\|}$	$\frac{\|AMt-b\|}{\|AM\|\|x\|+\|b\|}$	$\frac{\|M_2AM_1t-M_2b\|}{\|M_2AM_1\|\|t\|+\|M_2b\|}$
$\eta_{M,b}$	$\frac{\|MAx-Mb\|}{\|Mb\|}$	$\frac{\|AMt-b\|}{\|b\|}$	$\frac{\|M_2AM_1t-M_2b\|}{\|M_2b\|}$

Table 2.1: Backward error associated with preconditioned linear system in GMRES.

Chapter 3

An additive Schwarz preconditioner for Schur complement

3.1 Introduction

The design of preconditioners is essential for the successful use of iterative methods. Consequently research on preconditioners has moved to center stage in recent years. Preconditioning is a way of transforming the original linear system into another one having the same solution but better conditioning and thus easier to solve.

This chapter is mainly devoted to addressing the presentation of the algebraic additive Schwarz preconditioners studied in this thesis. In Section 3.2, we present the algebraic description of the additive Schwarz preconditioner. The main lines of the Schur complement are presented, and the main aspects of parallel preconditioning are introduced. We propose and study different variants of the preconditioner, based on either sparse techniques in Section 3.3, or mixed precision arithmetic in Section 3.4. We consider in Section 3.5, two-level preconditioner based on an algebraic construction of a coarse space component. Finally, we describe in Section 3.6 a diagonal scaling technique that is suitable for a parallel implementation.

3.2 Algebraic description

In this section we introduce the general form of the preconditioner considered in this work. We use the notation introduced in Section 2.1. For the sake of simplicity, we describe the basis of our local preconditioner in two dimensions as its generalization to three dimensions is straightforward. In Figure 3.1, we depict an internal subdomain Ω_i with its edge interfaces E_m, E_g, E_k, and E_ℓ

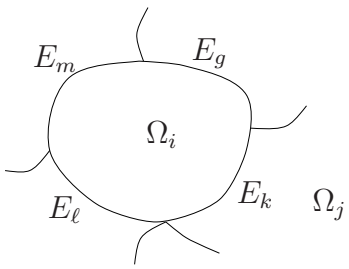


Figure 3.1: An internal subdomain.

that define $\Gamma_i = \partial\Omega_i \backslash \partial\Omega$. Let $\mathcal{R}_{\Gamma_i} : \Gamma \to \Gamma_i$ be the canonical pointwise restriction that maps full vectors defined on Γ into vectors defined on Γ_i, and let $\mathcal{R}_{\Gamma_i}^T : \Gamma_i \to \Gamma$ be its transpose. For a stiffness matrix $\mathcal{A}$ arising from a finite element discretization, the Schur complement matrix (2.6) can also be written

$$\mathcal{S} = \sum_{i=1}^{N} \mathcal{R}_{\Gamma_i}^T \mathcal{S}_i \mathcal{R}_{\Gamma_i},$$

where

$$\mathcal{S}_i = \mathcal{A}_{\Gamma_i\Gamma_i} - \mathcal{A}_{\Gamma_i\mathcal{I}_i}\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}^{-1}\mathcal{A}_{\mathcal{I}_i\Gamma_i} \tag{3.1}$$

is referred to as the local Schur complement associated with the subdomain Ω_i. The matrix $\mathcal{S}_i$ involves submatrices from the local stiffness matrix $\mathcal{A}_i$, defined by

$$\mathcal{A}_i = \begin{pmatrix} \mathcal{A}_{\mathcal{I}_i\mathcal{I}_i} & \mathcal{A}_{\mathcal{I}_i\Gamma_i} \\ \mathcal{A}_{\Gamma_i\mathcal{I}_i} & A_{\Gamma_i\Gamma_i} \end{pmatrix}. \tag{3.2}$$

The matrix $\mathcal{A}_i$ corresponds to the discretization of the PDE on the subdomain Ω_i with Neumann boundary condition on Γ_i and $\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}$ corresponds to the discretization on the subdomain Ω_i with homogeneous Dirichlet boundary conditions on Γ_i. The local Schur complement matrix, associated with the subdomain Ω_i depicted in Figure 3.1, is dense and has the following 4×4 block structure:

$$\mathcal{S}_i = \begin{pmatrix} \mathcal{S}_{mm}^{(i)} & \mathcal{S}_{mg} & \mathcal{S}_{mk} & \mathcal{S}_{m\ell} \\ \mathcal{S}_{gm} & \mathcal{S}_{gg}^{(i)} & \mathcal{S}_{gk} & \mathcal{S}_{g\ell} \\ \mathcal{S}_{km} & \mathcal{S}_{kg} & \mathcal{S}_{kk}^{(i)} & \mathcal{S}_{k\ell} \\ \mathcal{S}_{\ell m} & \mathcal{S}_{\ell g} & \mathcal{S}_{\ell k} & \mathcal{S}_{\ell\ell}^{(i)} \end{pmatrix}, \tag{3.3}$$

where each block accounts for the interactions between the degrees of freedom of the edges of the interface $\partial\Omega_i$.

The preconditioner presented below was originally proposed in [25] in two dimensions and successfully applied to large two dimensional semiconductor device modeling in [52]. To describe this preconditioner we define the local assembled Schur complement, $\bar{\mathcal{S}}_i = \mathcal{R}_{\Gamma_i} S \mathcal{R}_{\Gamma_i}^T$, that corresponds to the restriction of the Schur complement to the interface Γ_i. This local assembled preconditioner can be built from the local Schur complements $\mathcal{S}_i$ by assembling their diagonal blocks thanks to a few neighbour to neighbour communications. For instance, the diagonal blocks of the complete matrix $\mathcal{S}$ associated with the edge interface E_k, depicted in Figure 3.1, is $\mathcal{S}_{kk} = \mathcal{S}_{kk}^{(i)} + \mathcal{S}_{kk}^{(j)}$. Assembling each diagonal block of the local Schur complement matrices, we obtain the local assembled Schur complement, that is:

$$\bar{\mathcal{S}}_i = \begin{pmatrix} \mathcal{S}_{mm} & \mathcal{S}_{mg} & \mathcal{S}_{mk} & \mathcal{S}_{m\ell} \\ \mathcal{S}_{gm} & \mathcal{S}_{gg} & \mathcal{S}_{gk} & \mathcal{S}_{g\ell} \\ \mathcal{S}_{km} & \mathcal{S}_{kg} & \mathcal{S}_{kk} & \mathcal{S}_{k\ell} \\ \mathcal{S}_{\ell m} & \mathcal{S}_{\ell g} & \mathcal{S}_{\ell k} & \mathcal{S}_{\ell\ell} \end{pmatrix}.$$

With these notations the preconditioner reads

$$M_d = \sum_{i=1}^{N} \mathcal{R}_{\Gamma_i}^T \bar{\mathcal{S}}_i^{-1} \mathcal{R}_{\Gamma_i}. \tag{3.4}$$

If we considered the unit square partitioned into horizontal strips (1D decomposition), the resulting Schur complement matrix has a block tridiagonal structure as depicted in (3.5). For that particular structure of $\mathcal{S}$ the submatrices in boxes correspond to the $\bar{\mathcal{S}}_i$. Such diagonal blocks, that overlap, are similar to the classical block overlap of the Schwarz method when writing in a matrix form for 1D decomposition. Similar ideas have been developed in a pure algebraic context in earlier papers [22, 81] for the solution of general sparse linear systems. Because of this

link, the preconditioner defined by (3.4) is referred to as algebraic additive Schwarz for the Schur complement,

$$\mathcal{S} = \begin{pmatrix} \ddots & & & \\ & \boxed{\begin{matrix} S_{k,k} & S_{k,k+1} \\ S_{k+1,k} & S_{k+1,k+1} \end{matrix}} & \boxed{\begin{matrix} S_{k+1,k+2} \\ S_{k+2,k+2} \end{matrix}} & \\ & \boxed{\begin{matrix} S_{k+1,k} & S_{k+1,k+1} \end{matrix}} & & \\ & & & \ddots \end{pmatrix}. \tag{3.5}$$

One advantage of using the assembled local Schur complements instead of the local Schur complements (like in the Neumann-Neumann [19, 29] introduced in Section 2.1.3.2) is that in the SPD case the assembled Schur complements cannot be singular (as $\mathcal{S}$ is not singular).

3.3 Sparse algebraic Additive Schwarz preconditioner

The construction of the proposed local preconditioners can be computationally expensive because the dense matrices $\bar{\mathcal{S}}_i$ should be factorized. We intend to reduce the storage and the computational cost to form and apply the preconditioner by using sparse approximation of the M_d obtained by dropping the smallest entries. In three dimensional problems the size of the dense local Schur matrices can be large. Consequently it is computationally expensive to factorize and solve linear systems with them. One possible alternative to get a cheaper preconditioner is to consider a sparse approximation for $\bar{\mathcal{S}}_i$ in (3.4), which may result in a saving of memory to store the preconditioner and saving of computation to factorize and apply it. This approximation $\hat{\mathcal{S}}_i$ can be constructed by dropping the elements of $\bar{S}_{(i)}$ that are smaller than a given threshold. More precisely, the following symmetric dropping formula can be applied:

$$\hat{s}_{\ell j} = \begin{cases} 0, & \text{if } |\bar{s}_{\ell j}| \leq \xi(|\bar{s}_{\ell\ell}| + |\bar{s}_{jj}|), \\ \bar{s}_{\ell j}, & \text{otherwise,} \end{cases}, \tag{3.6}$$

where $\bar{s}_{\ell j}$ denotes the entries of $\bar{\mathcal{S}}_i$.

The resulting preconditioner based on these sparse approximations reads

$$M_{sp} = \sum_{i=1}^{N} \mathcal{R}_{\Gamma_i}^T \hat{S}_i^{-1} \mathcal{R}_{\Gamma_i}.$$

We notice that such a dropping strategy is suited for symmetric matrices. In particular it is shown in [25] that the resulting preconditioner is SPD for some classes of matrices. For unsymmetric problems, unsymmetric dropping policies could be considered. Such a study is beyond the work presented in this manuscript where only the symmetric dropping defined by (3.6) is considered in the numerical experiments.

To illustrate the time saving in the set-up of the preconditioner in its sparse version versus its dense counterpart, we display in Figure 3.2 the computing time for various values of the dropping threshold when the size of the local Schur is varied. For this illustration we consider a SPD case, where the Cholesky factorization is performed using the Lapack [4] kernel Dpotrf for the dense part and the sparse direct solver Mumps [2, 3] for the sparse counterpart. We mention that similar behaviour and time saving can be observed on unsymmetric matrices.

3.4 Mixed precision Additive Schwarz preconditioner

Motivated by accuracy reasons, many large-scale scientific applications and industrial numerical simulation codes are fully implemented in 64-bit floating-point arithmetic. On the other hand, many recent processor architectures exhibit 32-bit computational power that is significantly higher

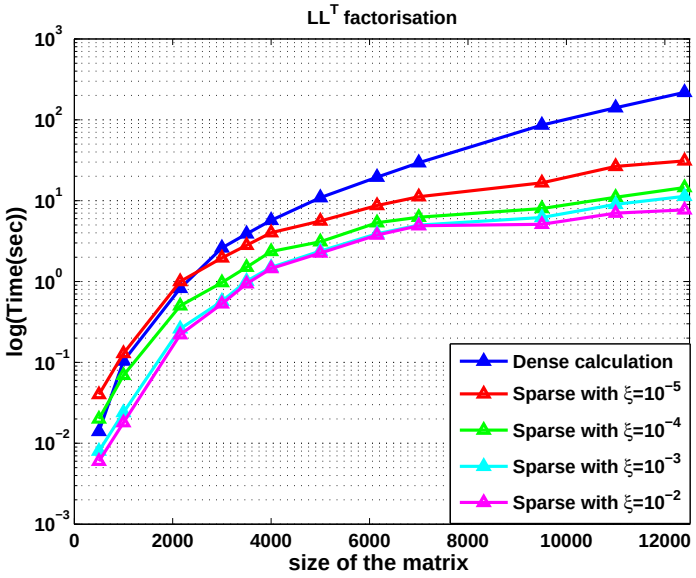


Figure 3.2: Performance comparison of dense v.s. sparse Cholesky factorization.

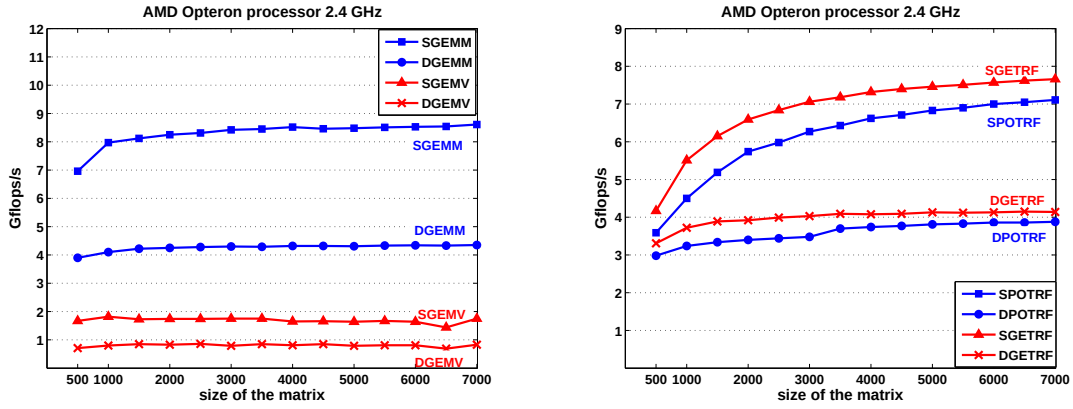
than that for 64-bit. One recent and significant example is the IBM CELL multiprocessor that is projected to have a peak performance near 256 GFlops in 32-bit and "only" 26 GFlops in 64-bit computation. More extreme and common examples are the processors that possess a SSE (streaming SIMD extension) execution unit can perform either two 64-bit instructions or four 32-bit instructions in the same time. This class of chip includes for instance the IBM PowerPC G5, the AMD Opteron and the Intel Pentium. For illustration purpose, are displayed below the time and the ratio of the time to perform a 32-bit operation over the time to perform the corresponding 64-bit one on some of the basic dense kernels involved in our hybrid solver implementation. In Table 3.4 are displayed the performance of BLAS-2 (_GEMV) and BLAS-3 (_GEMM) routines for various problems sizes. The comparison of the main LAPACK routines for the factorization and solution of dense problems is reported in Table 3.4 and Table 3.4.

It can be seen that 32-bit calculation generally outperforms 64-bit. For a more exhaustive set of experiments on various computing platforms, we refer to [21, 50, 64, 66]. The source of time reduction is not only the processing units that perform more operations per clock-cycle, but also a better usage of the complex memory hierarchy that provides ultra-fast memory transactions by reducing the stream of data block traffic across the internal bus and bringing larger blocks of computing data into the cache. This provides a speedup of two in 32-bit compared to 64-bit computation for BLAS-3 operations in most LAPACK routines. It can be shown that this strategy can be very effective on various, but not all architectures. Benefits are not observed on the Blue Gene/L machine.

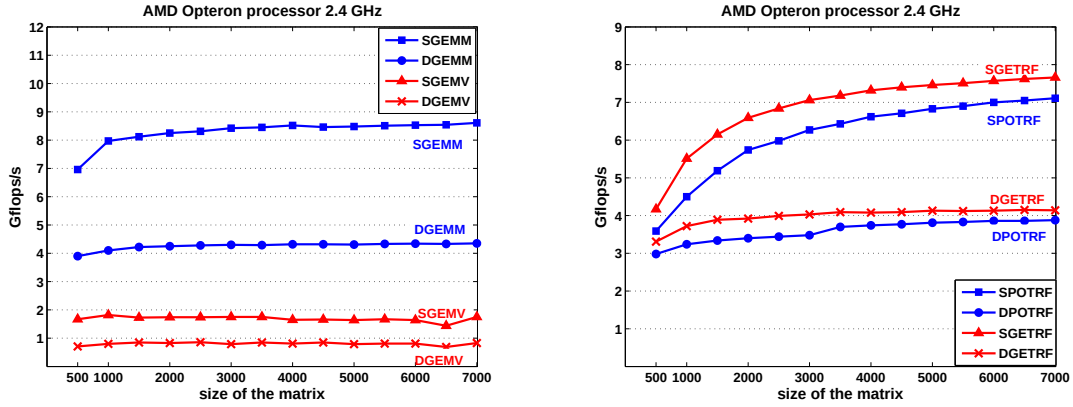
For the sake of readibility, the results reported in the tables are also plotted in graphs. In Figure 3.3, the graphs show the performance in GFlops/s of these various kernels. As mentioned above, the figures show that 32-bit performs twice as fast as 64-bit.

We might legitimately ask whether all the calculation should be performed in 64-bit or if some pieces could be carried out in 32-bit. This leads to the design of mixed-precision algorithms. Particular care is necessary when choosing the part to be computed in 32-bit arithmetic so that the introduced rounding error or the accumulation of these rounding errors does not produce a meaningless solution.

For the solution of linear systems, mixed-precision algorithms (single/double, double/quadruple)



CRAY AMD opteron processor 2.4 GHz (ACML).



Power PC G5 processor 2.3 GHz (VecLib).

Figure 3.3: Performance comparison of various BLAS-LAPACK Kernels.

CRAY XD1 AMD Opteron processor 2.4 GHz ACML

n	DGEMM	SGEMM	Ratio	DGEMV	SGEMV	Ratio
1000	0.49	0.25	1.95	0.003	0.001	2.27
2000	3.76	1.94	1.94	0.010	0.005	2.09
3000	12.55	6.42	1.96	0.023	0.010	2.21
4000	29.65	15.02	1.97	0.040	0.019	2.05
5000	57.99	29.47	1.97	0.064	0.030	2.09
6000	99.65	50.67	1.97	0.089	0.044	2.03
7000	157.55	79.72	1.98	0.118	0.056	2.11

IBM Power PC G5 processor 2.3 GHz VecLib

n	DGEMM	SGEMM	Ratio	DGEMV	SGEMV	Ratio
1000	0.22	0.12	1.76	0.005	0.002	2.53
2000	1.53	0.81	1.88	0.018	0.007	2.80
3000	4.89	2.71	1.80	0.043	0.020	2.15
4000	11.20	5.86	1.91	0.117	0.029	4.00
5000	21.64	11.52	1.88	0.114	0.052	2.21
6000	36.53	19.50	1.87	0.162	0.068	2.39
7000	58.09	31.04	1.87	0.225	0.102	2.20

BlueGene/L PowerPC440 processor 700 MHz ESSL

n	DGEMM	SGEMM	Ratio	DGEMV	SGEMV	Ratio
1000	0.87	0.78	1.11	0.005	0.003	1.71
2000	6.59	6.04	1.09	0.017	0.011	1.46
3000	21.85	20.05	1.09	0.036	0.025	1.43
4000	52.09	47.74	1.09	0.061	0.045	1.36
5000	100.59	92.91	1.08	0.099	0.069	1.43

Table 3.1: Elapsed time (sec) to perform BLAS-2 routines on various platforms when the size n of the matrices is varied.

have been studied in dense and sparse linear algebra mainly in the framework of direct methods (see [31, 30, 64, 66]). For such approaches, the factorization is performed in low precision, and, for not too ill-conditioned matrices, a few steps of iterative refinement in high precision arithmetic is enough to recover a solution to full 64-bit accuracy [30]. For nonlinear systems, though, mixed-precision arithmetic is the essence of algorithms such as inexact Newton.

For linear iterative methods, we might wonder if such mixed-precision algorithms can be designed. We propose to take advantage of the 32-bit speed and memory benefit and build some part of the code in 32-bit arithmetic. Our goal is to use costly 64-bit arithmetic only where necessary to preserve accuracy. A first possible idea, in Krylov subspace methods, is to perform all the steps except the preconditioning in 64-bit. Although this idea might appear natural at a first glance, the backward error stability result on GMRES [34, 77] indicates that such a variant would not enable achieve an accuracy below the 32-bit accuracy. For unsymmetric problems, different alternative can be considered. The first one is to use GMRES preconditioned with a 32-bit preconditioner to solve the residual equation within an iterative refinement scheme. This would correspond to a variant of the GMRES(ξ) described in [47]. The resulting algorithm ressembles very much to the classical right preconditioned GMRES, except that at restart the new residual is computed using the current approximation of x and not from $t = Mx$ as in the classical approach. Another alternative, would be to follow ideas in [5, 6, 24] and to use a 32-bit preconditioner for a FGMRES runs in 64-bit arithmetic. The 32-bit calculation is viewed as a variable 64-bit preconditioner for FGMRES.

CRAY XD1 AMD Opteron processor 2.4 GHz ACML

n	DPOTRF	SPOTRF	Ratio	DPOTRS	SPOTRS	Ratio
1000	0.10	0.07	1.39	0.003	0.001	2.36
2000	0.78	0.46	1.69	0.010	0.005	2.06
3000	2.59	1.43	1.80	0.021	0.010	2.07
4000	5.70	3.22	1.77	0.038	0.019	2.00
5000	10.95	6.10	1.79	0.060	0.029	2.07
6000	18.67	10.29	1.82	0.086	0.042	2.04
7000	29.44	16.07	1.83	0.116	0.057	2.04

IBM Power PC G5 processor 2.3 GHz VecLib

n	DPOTRF	SPOTRF	Ratio	DPOTRS	SPOTRS	Ratio
1000	0.10	0.07	1.43	0.007	0.005	1.38
2000	0.46	0.27	1.69	0.025	0.018	1.33
3000	1.33	0.73	1.83	0.057	0.041	1.41
4000	2.56	1.47	1.74	0.095	0.083	1.15
5000	4.74	2.73	1.74	0.147	0.126	1.17
6000	7.75	4.62	1.68	0.223	0.190	1.17
7000	12.12	6.82	1.78	0.279	0.264	1.05

BlueGene/L PowerPC440 processor 700 MHz ESSL

n	DPOTRF	SPOTRF	Ratio	DPOTRS	SPOTRS	Ratio
1000	0.24	0.21	1.17	0.006	0.003	1.87
2000	1.81	1.51	1.20	0.016	0.011	1.40
3000	5.88	4.94	1.19	0.034	0.024	1.42
4000	13.70	11.54	1.19	0.059	0.042	1.40
5000	26.43	22.31	1.18	0.092	0.064	1.42

Table 3.2: Elapsed time (sec) to perform LAPACK routines for SPD matrices on various platforms when the size n of the matrices is varied.

For symmetric positive definite case, no backward stability result exists for the preconditioned conjugate gradient method. In that context, and without theoretical explanation, we simply consider 32-bit preconditioner in a PCG where all the other computations are performed in 64-bit arithmetic.

In these variants, the Gaussian elimination [64, 66] (factorization) of the local assembled Schur complement (used as preconditioner), and the forward and the backward substitutions to compute the preconditioned residual, are performed in 32-bit while the rest of the algorithm is implemented in 64-bit.

Since the local assembled Schur complement is dense, cutting the size of this matrix in half has a considerable effect in terms of memory space. Another benefit is in the total amount of communication that is required to assemble the preconditioner. As for the memory required to store the preconditioner, the size of the exchanged messages is also half that for 64-bit. Consequently, if the network latency is neglected, the overall time to build the preconditioner for the 32-bit implementation should be half that for the 64-bit implementation. These improvements are illustrated by detailed numerical experiments with the mixed-precision implementation reported in Part II.

Finally, we mention that the ideas of sparsification and mixed precision arithmetic can be combined; that is, dropping the smallest entries of 32-bit $\bar{\mathcal{S}}_i$, to produce preconditioner cheap to compute and to store. In Chapter 5, we report some experiments combining the two strategies.

CRAY XD1 AMD Opteron processor 2.4 GHz ACML

n	DGETRF	SGETRF	Ratio	DGETRS	SGETRS	Ratio
1000	0.18	0.12	1.48	0.003	0.001	2.25
2000	1.36	0.81	1.68	0.010	0.004	2.24
3000	4.47	2.55	1.75	0.021	0.009	2.27
4000	10.47	5.83	1.80	0.039	0.017	2.30
5000	20.17	11.16	1.81	0.057	0.029	1.99
6000	34.88	19.02	1.83	0.080	0.042	1.91
7000	55.19	29.84	1.85	0.120	0.055	2.18

IBM Power PC G5 processor 2.3 GHz VecLib

n	DGETRF	SGETRF	Ratio	DGETRS	SGETRS	Ratio
1000	0.13	0.07	1.85	0.007	0.005	1.29
2000	0.69	0.39	1.76	0.026	0.019	1.40
3000	2.00	1.17	1.72	0.067	0.045	1.47
4000	4.41	2.50	1.76	0.106	0.105	1.01
5000	8.33	4.75	1.75	0.179	0.167	1.07
6000	14.42	7.77	1.86	0.245	0.240	1.02
7000	21.35	12.16	1.76	0.355	0.352	1.01

BlueGene/L PowerPC440 processor 700 MHz ESSL

n	DGETRF	SGETRF	Ratio	DGETRS	SGETRS	Ratio
1000	0.42	0.35	1.20	0.006	0.003	1.85
2000	3.18	2.57	1.24	0.017	0.012	1.43
3000	9.83	8.32	1.18	0.037	0.025	1.45
4000	24.04	19.74	1.22	0.061	0.045	1.37
5000	43.65	37.82	1.15	0.095	0.068	1.40

Table 3.3: Elapsed time (sec) to perform LAPACK routines for general matrices on various platforms when the size n of the matrices is varied.

3.5 Two-level preconditioner with a coarse space correction

The solution of elliptic problems is challenging on parallel distributed memory computers as their Green's functions are global. Consequently the solution at each point depends upon the data at all other points. Therefore, for solving the systems arising from the discretization of these equations, we have to provide a mechanism that captures the global coupling behaviour.

Various domain decomposition techniques, from the eighties and nineties, have suggested different global coupling mechanisms, referred to as the coarse space components, and various combinations between them and the local preconditioners. These can be based on geometric ideas (e.g. linear interpolation), finite element ideas (e.g. finite element basis functions corresponding to a coarse mesh), or algebraic ideas (e.g. using the matrix coefficients to define basis functions with minimal A-norm in the SPD case). Again, there are trade-offs in these different approaches. Geometric schemes are somewhat complicated to implement and are often tied to the resulting application code. Applications with complex geometric features can be particularly challenging to develop. Additionally, they may have robustness issues for problems with highly heterogeneous behaviour as the interpolation and restriction do not use material, PDE, or discretization properties. While finite element approaches are more closely tied to the discrete system, they require a more explicit notion of a coarse mesh which makes sense in a finite element context (e.g. all coarse elements are convex). This can be particularly difficult when irregular boundaries are present. Algebraic methods have an advantage in that they do not require an explicit

mesh and by using a matrix they have *indirect* access to material, PDE, and discretization properties. Unfortunately, it is not always computationally easy to deduce basic properties of an operator based only on matrix coefficients. In the framework of non-overlapping domain decomposition techniques, we refer for instance to algebraic two-level preconditioner for the Schur complement [25, 26], BPS (Bramble, Pasciak and Schatz) [20], Vertex Space [35, 93], and to some extended Balancing Neumann-Neumann [67, 69, 70], as well as FETI [41, 71], for the presentation of major two-level preconditioners.

Although the local preconditioner proposed in Section 3.2 introduces some exchanges of information, these exchanges remain local to neighbouring subdomains and introduce no global coupling mechanism. This mechanism is necessary for elliptic problems to prevent an increase in the number of iterations when the number of subdomains is increased. The literature on generating coarse spaces is quite extensive. Here, we simply mention one possible algebraic technique that has been applied successfully on several problems and that is relatively straight-forward to implement in parallel [26]. It also has an advantage in that it does not require any geometric information.

The preconditioners presented now are closely related to the BPS preconditioner, although we consider different coarse spaces to construct their coarse components. The class of two-level preconditioner that we define now can be described in a generic way as the sum of a local and global component:

$$M = M_{AAS} + M_0,$$

where :

M_{AAS} is one of the variants of the additive Schwarz preconditioner described in the previous section,

M_0 is a low rank correction computed by solving a coarse system.

For practical implementation purposes within a general purpose computer code, we do not want to refer explicitly to an underlying coarse grid, or to underlying basis functions, since these notions are always hard to identify in practice when using general grids, finite elements or mixed finite elements.

The coarse component can be described as follows. Let U be the algebraic space of nodal vectors where the Schur complement matrix is defined and U_0 be a q-dimensional subspace of U. Elements of U_0 are characterized by the set of nodal values that they can achieve. This subspace will be called coarse space.

Let $R_0 : U \to U_0$ be a restriction operator which maps full vectors of U into vectors in U_0, and let $R_0^T : U_0 \to U$ be the transpose of R_0, an extension operator which extends vectors from the coarse space U_0 to full vectors in the fine space U.

The Galerkin coarse space operator

$$\mathcal{S}_0 = R_0 \mathcal{S} R_0^T, \tag{3.7}$$

in some way, represents the Schur complement on the coarse space U_0.

The global coupling mechanism is introduced by the coarse component of the preconditioner which can thus be defined as $M_0 = R_0^T \mathcal{S}_0^{-1} R_0$.

Based on this algebraic construction various coarse-space preconditioners can be considered that only differ in the choice of the coarse space U_0 and the interpolation operator R_0^T. For convergence reasons, and similarly to the Neumann-Neumann and Balancing Neumann-Neumann preconditioner [67, 69], R_0^T must be a partition of the unity in U in the sense that

$$R_0^T \mathbf{1} = \mathbf{1}, \tag{3.8}$$

where the symbol $\mathbf{1}$ denotes the vectors of all 1's that have different size in the right and left-hand side of (3.8).

In this work we consider a coarse space where we associate one coarse point with each subdomain. Let B be the set of unknowns belonging to the interface Γ between subdomains. Let Ω_k be a subdomain and $\partial\Omega_k$ its boundary. Then

$$\overline{\mathcal{I}}_k = \partial\Omega_k \cap B$$

is the set of indices we associate with the domain Ω_k. Figure 3.4 shows the elements of a certain set $\overline{\mathcal{I}}_k$.

Let $\mathcal{Z}_k$ be a vector defined on Γ and $\mathcal{Z}_k(i)$ its i-th component. The support of the basis vectors $\mathcal{Z}_k$ has inspired the name of the coarse spaces. Then, the subdomain-based coarse space U_0 can be defined as

$$U_0 = \text{span}[\mathcal{Z}_k : k = 1, \ldots, N], \text{ where } \mathcal{Z}_k(i) = \begin{cases} 1, & \text{if } i \in \overline{\mathcal{I}}_k \text{and} \\ 0, & \text{otherwise.} \end{cases}$$

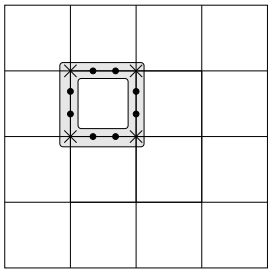


Figure 3.4: *Support of one basis vector of the "subdomain" coarse space.*

Notice that for the example depicted in Figure 3.4, $[\mathcal{Z}_k]$ is rank deficient. Indeed, if we consider $\tilde{v} = \sum_{i=1}^{N} \alpha_i \mathcal{Z}_i$ where the α_i are , in a checker-board pattern, equal to -1 and $+1$, it is easy to see that $\tilde{v} = 0$.
Nevertheless, this rank deficiency can be easily removed by discarding one of the vectors of $[\mathcal{Z}_k]$. In this particular situation, the set of vectors $\mathcal{B} = \{\mathcal{Z}_1, \mathcal{Z}_2, \ldots, \mathcal{Z}_{N-1}\}$ forms a basis for the subspace U_0.
The considered restriction operator R_0 returns for each subdomain $(\Omega_i)_{i=1,N-1}$ the weighted sum of the values at all the nodes on the boundary of that subdomain. The weights are determined by the inverse of the number of subdomains in $(\Omega_i)_{i=1,N-1}$ each node belongs to. For all the nodes but the ones on $\partial\Omega_N$ (in our particular example) this weight is: $1/2$ for the points on an edge and $1/4$ for the cross points. These weights can be replaced as in [67] by operator dependent weights $R_0(i,k) = a_i/(a_i + a_j)$ on the edge separating Ω_i from Ω_j , but this choice has not been tested numerically in the present work.

Remark 3.5.1 *Although used in a completely different context, this coarse space is similar to the one used in the Balancing Neumann-Neumann preconditioner for Poisson-type problems [69]. We use one basis vector for each subdomain, whereas in Balancing Neumann-Neumann the basis vectors are only defined for interior subdomains for solving the Dirichlet problem, that are the subdomains where the local Neumann problems are singular.*

To conclude, although we have also consider the solution of unsymmetric problem in this work, we have not investigated coarse space correction based on Petrov-Galerkin approaches that are sometimes used in multigird.

3.6 Scaling the Schur complement

In some simulations the dynamic of the computed quantities is high and leads to variations in the coefficients of the linear systems. Consequently these variations also appear in the associated Schur complement systems. Therefore, we investigate a scaling technique that has been implemented and evaluated. For most of the linear systems the largest entries are located on the diagonal. In that context, a diagonal scaling was the best trade-off between the numerical efficiency and the parallel implementation efficiency. This technique is relatively easy to implement for scaling the Schur complement system when the local Schur complement are built explicitly.

We consider the solution of

$$\mathcal{S}u = f, \tag{3.9}$$

and denote by (s_{ij}) the entries of $\mathcal{S}$. The diagonal scaling of (3.9) consists in solving

$$D\mathcal{S}Dv = Df, \quad u = Df, \tag{3.10}$$

where $D = diag((\sqrt{|s_{ii}|})^{-1})$.

Remark 3.6.1 *When the original matrix $\mathcal{S}$ is symmetric, by construction, the diagonal scaling preserves this property as well as the positive definiteness, if $\mathcal{S}$ is.*

Let $\mathcal{S}$ denote the Schur complement matrix associated with the original matrix $\mathcal{A}$. Instead of scaling the Schur complement system, it is also possible to scale the original matrix $\mathcal{A}$ before computing the local Schur complement matrices. We consider the diagonal scaling for $\mathcal{A}$ meaning that the system $\mathcal{A}x = b$, is replaced by the by $D\mathcal{A}Dy = Db,\ x = Dy$ where D is the scaling matrix computed from the diagonal entries of $\mathcal{A}$ by $d_{ii} = (\sqrt{|a_{ii}|})^{-1}$. If we order first the internal unknowns and then the ones on the interface we obtain

$$\begin{pmatrix} \mathcal{A}_{II} & \mathcal{A}_{I\Gamma} \\ \mathcal{A}_{\Gamma I} & \mathcal{A}_{\Gamma\Gamma} \end{pmatrix}. \tag{3.11}$$

Reordering in a consistent manner the diagonal scaling matrix leads to

$$\begin{pmatrix} D_I & 0 \\ 0 & D_\Gamma \end{pmatrix} \begin{pmatrix} \mathcal{A}_{II} & \mathcal{A}_{I\Gamma} \\ \mathcal{A}_{\Gamma I} & \mathcal{A}_{\Gamma\Gamma} \end{pmatrix} \begin{pmatrix} D_I & 0 \\ 0 & D_\Gamma \end{pmatrix} = \begin{pmatrix} D_I\mathcal{A}_{II}D_I & D_I\mathcal{A}_{I\Gamma}D_\Gamma \\ D_\Gamma\mathcal{A}_{\Gamma I}D_I & D_\Gamma\mathcal{A}_{\Gamma\Gamma}D_\Gamma \end{pmatrix}. \tag{3.12}$$

Eliminating the internal scaled equations we obtain

$$\mathcal{S}_{scaled} = D_\Gamma\mathcal{A}_{\Gamma\Gamma}D_\Gamma - D_\Gamma\mathcal{A}_{\Gamma I}D_I(D_I\mathcal{A}_{II}D_I)^{-1}D_I\mathcal{A}_{I\Gamma}D_\Gamma = D_\Gamma S D_\Gamma$$

where $\mathcal{S}$ is the Schur system associated with the unscaled matrix $\mathcal{A}$. This observation is also true for the row, the column scalings that are not considered in our study. This indicates that scaling the original matrix leads to scale the Schur complement $\mathcal{S}$ using entries of $\mathcal{A}$.

Chapter 4

Design of parallel distributed implementation

4.1 Introduction

Massively parallel computers promise unique power for large engineering and scientific simulations. The development of efficient parallel algorithms and methods that fully exploit this power is a grand challenge for computational researchers. Large parallel machines are likely to be the most widely used machines in the future, involving an important consideration in parallel methods and algorithm designs.

Domain decomposition is a natural approach to split the problem into subproblems that are allocated to the different processors in a parallel algorithm. This approach is referred to as the classical parallel domain decomposition method. A *2-level parallel* algorithm [49] will attempt to express parallelism between the subproblems but also in the treatment of each subproblem. Since each subdomain will be handled by more than one processor, communication strategies, data structures and algorithms need to be rewritten to reflect the distribution of the work across multiple processors.

In this chapter, the classical parallel domain decomposition implementation (refer to as *1-level parallel*) is described. It is followed by a discussion of the *2-level parallel* method, program design and performance considerations. We describe the implementation of the *2-level parallel* method, that allows us to provide an efficient, parallel algorithm for scientific applications possessing multi-level of tasks and data parallelism. Achieving high performance is at the top level on our list of priority. In this context, we strive to center the *2-level parallel* implementation around an efficient use of the available parallel numerical linear algebra kernels such as ScaLapack, Pblas, Blas [18] and Mumps [2, 3] on top of MPI [56]. For the Krylov subspace solvers, we consider the packages suited for parallel distributed computing [42, 43, 44].

4.2 Classical parallel implementations of domain decomposition method

4.2.1 Introduction

In the Schur substructuring method, the underlying mesh is subdivided into blocks (submeshes). The idea is to map the blocks to processors. Then, for each block, the internal degree of freedoms are eliminated, using a direct method, leading to a reduced system of equations that involve only the interface degrees of freedoms. The internal elimination is carried out independently on each processor and requires no communication. The remaining parallel problem, the reduced system,

is then solved using an appropriate preconditioned Krylov subspace solver such as CG, GMRES, MINRES, BiCGSTAB, etc... [86]. Iterative solver is used because it exhibits better performance and it is easier to implement on large parallel machines than the sparse direct solvers. Once the iterative process has converged to the desired accuracy, the solution of the reduced system is used simultaneously by the direct solver to perform the solution for the interior degree of freedoms. Thus the hybrid approach can be decomposed into three main phases:

- the first phase *phase1* consists into the local factorization and the computation of the local Schur complements,
- the setup of the preconditioners (*phase2*),
- the iterative phase (*phase3*).

We describe below the main algorithmic and software tools we have used for our parallel implementation. In Section 4.2.2, we present briefly the multifrontal method and the direct software MUMPS which is a parallel package for distributed platforms. In Section 4.2.3, we discuss the efficient implementation of both local ($M_{\mathrm{d-64}}$, $M_{\mathrm{d-mix}}$ and $M_{\mathrm{sp-64}}$) and global components of the preconditioner. Finally in Section 4.2.4, we describe the parallel implementation of the main kernels of the iterative solvers.

4.2.2 Local solvers

Many parallel sparse direct algorithms have been developed such as multifrontal approaches [37, 38], supernodal approaches [32] and Fan-both algorithms [8]. Our work is based on the multifrontal approach. This method is used to compute the LU or LDL^T factorizations of general sparse matrix. Among the few available parallel distributed direct solvers, MUMPS offers a unique feature, which is the possibility to compute the Schur complements defined in Equation (4.1) using efficient sparse calculation techniques,

$$\mathcal{S}_i = \mathcal{A}_{\Gamma_i\Gamma_i} - \mathcal{A}_{\Gamma_i\mathcal{I}_i}\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}^{-1}\mathcal{A}_{\mathcal{I}_i\Gamma_i}. \tag{4.1}$$

This calculation is performed very efficiently as MUMPS implements a multifrontal approach [37] where local Schur complements are computed at each step of the elimination tree process (during the factorization of each frontal matrix) and is based on level 3 BLAS routines. Basically, the Schur complement feature of MUMPS can be viewed as an partial factorization, where the factorization of the root, associated with the indices of $\mathcal{A}_{\Gamma_i\Gamma_i}$, is disabled. Consequently this feature fully benefits from the general overall efficiency of the multifrontal approach implemented by MUMPS. From a software point of view, the user must specify the list of indices associated with $\mathcal{A}_{\Gamma_i\Gamma_i}$. The code then provides a factorization of the $\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}$ matrix and the explicit Schur complement matrix $\mathcal{S}_i$. The Schur complement matrix is returned as a dense matrix. The partial factorization that builds the Schur complement matrix can also be used to solve linear systems associated with the matrix $\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}$.

The MUMPS software

The software MUMPS (MUltifrontal Massively Parallel Solver) is an implementation of the multifrontal techniques for parallel platforms. It is written in Fortran 90 and use new functionalities of this language (modularity, dynamic memory allocation). We present here the main features of this package.

- **Factorization:** of sparse symmetric positive definite matrices (LDL^T factorization), general symmetric matrices and general unsymmetric matrices (LU factorization).
- **Entry format for the matrices:** The matrix can be given in different formats. The three formats that can be used are:

  - the centralized format where the matrix is stored in coordinate format on the root processor,
  - the distributed format where each processor own a subset of the matrix described in a coordinate format, defined in global ordering,
  - the elemental format where the matrix is described as a sum of dense elementary matrices.

- **Ordering and scaling:** the code implements different orderings such as AMD [1], QAMD, PORD [89], METIS [62] nested dissection, AMF, and user defined orderings.

- **Distributed or centralized Schur complement:** the software enables us to compute the Schur complement in a explicit way. The Schur complement matrix is returned as a dense matrix. It can be returned as a centralized matrix on the root processor or as a distributed 2D block-cyclic matrix.

4.2.3 Local preconditioner and coarse grid implementations

In this subsection we discuss both the local and the global (coarse grid correction) component of the preconditioner considered in our work.

Local preconditioner:

This phase depends on the variant of the preconditioner used. For dense preconditioner it consists in assembling the local Schur complement computed by the direct solver, and then to factorize them concurrently using LAPACK kernels. For mixed arithmetic preconditioner, it consists in assembling the local Schur complement in 32-bit arithmetic, and then to factorize them concurrently using LAPACK kernels. For the sparse preconditioner, it consists in assembling the local Schur complement, to sparsify them concurrently, then to factorize them using the sparse direct solver MUMPS. The assembly phase consists in exchanging part of the local Schur data between neighbouring subdomains. This step can be briefly described by Algorithm 4.

Algorithm 4 Assembling the local Schur complement

1: $\bar{\mathcal{S}}_i = \mathcal{S}_i$ or $\bar{\mathcal{S}}_i = sngl(\mathcal{S}_i)$ for $M_{\mathrm{d-mix}}$
2: **for** $k = 1, nbneighbour$ **do**
3: Bufferize SEND part of $\mathcal{S}_i$ to neighbour k;
4: **end for**
5: **for** $k = 1, nbneighbour$ **do**
6: Receive RECV part of $\mathcal{S}_i$ from neighbour k: $buffer_{temp} \leftarrow$ RECV()
7: Update $\bar{\mathcal{S}}_i \leftarrow \bar{\mathcal{S}}_i + buffer_{temp}$.
8: **end for**

Construction of the coarse part:

The coarse matrix is computed once as described in Algorithm 5. Because the matrix associated with the coarse space is small, we decide to redundantly build and store this matrix on all the processors. By this way we expect that applying the coarse correction at each step of the iterative process only implies one global communication for the right-hand side construction [26]. The coarse solution is then performed simultaneously by all processors. So at the slight cost of storing the coarse matrix, we can cheaply apply this component of the preconditioner.

4.2.4 Parallelizing iterative solvers

The efficient implementation of a Krylov method strongly depends on the implementation of three computational kernels, that is the matrix-vector product, applying the preconditioner to a vector, and the dot product calculation.

Algorithm 5 Construction of the coarse component

1: Each processor calls GEMM to compute $tempS \leftarrow \mathcal{S}R_0^T$
2: Each processor calls GEMM to compute $S_0loc \leftarrow R_0 tempS$
3: Each processor reorders $\mathcal{S}_0 \leftarrow S_0loc$ in subdomains order
4: Assemble $\mathcal{S}_0$ in all processors
5: Factorize $\mathcal{S}_0$ simultaneously in all processors

matrix-vector product: $y_i = \mathcal{S}_i x_i$

It can be performed in two ways, explicitly using BLAS-2 routine or implicitly using sparse matrix-vector calculations. The explicit computation is described by Algorithm 6, whereas the implicit one is given by Algorithm 7.

Algorithm 6 Explicit matrix-vector product

1: Completely parallel and does not need any communication between processors.
Each processor call to DGEMV compute $y_i \leftarrow \mathcal{S}_i x_i$
2: Update data: it needs some exchange of informations between neighbouring subdomains.
Each processor assembles $y \leftarrow \sum_{i=1}^{nbneighbour} \mathcal{R}_{\Gamma_i} y_i$
3: **for** $k = 1, nbneighbour$ **do**
4: Bufferize SEND part of y_i to neighbour k;
5: **end for**
6: **for** $k = 1, nbneighbour$ **do**
7: Receive RECV part of y_i from neighbour k: $y_{temp} \leftarrow$ RECV()
8: Update $y_i \leftarrow y_i + y_{temp}$.
9: **end for**

Algorithm 7 Implicit matrix-vector product

1: Each processor compute a sparse matrix vector product $y_i \leftarrow \mathcal{A}_{\mathcal{I}_i\Gamma_i} x_i$
We use a special subroutine for sparse matrix vector product
2: Concurrently, each processor call MUMPS to perform a forward/backward substitution $y_i \leftarrow \mathcal{A}^{-1}_{\mathcal{I}_i\mathcal{I}_i} y_i$ using the computed factors of $\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}$
3: Then also in parallel, each processor computes the sparse matrix-vector product
$y_i \leftarrow \mathcal{A}_{\Gamma_i\Gamma_i} x_i - \mathcal{A}_{\Gamma_i\mathcal{I}_i} y_i$
4: Last step (update data): it needs some exchange of informations between neighbouring subdomains.
Each processor assembles $y \leftarrow \sum_{i=1}^{nbneighbour} \mathcal{R}_{\Gamma_i} y_i$

Applying the preconditioner: $y_i = M_i^{-1} x_i$

This step described in Algorithm 8 can be performed using either LAPACK kernels for the dense preconditioner or a forward/backward substitution using the sparse solver MUMPS for the sparse preconditioner.

The dot product: $y_i = y_i^T x_i$

The dot product calculation is simply a local dot-product computed by each processor followed by a global reduction to assemble the complete result as described in Algorithm 9.

Algorithm 8 Applying the preconditioner

1: In parallel each processor performs the triangular solve $y_i \leftarrow M_i^{-1} x_i$
2: Update data: exchange of informations between the neighbouring subdomains.
Each processor assemble $y \leftarrow \sum_{i=1}^{nbneighbour} \mathcal{R}_{\Gamma_i} y_i$,

Algorithm 9 Parallel dot product

1: In parallel each processor performs the local dot product $y_i \leftarrow y_i^T x_i$
2: Global reduction across all the processors: MPI_ALLREDUCE(y_i)

4.3 Two-level parallelization strategy

4.3.1 Motivations for multi-level parallelism

Initially this work was motivated by the fact that, many challenge real simulations scale well in parallel, but execute at an unsatisfying percentage of the peak performance. The main goal of the development of the *2-levels* of parallelism approach is the investigation of numerical methods for the efficient use of parallel modern machines. Classical parallel implementations (*1-level parallel*) of domain decomposition techniques assign one subdomain per processor. We believe that applying only this paradigm to very large applications has some drawbacks and limitations:

- For many applications, increasing the number of subdomains often leads to increasing the number of iterations to converge. If no efficient numerical mechanism, such as coarse space correction for elliptic problems [17, 94], is available the solution of very large problems might become ineffective. To avoid this, one can instead of increasing the number of subdomains, keep it small while handling each subdomain by more than one processor introducing *2-levels* of parallelism. This latter benefit is what we called "*the numerical improvement*" of the *2-level parallel* method. The description of this idea is illustrated in Figure 4.1.

- Large $3D$ systems often require a huge amount of data storage so that the memory required to handle each subdomain is not available for each individual processor. On SMP (Symmetric Multi-Processors) node this constraint can be relaxed as we might only use a subset of the available processors to allow the exploited processors to access more memory. Although such a solution enables simulations to be performed, some processors are "wasted", as they are "idle" during the computation. In that context, the simulation executes at an unsatisfying percentage of per-node peak floating-point operation rates. The "idle" processors might contribute to the treatment of the data stored into the memory of the node. This takes advantage of the "idle" processors and runs closer to the peak of per-node performance as described in Figure 4.2. We call this "*the parallel performance improvement*" of the *2-level parallel* method.

- Very large simulations might require substantially larger computational resources than available on a node of the target machine. The memory required by each subdomain computation is larger than the memory available on each node, thus the solution of the sparse linear system cannot be performed using *1-level* of parallelism. Such a situation can also be addressed using our *2-level parallel* implementation as described in Figure 4.3 for the case of a cluster of SMP target computer. This idea is also called "*the parallel performance improvement*" of the *2-level parallel* method.

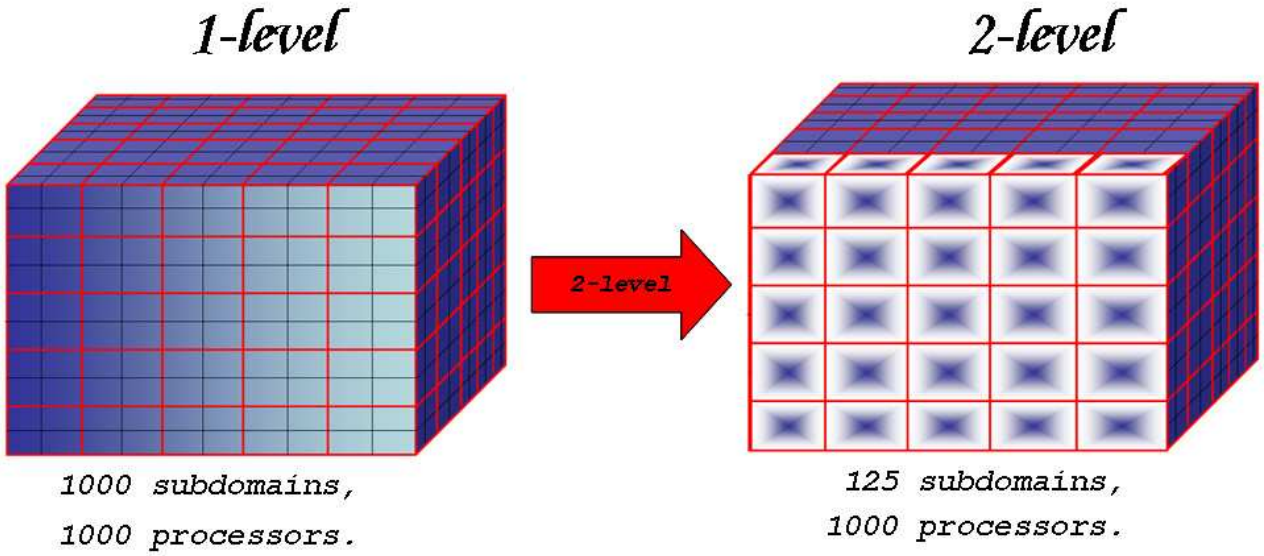
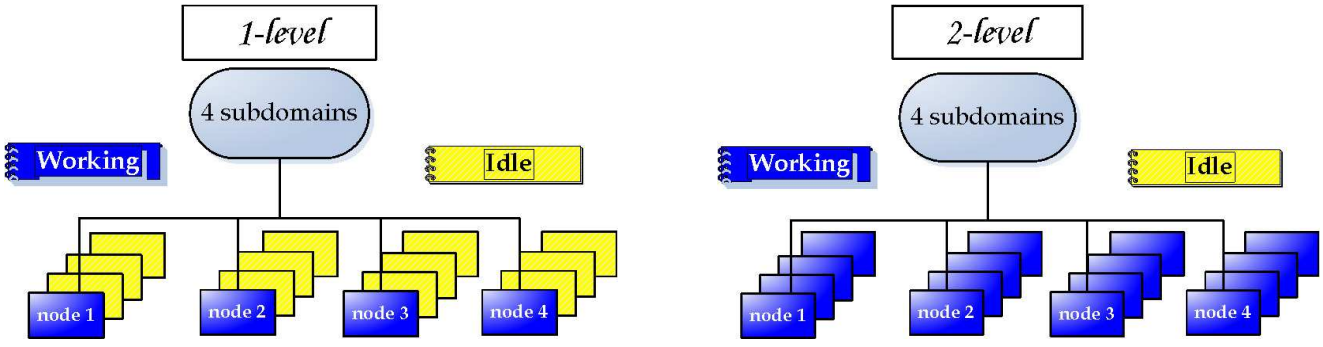


Figure 4.1: Comparison between *1-level parallel* and *2-level parallel* method on 1000 processors, when instead of having 1000 subdomains, we decrease the number of subdomains to 125 while running each one onto 4 processors.



(a) 1 processor working, 3 processors "*idle*".

(b) 4 processors working, 0 processor "*idle*".

Figure 4.2: Comparison between *1-level parallel* and *2-level parallel* method on 4 SMP-node quadri-processors, when each subdomain require the overall memory of a node.

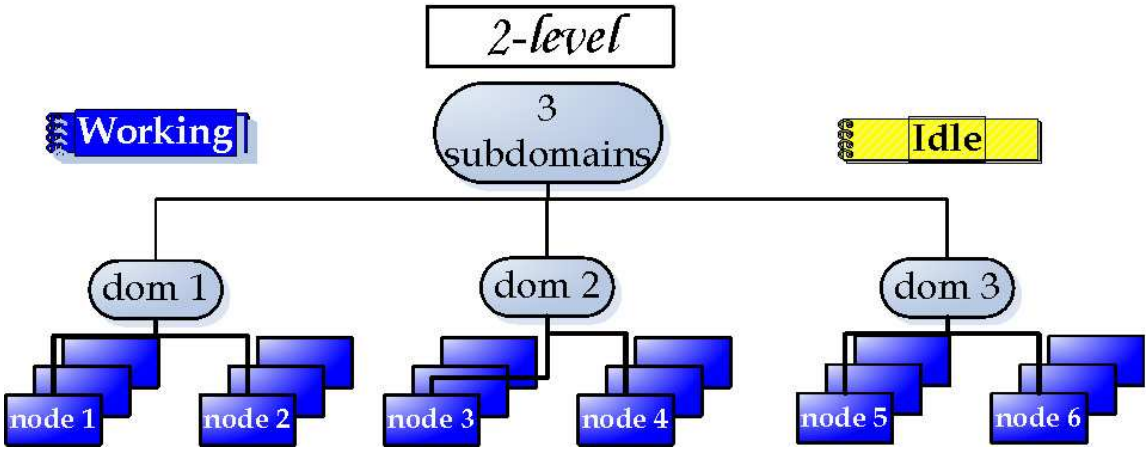


Figure 4.3: *2-level parallel* method, when each subdomain require more than the memory available on an SMP-node.

4.3.2 Parallel BLACS environments

We describe in this section, the basics of the BLACS environment used as a communication layer for both PBLAS and ScaLAPACK libraries. Before calling the parallel linear algebra routines for each subdomain, we need to define a grid of processors on which the linear algebra routines will operate in parallel. A set of parallel processors (group/communicator) with k processors is often thought of as a one dimensional array of processes labeled 0,1,..k-1. For performance reasons, it is sometimes better to map this one dimensional array into a logical two-dimensional rectangular grid, which is also referred to as process grid of processors. The process grid can have p processor rows and q processor columns, where $p \times q = k$. A processor can now be indexed by row and column. This logical rectangular grid may not necessarily be reflected by the underlying hardware. The user must define the number of processors row and processors column of a grid, which is nothing else than the number of processors of the group/communicator. In our *2-level parallel* implementation the code initializes several grid of processors, as many as the number of subdomains.

4.3.3 Multi-level of task and data parallelism

In this subsection, we present the basic concepts of data distribution over the processor grid. When each subdomain is handled by a group of processors (grid), all the linear algebra objects (vector and matrices) should be distributed across the processors of the grid. So, each subdomain data are mapped to the memory of the grid processors assuming specific data distributions. The local data on each processor of the grid is referred to as the local array. Parallel linear algebra routines assume that data has been distributed to the processors with one-dimensional or two-dimensional block-cyclic data scheme. This distribution is a natural expression of the block partitioning algorithms available in ScaLAPACK. On the left-hand side of Figure 4.4 we depict the one-dimensional block-cyclic data distribution over a grid of $1 \times 4 = 4$ processors. On the right-hand side of Figure 4.4 we display the two-dimensional block-cyclic data distribution over a grid of $2 \times 2 = 4$ processors. We refer the reader to the ScaLAPACK user guide [18] for more details.

As consequence, in the context of our hybrid domain decomposition method, the first distribution of data is performed naturally by the domain decomposition partitioning of the physical problem into N subdomains. The second data distribution is done within each subdomain on the local associated grid of processors (group) and according to the $2D$ block-cyclic distribution as shown in Figure 4.5

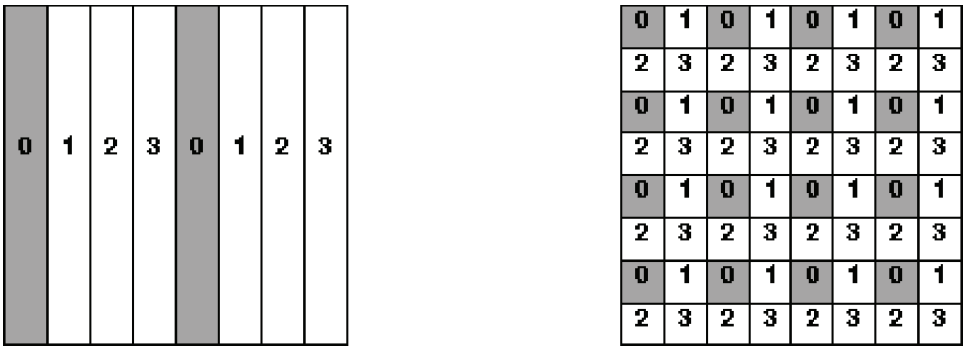


Figure 4.4: One dimensional (left) and two dimensional (right) block cyclic data distribution.

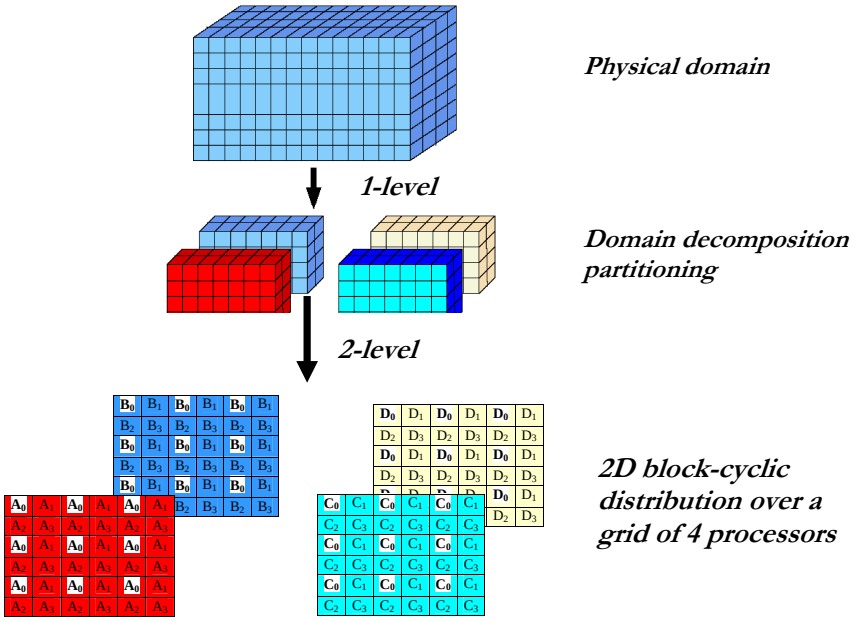


Figure 4.5: Multi-level of data and task distribution.

4.3.4 Mixing *2-levels* of parallelism and domain decomposition techniques

We focus here on the description of the *2-level parallel* method in the context of domain decomposition algorithms. The *2-level parallel* implementation will be effective for our hybrid solver if its main three phases can be efficiently performed in parallel. Let us quickly recall the main numerical kernels of our algorithm and for each of them describe the parallel strategy.

Initialization *phase1*

The idea of the *2-level parallel* method is to handle each subdomain using several processors. To allow multi-processing per subdomain, the algorithm should control several groups of processors, each of them working on different tasks associated with the subdomains. We define as many groups as subdomains and associate one MPI-communicator with each of them. As result, we perform simultaneously each local factorization in parallel taking the advantage of the sparse direct solver in a grid of processors (group). The computed Schur matrix is stored over the grid processors according to the $2D$ block-cyclic data distribution.

Preconditioner setup *phase2*

The local Schur complements are dense and distributed over the processor grid. This means that each processor stores blocks of rows and columns of the "local" Schur complement. Because processors must exchange data during the assembly step, the cost of this latter must be considered. This step does not depend on the number of processors and depend only on the number of neighbouring subdomains. We notice that, for the large simulations, the local Schur matrices are large. For the standard *1-level parallel* algorithm, only one processor has to communicate with its neighbouring subdomain/processor in order to assemble the local Schur matrix. In a multi-level parallel framework, we have to pay attention to perform this phase efficiently. For this purpose, each processor that stores part of the "local" Schur knows the identity of the processors handling the corresponding part of the neighbouring "local" Schur complements to have efficient point-to-point communication. This enables parallel communication and assembling of the preconditioner.

Iterative loop *phase3*

This phase involves three numerical kernels that are: the matrix-vector product, the preconditioner application and finally the global reduction step. The local Schur matrices are distributed over the local grid of processors so that both PBLAS and ScaLAPACK can be used easily.

For the matrix-vector product, the implementation is performed using the PBLAS routines to multiply the dense distributed local Schur complement with also the distributed vector u^k. The resulting distributed vector is updated directly between neighbouring subdomains as each processor associated with one subdomains knows its neighbours associated with neighbouring subdomains.

The preconditioner application relies either on ScaLAPACK kernels for the dense $M_{\mathrm{d}-64}$ preconditioner or MUMPS for the sparse $M_{\mathrm{sp}-64}$ preconditioner. Similarly to the matrix-vector product, the resulting distributed vector is updated.

For the dot product calculation, each processor owning the distributed vectors performs its local dot product then the results are summed using a simple global reduction.

Because each step can be parallelized, the iterative loop calculation greatly benefits from the *2-level parallel* implementation.

We summarize in Algorithm 10, the main algorithmic steps of our 2-level parallel implementation. The complexity of the actual code is of course by no means reflected by these few lines of algorithmic description.

Algorithm 10 *2-level parallel* method implementation

1: Define a set of groups of processors.
2: Define communicators for groups, masters of groups.
3: Initialize BLACS environments
4: **if** (I am Master of group) **then**
5: Partition problem into N subdomains (or generate N subdomains)
6: **end if**
7: Define new data-structure and new sub-indexing of variables
8: Simultaneously initialize parallel instance (over the grid processors) of direct solver
9: Perform on each subdomain parallel factorization and computation of the Schur complement.
10: Assemble local Schur complement $2D$ block-cyclic data distribution
11: Setup the preconditioner locally in parallel
12: **if** (I am Master of group) **then**
13: Distribute the RHS over the column grid of processors
14: **end if**
15: Perform the iterative loop
16: **for** $k = 1, convergence$ **do**
17: Perform matrix-vector product in parallel over the grid processors $y_i \leftarrow \mathcal{S}_i x_i$
18: Column processors communicate the value of the result $y \leftarrow \sum_{1}^{nbneighbour} \mathcal{R}_{\Gamma_i} y_i$
19: Perform preconditioner applications
20: Column processors communicate the value of the result $y \leftarrow \sum_{1}^{nbneighbour} \mathcal{R}_{\Gamma_i} y_i$
21: Perform dot product by the column processor.
22: **end for**
23: **if** (I am column processor or Master of group) **then**
24: Scatter the interface solution
25: **end if**
26: All processors of a group perform simultaneously the interior solution

II

Part II

Study of parallel scalability on large $3D$ model problems

Part II: résumé

Dans ces deux chapitres, nous allons illustrer le comportement numérique et les performances parallèles de notre approche par une série exhaustive d'expériences parallèles numériques dans un contexte académique. Cette étude exhaustive de l'extensibilité et l'efficacité parallèle de notre préconditionneur et de sa mise en oeuvre est réalisée sur des problèmes modèles de type équations de diffusions $3D$ au Chapitre 5 et déquations de convection-diffusion $3D$ au Chapitre 6. Pour chacun de ces modèles types, on considère différents problèmes en faisant varier la difficulté du système à résoudre. Pour les problèmes elliptiques, des situations avec de fortes discontinuités et anisotropie sont considérées. Pour les problèmes de convection-diffusion, on sintéresse en particulier à l'effet du nombre de Péclet sur la robustesse du préconditionneur. Cette étude est menée sur des machines jusquà 2048 processeurs pour résoudre des problèmes $3D$ à plus de 50 millions d'inconnues. Ces études ont été menées sur des machines telles que le SystemX de Virginia Tech ou l'IBM Blue-Gene du CERFACS.

Une étude sur l'influence de la sparsification a illustré le comportement du préconditionneur creux en le comparant au dense. La Figure 4.6 montre que le préconditionneur creux peut-être considérré comme robuste et efficace. Autrement dit pour des très petites valeurs du paramètre de seuil, 20% des entrées du complément du Schur sont retenues . Dans ce contexte, on observe un gain en mémoire et en calcul considérable alors que les préconditionneurs denses et creux ont des convergences très similaires (courbe en rouge). Pour des valeurs optimales du paramètre de seuil, on observe un gain énorme aussi bien en mémoire qu'en calcul (on garde moins que 5% des entrées avec un gain d'un facteur 3 en temps). Ici le préconditionneur creux nécessite quelques itérations de plus pour converger mais chaque itération est significativement plus rapide (courbe en vert). Par contre pour des valeurs très grandes du paramètre de seuil, situation oú seulement 1% des entrées du complément de Schur sont conservées, la convergence se détériore pour un gain en temps de calcul qui n'est pas très significatif. Donc un choix optimal du paramètre de seuil doit assurer un bon compromis entre le coût de construction et d'application du préconditionneur tout en assurant une bonne convergence.

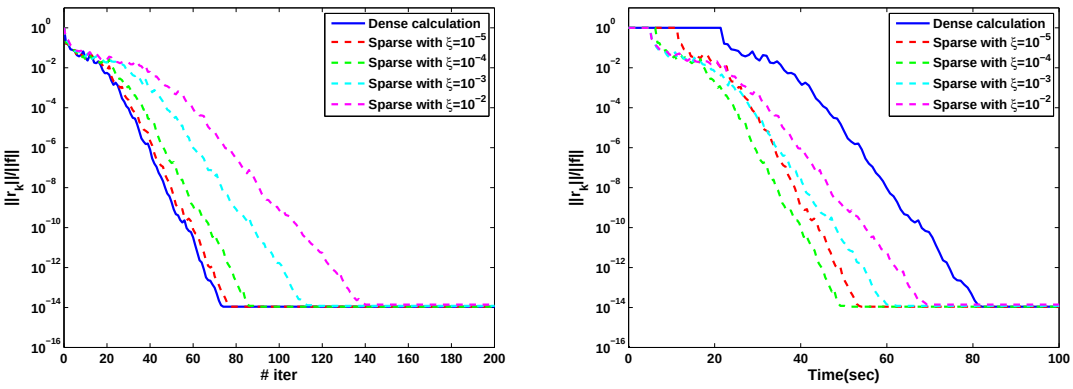


Figure 4.6: Comportement numérique de la variante creuse.

Suivant la même méthodologie, une étude sur l'effet de la précision mixte a été menée. La Figure 4.7 illustre une comparaison avec les préconditionneurs mixte et double précision. On peut tout d'abord observer que le préconditionneur en précision mixte atteint le même niveau de convergence que celui en double précision sans trop pénaliser la convergence. De plus en regardant le temps de calcul, on observe un gain acceptable. On note que ce gain varie d'une plateforme à une autre. Par

exemple sur des machines IBM SP4 on observe un facteur de 1.8 entre un calcul simple et double précision tandis que ce facteur n'est pas observé sur une machine BlueGene sur laquelle les deux arithmétiques sont traitées à la même vitesse. On note que seul le préconditionneur est calculé en simple précision. Donc un calcul en précision mixte nous apporte un gain d'un facteur 2 au niveau de stockage ainsi qu'un gain en temps de calcul dépendant de la machine de calcul.

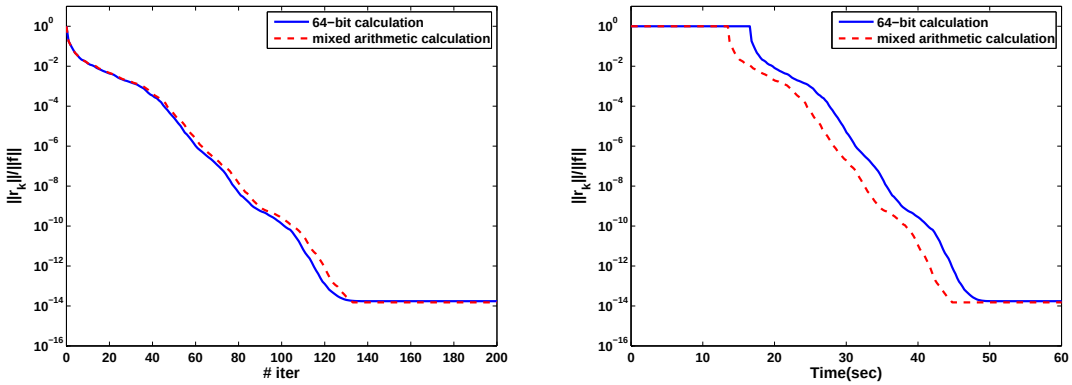


Figure 4.7: Comportement numérique de la variante précision mixte.

Dans une seconde étape, nous nous intéressons à lévolutivité aussi d'un point de vue numérique que d'un point de vue performance parallèle des préconditionneurs. Différentes analyses peuvent être réalisées pour étudier les performances parallèles d'une approche lorsqu'on augmente le nombre de processeurs. Dans notre cas, nous avons considéré une étude de l évolutivité où l'on fait varier linéairement la taille du problème traité en fonction du nombre de processeurs utilisés (scaled scalabilty en anglais). Pour un algorithme idéal d'un point de vue de son comportement numérique et de sa mise en oeuvre parallèle, le temps de restitution reste constant et indépendant du nombre de processeurs utilisés.

Dans ce contexte expérimental, nous illustrons dans la Figure 4.8 à gauche le nombre d'itérations nécessaires lorsqu'on augmente le nombre de processeurs tandis quà droite, on représente le temps de calcul nécessaire pour réaliser la simulation. Dun point de vue convergence, on peut observer que lorsqu'on augmente le nombre de processeurs de 27 à 1728, le nombre d'itérations n'augmente que de 23 à 60 ; autrement dit lorsqu'on augmente la taille du problème 64 fois le nombre d'itérations n'augmente que de 3 fois. Par contre d'un point de vue temps de calcul, on peut observer que lorsqu'on augmente la taille du problème 64 fois le temps de calcul n'est multiplié que par un facteur inférieur à 1.3 ce qui est proche de la situation idéale. En effet, la partie incompressible et commune à chacune de ses experimentations de factorisation des problèmes locaux et d'initialisation du préconditionneur constitue une part significative du calcul complet ; ceci masque partiellement l'augmentation du nombre d'itérations. Donc on peut conclure que cette méthode présente une évolutivité parallèle intéressante au niveau numérique ainsi qu'au niveau performance.

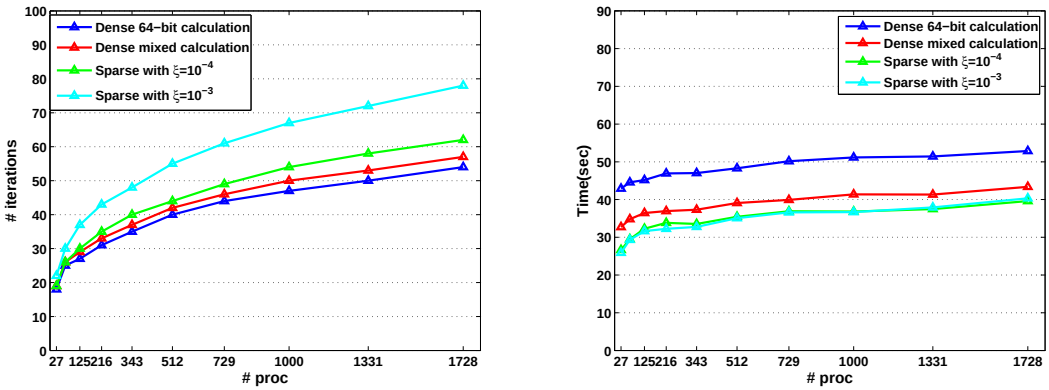


Figure 4.8: scalabilité numérique et paralléle performance.

Chapter 5

Numerical investigations on diffusion equations

5.1 Introduction

In this chapter, we first describe in Section 5.2 the computational framework and detail the academic model problems considered for our parallel numerical experiments. We investigate in Section 5.3 the numerical behaviours of the sparsified and mixed arithmetic variants that are compared with the classical dense 64-bit additive Schwarz preconditioner. Section 5.4 is the core of the parallel study where we first illustrate through classical speedup experiments the advantage of increasing the number of processors for solving a problem of a prescribed size; then we study the numerical scalability and the parallel performance of the preconditioners by conducing scaled speedup experiments where the problem size is increased linearly with the number of processors [51]. Finally we end this section by considering the effect of a two-level preconditioner.

5.2 Experimental environment

Although many runs have been performed on various parallel platforms, we only report in this chapter on experiments performed on the System X computer installed at Virginia Tech. This parallel distributed computer is a 1100 dual node Apple Xserve G5 cluster machine based on 2.3 GHz PowerPC 970FX processors with a 12.25 TFlops peak performance. This computer has a distributed memory architecture, where each node has 4 GBytes ECC DDR400 (PC3200) of RAM. Thus, data sharing among processors is performed using the message passing library MVAPICH. The interconnection networks between processors are 10 Gbps InfiniBand with 66 SilverStorms 9xx0 family switches and Gigabit Ethernet with 6 Cisco Systems 240-port 4506 switches.

To investigate the robustness and the scalability of the preconditioners we consider various academic 3D model problems by considering the diffusion coefficient matrix K in Equation (5.1) as diagonal with piecewise constant function entries defined in the unit cube as depicted in Figure 5.1. The diagonal entries $a(x,y,z)$, $b(x,y,z)$, $c(x,y,z)$ of K are bounded positive functions on Ω enabling us to define heterogeneous and/or anisotropic problems,

$$\left\{ \begin{array}{rcl} -\mathrm{div}(K.\nabla u) & = & f \quad \text{in} \quad \Omega, \\ u & = & 0 \quad \text{on} \quad \partial\Omega. \end{array} \right. \tag{5.1}$$

To vary the difficulties we consider both discontinuous and anisotropic PDE's where constant diffusion coefficients are defined either along vertical beams (Pattern 1 type problems) or horizontal beams (Pattern 2 type problems). This latter pattern corresponds to MOSFET problems arising in device modeling simulation. For the sake of completeness we also consider the classical Poisson

problem where all the coefficient functions a, b and c are identically one. More precisely we define the following set of problems:

Problem 1: Poisson where $a(\cdot) = b(\cdot) = c(\cdot) = 1$.

Problem 2: heterogeneous diffusion problem based on Pattern 1;

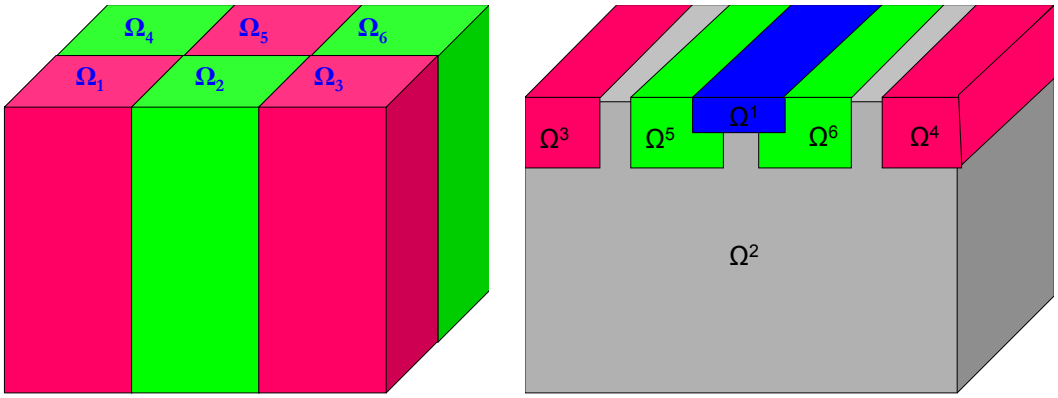
$$a(\cdot) = b(\cdot) = c(\cdot) = \begin{cases} 1 & \text{in} \quad \Omega^1 \cup \Omega^3 \cup \Omega^5, \\ 10^3 & \text{in} \quad \Omega^2 \cup \Omega^4 \cup \Omega^6. \end{cases}$$

Problem 3: heterogeneous and anisotropic diffusion problem based on Pattern 1; $a(\cdot) = 1$ and

$$b(\cdot) = c(\cdot) = \begin{cases} 1 & \text{in} \quad \Omega^1 \cup \Omega^3 \cup \Omega^5, \\ 10^3 & \text{in} \quad \Omega^2 \cup \Omega^4 \cup \Omega^6. \end{cases}$$

Problem 4: heterogeneous and anisotropic diffusion problem based on Pattern 2; $a(\cdot) = 1$ and

$$b(\cdot) = c(\cdot) = \begin{cases} 1 & \text{in} \quad \Omega^1, \\ 10^3 & \text{in} \quad \Omega^2, \\ 10^{-3} & \text{in} \quad \Omega^3 \cup \Omega^4 \cup \Omega^5 \cup \Omega^6. \end{cases}$$



(a) Pattern 1. (b) Pattern 2.

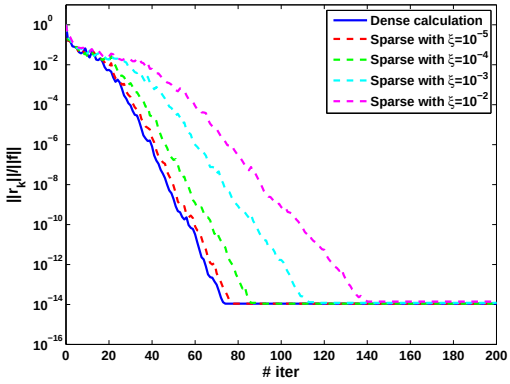
Figure 5.1: variable coefficient domains.

5.3 Numerical performance behaviour

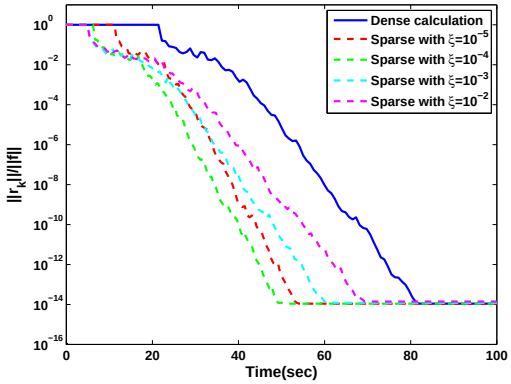
In this section we investigate the numerical behaviour of the sparsified and mixed arithmetic preconditioners and compare them to the classical $M_{\mathrm{d}-64}$. For that purpose, the performance and robustness of the preconditioners are evaluated for the different PDE problems using a prescribed mesh size with 43 millions unknowns solved using 1000 processors/subdomains. To this end we consider the convergence history of the normwise backward error $\frac{\|r_k\|}{\|f\|}$ along the iterations, where f denotes the right-hand side of the Schur complement system to be solved and r_k the true residual at the k^{th} iteration (i.e., $r_k = f - \mathcal{S}u_\Gamma^{(k)}$).

5.3.1 Influence of the sparsification threshold

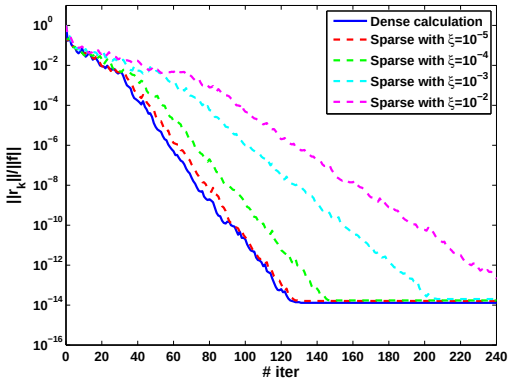
The attractive feature of $M_{\mathrm{sp}-64}$ compared to $M_{\mathrm{d}-64}$ is that it enables us to reduce both the memory requirement to store the preconditioner and the computational cost to construct it (dense versus sparse factorization). However, the counterpart of this computing resource saving could be a deterioration of the preconditioner quality that would slow down the convergence of PCG. In order to study the effect of the sparsification of the preconditioner on the convergence rate we display in Figure 5.2 and 5.3 the convergence history for various choices of the dropping parameter ξ involved in the definition of $M_{\mathrm{sp}-64}$ in Equation (3.6). On the left-hand side we display the convergence history as a function of the iterations. On the right-hand side, the convergence is given as a function of the computing time. In these latter graphs, the initial plateaus correspond to the setup time of the preconditioner. It can be observed that, even though they require more iterations, the sparsified variants converge faster as the time per iteration is smaller and the setup of the preconditioner is cheaper.



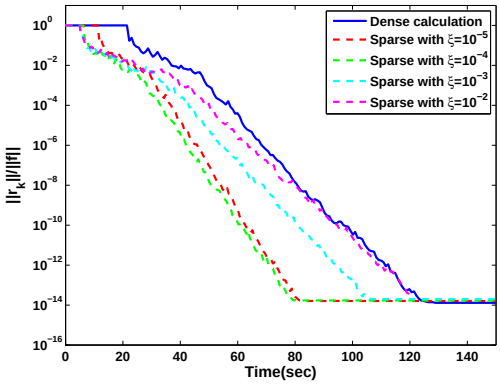
(a) Poisson problem (history v.s. iterations).



(b) Poisson problem (history v.s. time).



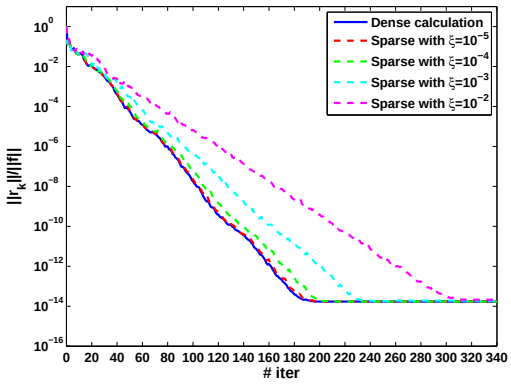
(c) Problem 2 (history v.s. iterations).



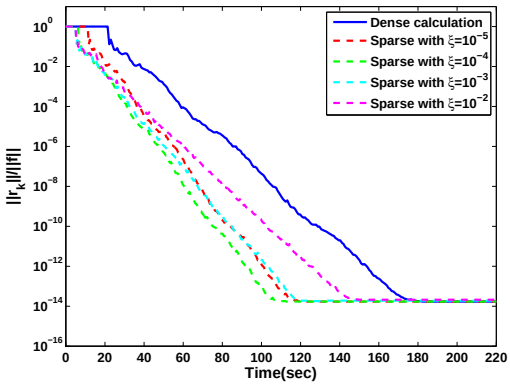
(d) Problem 2 (history v.s. time).

Figure 5.2: Convergence history for a $350 \times 350 \times 350$ mesh mapped onto 1000 processors for various dropping thresholds (Left: scaled residual versus iterations, Right: scaled residual versus time).

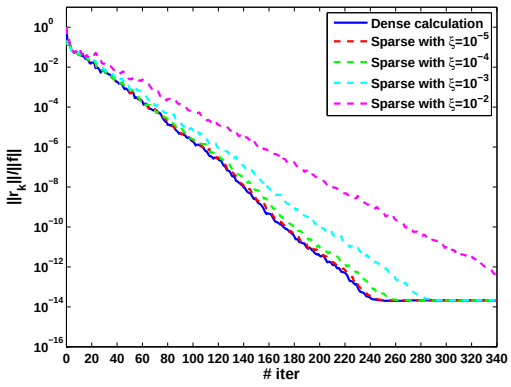
The trends that can be observed on these particular choices of problems (underlying PDE's: Poisson, Problem 2, Problem 3 and Problem 4; domain partition: $350 \times 350 \times 350$ mesh partitioned into 1000 subdomains) have been observed on many other examples. That is, for small values of the dropping parameter the convergence is marginally affected while the memory saving is



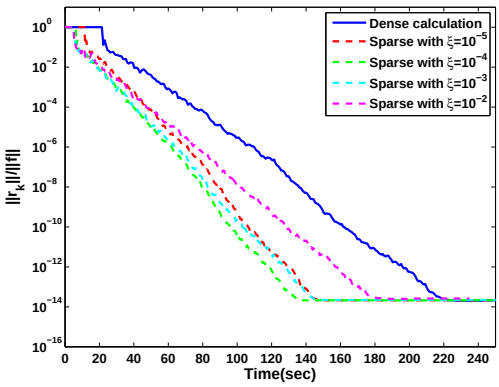
(a) Problem 3 (history v.s. iterations).



(b) Problem 3 (history v.s. time).



(c) Problem 4 (history v.s. iterations).



(d) Problem 4 (history v.s. time).

Figure 5.3: Convergence history for a $350 \times 350 \times 350$ mesh mapped onto 1000 processors for various dropping thresholds (Left: scaled residual versus iterations, Right: scaled residual versus time).

already significant; for larger values of the dropping parameter a lot of resources are saved in the construction of the preconditioner but the convergence becomes very poor. A reasonable trade-off between computing resource savings and convergence rate is generally for a choice of the dropping parameter equal to 10^{-4} that enables us to retain around 2% of the entries of the local Schur complements. This value for the dropping threshold is used in the rest of this chapter to define $M_{\mathrm{sp-64}}$ and $M_{\mathrm{sp-mix}}$.

For the various choices of this parameter the memory spaces required by the preconditioners on each processor are given in Table 5.1.

ξ	0	10^{-5}	10^{-4}	10^{-3}	10^{-2}
Memory	367_{MB}	29.3_{MB}	7.3_{MB}	1.4_{MB}	0.4_{MB}
Percentage	100%	8%	2%	0.4%	0.1%

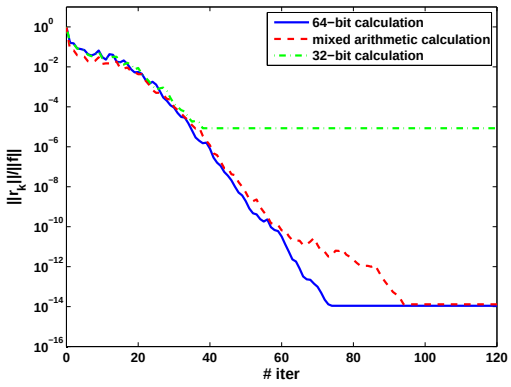
Table 5.1: Amount of memory in $M_{\mathrm{sp-64}}$ v.s. $M_{\mathrm{d-64}}$ for various choices of the dropping parameter.

5.3.2 Influence of the mixed arithmetic

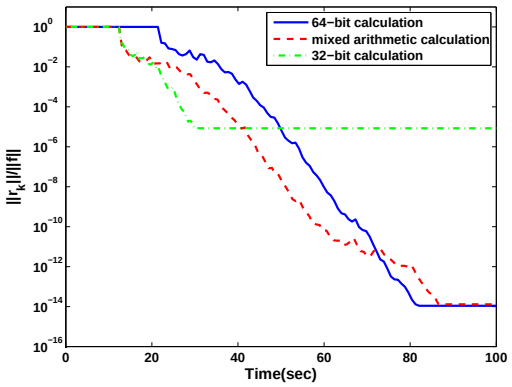
A distinctive framework feature of this work is the use of mixed-precision preconditioners in domain decomposition [50], where the 32-bit calculations are expected to significantly reduce not only the elapsed time of a simulation but also the memory required to implement the preconditioner. In that respect all but the preconditioning step are implemented in high precision. In our implementation all the PCG variables are 64-bit variables but the preconditioner and the preconditioned residual (denoted by z in Algorithm 3) are 32-bit variables. As in the previous section, the performance and robustness of the preconditioners are evaluated for the different PDE problems using a prescribed mesh size with 43 millions unknowns solved using 1000 processors/subdomains.

In order to compare the convergence rate of a fully 32-bit, a fully 64-bit, and a mixed-precision implementation, we depict in Figure 5.4 and 5.5 the convergence history for the three implementations. We display in Figure 5.4 (a) the convergence history as a function of the iterations for the Poisson problem, while Figure 5.4 (c) and Figure 5.5 (a) and (c) corresponds respectively to Problem 2, Problem 3 and Problem 4.

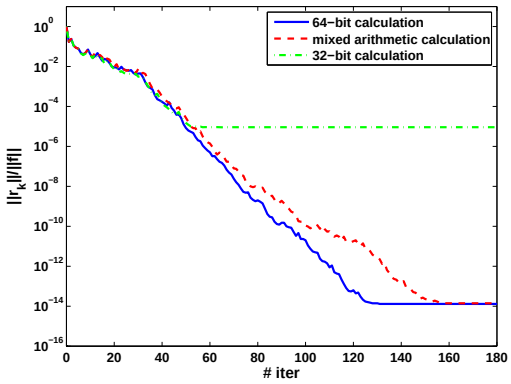
It can be observed that for these not too ill-conditioned problems, the 32-bit calculation of the preconditioning step does not delay too much the convergence of PCG. Down to the accuracy of about 32-bit machine precision, the three curves have very similar paths. As it could have been expected, the 32-bit implementation of CG reaches a limiting accuracy at the level of the single precision machine epsilon, while the full 64-bit and the mixed arithmetic implementations both attained an accuracy at the level of 64-bit machine precision. On the right-hand side of these figures we display the convergence history as a function of time. Again the initial plateaus correspond to the setup of the preconditioner. As could have been expected, down to the single precision machine precision, the 32-bit calculation is the fastest, then down to an accuracy (that is problem dependent), the mixed precision approach is the fastest. Finally, at even higher accuracy the 64-bit implementation outperforms the mixed one; the time saving per iteration is outweighed by the few extra iterations performed by the mixed approach. We should point out that the mixed strategy can only be considered for problems where the preconditioner is not too ill-conditioned (respectively, the initial problem is not too ill-conditioned) so that it is not singular in 32-bit arithmetic. Finally, we point out the current lack of theoretical results to explain the surprising numerical behaviour of the mixed arithmetic PCG.



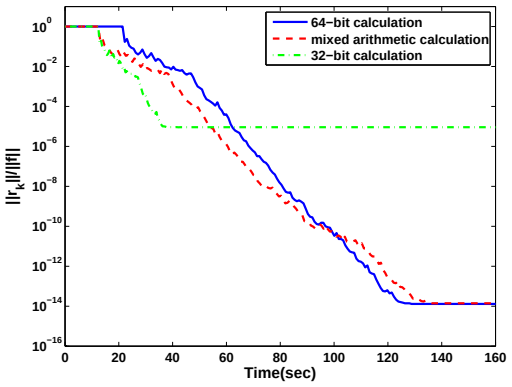
(a) Poisson problem (history v.s. iterations).



(b) Poisson problem (history v.s. time).

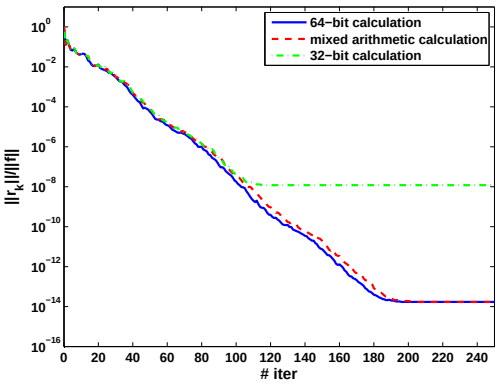


(c) Problem 2 (history v.s. iterations).

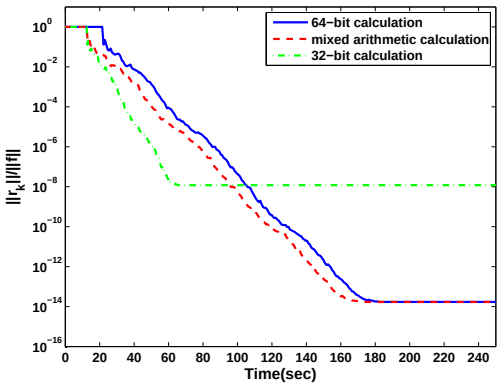


(d) Problem 2 (history v.s. time).

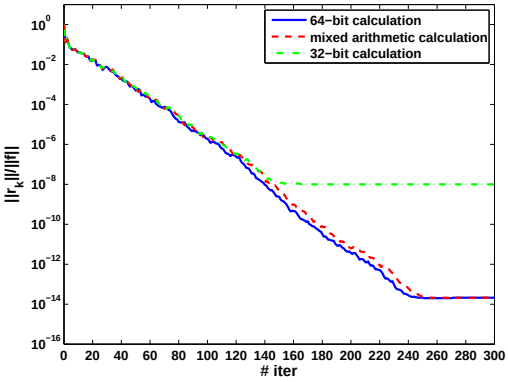
Figure 5.4: convergence history for a $350 \times 350 \times 350$ mesh mapped onto 1000 processors (Left: scaled residual versus iterations, Right: scaled residual versus time).



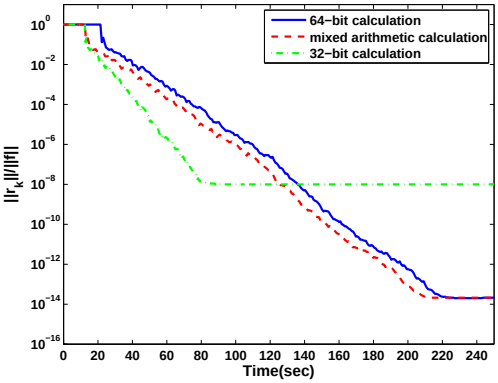
(a) Problem 3 (history v.s. iterations).



(b) Problem 3 (history v.s. time).



(c) Problem 4 (history v.s. iterations).



(d) Problem 4 (history v.s. time).

Figure 5.5: convergence history for a $350 \times 350 \times 350$ mesh mapped onto 1000 processors (Left: scaled residual versus iterations, Right: scaled residual versus time).

5.4 Parallel numerical scalability

For all the experiments reported in the this parallel scalability study the stopping criterion for the linear solver is based on the normwise backward error on the right-hand side. It is defined by

$$\frac{\|\bar{r}_k\|}{\|f\|} \leq 10^{-8},$$

where $\bar{r}_k$ is the residual computed by PCG (i.e., given by the recurrence) and f the right-hand side of the Schur complement system; the initial guess is always the zero vector. We first consider experiments where the size of the initial linear system (i.e., mesh size) is kept constant when the number of processors is varied. Such iso-problem size experiments mainly emphasize the interest of parallel computation in reducing the elapsed time to solve a problem of a prescribed size. We then perform scaled experiments where the problem size is varied linearly with the number of processors. Such iso-granularity experiments illustrate the ability of parallel computation in performing large simulations (fully exploiting the local memory of the distributed platform) in ideally a constant elapsed time. For all these experiments, each subdomain is allocated to one processor.

5.4.1 Parallel speedup experiments

In these experiments we consider both the Poisson problem and a heterogeneous anisotropic problem (Problem 4) discretized on a $211 \times 211 \times 211$ grid. The number of processors is varied from 216 to 1000 and Table 5.2 displays the corresponding parallel elapsed time, memory requirements, and number of iterations. The number of iterations carried out is given in the row headed "# iter". The size of each of the local Schur complement matrices is given in Mbytes in the row entitled "Memory". We also display the wall-clock time taken in carrying out the computation of the Schur complement, the construction and the factorization of the preconditioner ("Setup"), and the time taken by one CG iteration ("Time/iter"). The total time to solve the problem is given by "Total". Finally we give the speedup computed using the elapsed time on 216 processors as reference. We

	# subdomains ≡ # processors				
	216	343	512	729	1000
	Poisson problem				
Memory (MB) per processor	413	223	126	77	54
# iter	33	35	37	40	43
Time/iter	0.86	0.48	0.29	0.21	0.13
Setup	67.13	26.68	12.90	6.85	4.42
Total	95.45	43.63	23.84	15.47	10.15
speedup	1	2.18	4.00	6.17	9.40
Problem 4					
Memory (MB) per processor	413	223	126	77	54
# iter	155	184	210	237	246
Time/iter	0.88	0.51	0.28	0.18	0.13
Setup	68.73	26.60	12.81	6.80	4.58
Total	205.40	121.72	72.15	51.22	38.45
speedup	1	1.69	2.84	4.01	5.34

Table 5.2: Classical speedups - Performance on on a $211 \times 211 \times 211$ mesh when the number of processors is varied using $M_{\mathrm{d}-64}$ for both Poisson and Problem 4.

note that, for a fixed size problem, increasing the number of processors means decreasing the local sub-problem size. This leads to smaller local Schur complements but the global Schur system

becomes larger, which contributes to an increase in the number of iterations. We observe that the growth in the number of iterations is not significant while the reduction in the data storage is very important. The amount of data managed by each processor is smaller, providing a speedup for the BLAS-2 operations and BLAS-3 operations in the direct solvers. The size of the local problems becomes smaller and the sparse direct solver, the dense direct solver, used on the subdomains, to factorize the local problem and the assembled Schur respectively, becomes much faster. This produces a significant drop in the setup time and in the time per iteration. Thus one can easily conclude that the growth in the number of iterations was offset by decreasing the direct solver execution time and by reducing the amount of data. We notice that superlinear speedups (i.e., speedup larger than the increase in processor number) are observed for all these experiments. This is mainly due to the fact that when the number of processors is increased the size of the local subdomain decreases; because the complexity of the direct solvers is superlinear the saving of time is also superlinear.

5.4.2 Numerical scalability study on massively parallel platforms

In this section we study the numerical efficiency of the preconditioners. We perform scaled experiments where either

- the size of the subdomains is kept constant (i.e., $\frac{H}{h}$ constant where H is the diameter of the subdomains and h the mesh size) when the number of subdomains is increased;
- or the number of processors is kept fixed while increasing the size of the underlying subdomain mesh (i.e., $\frac{H}{h}$ varies).

5.4.2.1 Effect of the sparsification dropping threshold on the performance

In this section, we illustrate the effect of the sparsification strategy both on the convergence rate and the computation cost (memory and computing time). Although many experiments have been performed, we only report here numerical results obtained on Problem 2; similar behaviours have been observed on the other model problems. The size of the subdomains is equal to about 27,000 degrees of freedom and we vary the number of subdomains from 27 up to 1000 (i.e., varying the decomposition of the cube from $3 \times 3 \times 3$ up to $10 \times 10 \times 10$). In Table 5.3 we display the

		# subdomains ≡ # processors							
		27	64	125	216	343	512	729	1000
$\xi = 0$	# iterations	24	34	35	46	49	61	65	71
	elapsed time	39.2	46.9	47.8	56.2	58.9	68.8	72.5	79.4
$\xi = 10^{-5}$	# iterations	25	34	38	48	50	64	69	76
	elapsed time	26.3	30.1	32.0	36.5	37.3	43.7	45.8	50.2
$\xi = 10^{-4}$	# iterations	27	37	41	53	57	74	77	85
	elapsed time	24.6	28.7	30.7	36.0	37.7	45.4	46.6	51.7
$\xi = 10^{-3}$	# iterations	35	49	56	74	78	100	108	122
	elapsed time	26.9	32.7	36.0	43.9	45.6	55.6	58.9	67.3
$\xi = 10^{-2}$	# iterations	44	64	69	95	100	127	133	152
	elapsed time	30.2	38.4	41.0	52.4	54.5	66.8	69.3	80.3

Table 5.3: Number of preconditioned conjugate gradient iterations and corresponding elapsed time on Problem 2 when the number of subdomains and the dropping parameter ξ is varied.

number of iterations and the corresponding computing time to solve the associated linear system. It can be seen than smaller the dropping threshold ξ faster is the convergence ($\xi = 0$ reduces to dense calculation). Allowing more entries by decreasing dropping parameter close to zero generally

helps the convergence. On the other hand, larger the dropping faster is one PCG iteration as the cost of the preconditioner is smaller. The best trade-off, between memory saving to store the preconditioner and its ability to reduce the solution time is for $\xi = 10^{-4}$. For this choice of the dropping parameter we only retain around $1\% - 5\%$ of the entries of the dense preconditioning matrix.

5.4.2.2 Effect of the mixed arithmetic on the performance

In order to illustrate the effect on the convergence rate, we report in the tables below (Tables 5.4–5.7) the number of PCG iterations for the four considered preconditioners on various model problems and various local problem sizes. Recall that the threshold used to construct the sparse preconditioners $M_{\text{sp}-64}$ and $M_{\text{sp}-\text{mix}}$ is 10^{-4}, which enables us to retain around 2% of the entries of the local Schur complements. In these tables reading across a row shows the behaviour with fixed subdomain size when the number of the processors goes from 27 up to 1000 while the overall problem size increases; for every column the number of processors (subdomains) is kept constant while refining the mesh size within each subdomain. In optimal situations, numerical scalability would mean that the convergence rate would not depend on the number of subdomains; this would lead to constant computing time when the overall size of the problem and the number of processors increase proportionally.

		# subdomains ≡ # processors							
subdomain grid size		27	64	125	216	343	512	729	1000
$20 \times 20 \times 20$	$M_{\text{d}-64}$	16	23	25	29	32	35	39	42
	$M_{\text{d}-\text{mix}}$	18	24	26	31	34	38	41	46
	$M_{\text{sp}-64}$	16	23	26	31	34	39	43	46
	$M_{\text{sp}-\text{mix}}$	18	25	27	34	37	41	45	49
$25 \times 25 \times 25$	$M_{\text{d}-64}$	17	24	26	31	33	37	40	43
	$M_{\text{d}-\text{mix}}$	19	26	28	33	36	40	44	47
	$M_{\text{sp}-64}$	17	25	28	34	37	42	45	49
	$M_{\text{sp}-\text{mix}}$	19	26	29	36	41	44	48	53
$30 \times 30 \times 30$	$M_{\text{d}-64}$	18	25	27	32	34	39	42	45
	$M_{\text{d}-\text{mix}}$	20	27	29	34	38	41	48	49
	$M_{\text{sp}-64}$	18	26	29	36	40	44	48	52
	$M_{\text{sp}-\text{mix}}$	19	28	31	39	42	46	52	57
$35 \times 35 \times 35$	$M_{\text{d}-64}$	19	26	30	33	35	40	44	47
	$M_{\text{d}-\text{mix}}$	21	29	30	35	39	42	46	50
	$M_{\text{sp}-64}$	19	28	30	38	46	46	50	56
	$M_{\text{sp}-\text{mix}}$	21	30	33	41	44	49	54	59

Table 5.4: Number of preconditioned conjugate gradient iterations for the Poisson problem when the number of subdomains and the subdomain mesh size is varied.

Table 5.4 is devoted to experiments on the Poisson problem, Table 5.5 to Problem 2, and Tables 5.6, 5.7 reports respectively results on the heterogeneous and anisotropic Problem 3, Problem 4. We can first observe that the problems with both heterogeneity and anisotropy are the most difficult to solve and that the Poisson problem is the easiest.

For all the problems, the dependency of the convergence rate on the mesh size can be observed although it is moderated. This behaviour is similar for the four preconditioners. When we go from subdomains with about 8,000 degrees of freedom (dof) to subdomains with about 43,000 dof, the number of iterations can increase by over 25%. Notice that with such an increase in the subdomain size, the overall system size is multiplied by more than five; on 1000 processors the global system size varies from eight million dof up to about 43 million dof.

		# subdomains ≡ # processors							
subdomain grid size		27	64	125	216	343	512	729	1000
$20 \times 20 \times 20$	$M_{\mathrm{d}-64}$	22	32	34	41	45	55	60	67
	$M_{\mathrm{d}-\mathrm{mix}}$	23	33	37	44	48	58	63	70
	$M_{\mathrm{sp}-64}$	23	34	39	47	49	62	70	76
	$M_{\mathrm{sp}-\mathrm{mix}}$	24	35	40	48	52	64	70	79
$25 \times 25 \times 25$	$M_{\mathrm{d}-64}$	23	33	36	44	47	58	64	69
	$M_{\mathrm{d}-\mathrm{mix}}$	24	34	39	45	50	60	67	72
	$M_{\mathrm{sp}-64}$	25	34	41	50	53	67	74	82
	$M_{\mathrm{sp}-\mathrm{mix}}$	26	36	43	51	57	69	78	84
$30 \times 30 \times 30$	$M_{\mathrm{d}-64}$	24	34	35	46	49	61	65	71
	$M_{\mathrm{d}-\mathrm{mix}}$	25	35	38	47	52	64	69	74
	$M_{\mathrm{sp}-64}$	27	37	41	53	57	74	77	85
	$M_{\mathrm{sp}-\mathrm{mix}}$	28	39	44	57	61	76	82	92
$35 \times 35 \times 35$	$M_{\mathrm{d}-64}$	25	35	40	47	50	61	67	73
	$M_{\mathrm{d}-\mathrm{mix}}$	25	37	42	49	54	65	70	77
	$M_{\mathrm{sp}-64}$	28	41	45	56	60	74	84	90
	$M_{\mathrm{sp}-\mathrm{mix}}$	29	43	49	59	64	80	88	96

Table 5.5: Number of preconditioned conjugate gradient iterations for Problem 2 when the number of subdomains and the subdomain mesh size is varied.

		# subdomains ≡ # processors							
subdomain grid size		27	64	125	216	343	512	729	1000
$20 \times 20 \times 20$	$M_{\mathrm{d}-64}$	39	56	67	87	90	104	123	132
	$M_{\mathrm{d}-\mathrm{mix}}$	45	58	69	91	94	108	126	135
	$M_{\mathrm{sp}-64}$	39	57	69	90	92	106	126	134
	$M_{\mathrm{sp}-\mathrm{mix}}$	42	59	71	93	96	111	129	139
$25 \times 25 \times 25$	$M_{\mathrm{d}-64}$	43	57	70	91	93	106	125	138
	$M_{\mathrm{d}-\mathrm{mix}}$	48	61	73	94	97	111	131	142
	$M_{\mathrm{sp}-64}$	44	60	73	94	97	112	131	143
	$M_{\mathrm{sp}-\mathrm{mix}}$	45	63	76	98	101	116	135	148
$30 \times 30 \times 30$	$M_{\mathrm{d}-64}$	44	60	71	93	95	109	129	138
	$M_{\mathrm{d}-\mathrm{mix}}$	50	63	74	96	99	114	136	143
	$M_{\mathrm{sp}-64}$	45	63	75	99	100	118	139	145
	$M_{\mathrm{sp}-\mathrm{mix}}$	47	65	78	103	104	121	140	151
$35 \times 35 \times 35$	$M_{\mathrm{d}-64}$	44	62	72	94	96	111	131	137
	$M_{\mathrm{d}-\mathrm{mix}}$	52	65	76	97	101	115	136	142
	$M_{\mathrm{sp}-64}$	46	66	77	102	105	120	141	149
	$M_{\mathrm{sp}-\mathrm{mix}}$	49	69	80	106	108	126	145	155

Table 5.6: Number of preconditioned conjugate gradient iterations for the heterogeneous and anisotropic Problem 3 when the number of subdomains and the subdomain mesh size is varied.

subdomain grid size		# subdomains ≡ # processors							
		27	64	125	216	343	512	729	1000
$20 \times 20 \times 20$	$M_{\mathrm{d}-64}$	49	69	81	110	127	152	156	174
	$M_{\mathrm{d}-\mathrm{mix}}$	51	71	85	116	132	158	160	179
	$M_{\mathrm{sp}-64}$	50	69	84	111	131	154	159	177
	$M_{\mathrm{sp}-\mathrm{mix}}$	52	72	87	116	132	157	163	181
$25 \times 25 \times 25$	$M_{\mathrm{d}-64}$	52	72	85	114	129	154	162	178
	$M_{\mathrm{d}-\mathrm{mix}}$	55	76	89	119	134	162	171	183
	$M_{\mathrm{sp}-64}$	53	74	89	116	136	158	168	184
	$M_{\mathrm{sp}-\mathrm{mix}}$	56	77	92	121	138	166	174	188
$30 \times 30 \times 30$	$M_{\mathrm{d}-64}$	54	75	88	118	132	158	167	180
	$M_{\mathrm{d}-\mathrm{mix}}$	56	79	91	122	140	163	175	186
	$M_{\mathrm{sp}-64}$	55	77	92	121	146	164	173	189
	$M_{\mathrm{sp}-\mathrm{mix}}$	58	81	96	125	143	172	180	194
$35 \times 35 \times 35$	$M_{\mathrm{d}-64}$	55	77	89	120	133	158	169	183
	$M_{\mathrm{d}-\mathrm{mix}}$	57	80	92	126	141	166	178	188
	$M_{\mathrm{sp}-64}$	58	81	96	124	148	167	177	195
	$M_{\mathrm{sp}-\mathrm{mix}}$	60	84	99	129	147	175	187	200

Table 5.7: Number of preconditioned conjugate gradient iterations for the heterogeneous and anisotropic Problem 4 when the number of subdomains and the subdomain mesh size is varied.

None of the preconditioners implements any coarse space correction to account for the global coupling of the elliptic PDE's, hence they do not scale perfectly when the number of subdomains is increased. However, the numerical scalability is not that bad and clearly much better than that observed on two dimensional examples [25]. The number of iterations is multiplied by about two to 3.5 when going from 27 to 1000 processors (i.e., multiplying by about 40 the number of processors). The trend of the growth is similar for all the problems and is comparable for all the variants of the preconditioners on a given problem.

5.4.3 Parallel performance scalability on massively parallel platforms

In the sequel, we mainly report experiments with fixed subdomain size of about 43,000 degrees of freedom while increasing the number of processors from 125 to 1000. We look at the scaled experiments from a parallel elapsed time perspective considering the overall elapsed time to solve the problems as well as the elapsed times for each individual step of the solution process. These steps are initialization, preconditioner setup, and the iterative loop.

- The initialization phase, referred to as initialization, is shared by all the implementations. It corresponds to the time for factorizing the matrix associated with the local Dirichlet problem and constructing the local Schur complement using the MUMPS package. This phase also includes the final solution for the internal dof, once the interface problem has been solved (i.e., solution of the local Dirichlet problem). The computational cost of this initialization phase is displayed in Table 5.8 for various subdomain sizes.
- The preconditioner setup time is the time required to build the preconditioner, which is the time for assembling the local assembled Schur matrix, using neighbour to neighbour communication, and factorizing this local assembled Schur matrix using LAPACK for $M_{\mathrm{d}-64}$ and $M_{\mathrm{d}-\mathrm{mix}}$, or first sparsifying and then factorizing using MUMPS for $M_{\mathrm{sp}-64}$ and $M_{\mathrm{sp}-\mathrm{mix}}$. The elapsed time for various problems sizes are reported in Table 5.9.
- The iterative loop is the PCG iterations.

The initialization times, displayed in Table 5.8, are independent of the number of subdomains and only depend on their size. We can again observe the nonlinear cost of the direct solver with respect to the problem size. This nonlinear behaviour was the main origin of the superlinear speedups observed in the iso-problem size experiments in Section 5.4.1.

Subdomain grid size	$20 \times 20 \times 20$	$25 \times 25 \times 25$	$30 \times 30 \times 30$	$35 \times 35 \times 35$
Time	1.3	4.2	11.2	26.8

Table 5.8: Initialization time (sec).

The setup time to build the preconditioner is reported in Table 5.9. We should mention that the assembly time does not depend much on the number of processors (because the maximum communication is performed among 26 neighbours for the internal subdomains), but rather on whether 64-bit or 32-bit calculation is used. Using a mixed approach enables a reduction by a factor around 1.7 for the dense variants and 1.3 for the sparse ones. Larger savings are observed when dense and sparse variants are compared, the latter being about three times faster.

Subdomain grid size	$M_{\mathrm{d}-64}$	$M_{\mathrm{d}-\mathrm{mix}}$	$M_{\mathrm{sp}-64}$	$M_{\mathrm{sp}-\mathrm{mix}}$
$20 \times 20 \times 20$	0.93	0.56	0.50	0.23
$25 \times 25 \times 25$	3.05	1.85	1.64	1.15
$30 \times 30 \times 30$	8.73	4.82	3.51	3.01
$35 \times 35 \times 35$	21.39	12.36	6.22	4.92

Table 5.9: Preconditioner setup time (sec).

# processors	125	216	343	512	729	1000
$M_{\mathrm{d}-64}$	0.76	0.76	0.77	0.78	0.79	0.82
$M_{\mathrm{d}-\mathrm{mix}}$	0.73	0.75	0.75	0.76	0.77	0.80
$M_{\mathrm{sp}-64}$	0.40	0.41	0.41	0.42	0.42	0.44
$M_{\mathrm{sp}-\mathrm{mix}}$	0.39	0.39	0.40	0.41	0.42	0.43

Table 5.10: Parallel elapsed time for one iteration of the preconditioned conjugate gradient (sec).

In Table 5.10 we illustrate the elapsed time scalability of the parallel PCG iteration. In that table we only give times for 43,000 dof subdomains. The first observation is that the parallel implementation of the preconditioned conjugate gradient method scales almost perfectly as the time per iteration is nearly constant and does not depend much on the number of processors (i.e, 0.76 seconds on 125 processors and 0.82 on 1000 processors for $M_{\mathrm{d}-64}$). The main reason for this scalable behaviour is the efficiency of the global reduction involved in the dot product calculation that does not depend much on the number of processors; all the other communications are neighbour to neighbour and their costs do not depend on the number of processors. Furthermore, the relative cost of the reduction is negligible compared to the other steps of the algorithm. It can be observed that 32-bit arithmetic does not reduce much the time per iteration for both $M_{\mathrm{d}-\mathrm{mix}}$ and $M_{\mathrm{sp}-\mathrm{mix}}$ in comparison with the 64-bit ones $M_{\mathrm{d}-64}$ and $M_{\mathrm{sp}-64}$, respectively. This is due to the fact that the ratio of the 64-bit to the 32-bit forward/backward substitution time is only about 1.13 (compared to almost two for the factorization involved in the setup of the preconditioner phase). This reduces the impact of the 32-bit calculation in the preconditioning step and makes the time per iteration for both full 64-bit and mixed arithmetic very similar. Finally, it is clear that the sparsified variants are of great interest as they cut in half the time per iteration compared to their dense counterparts. Applying the sparsified preconditioners is almost twice faster than using the dense ones.

A comparison of the overall solution times is given in Table 5.11 for the standard Poisson problem, in Table 5.12 for Problem 2, and in Table 5.13 and Table 5.14 for the two heterogeneous and anisotropic problems. The block row "Total" is the parallel elapsed time for the complete solution of the linear system. It corresponds to the sum of the times for all the steps of the algorithm, which are the initialization, the setup of the preconditioner, and the iterative loop. We notice that the row "Total" permits us to evaluate the parallel scalability of the complete methods (i.e., combined numerical and parallel behaviour); the time should remain constant for perfectly scalable algorithms. It can be seen that the growth in the solution time is rather moderate, when the number of processors grows from 125 (about 5.3 million unknowns) to 1000 (about 43 million unknowns). Although the methods do not scale well numerically, their parallel elapsed time performances scale reasonably well. The ratio of the total elapsed time for running on 1000 processors to the time on 125 processors is about 1.22 for $M_{\mathrm{d}-64}$ and around 1.28 for the other three variants for the Poisson problem. These ratios are larger for the more difficult problems as the number of iterations grows more.

For the Poisson problem represented in Table 5.11, we observe that the most expensive kernels are the initialization and the preconditioner setup. Thus for the two mixed arithmetic algorithms $M_{\mathrm{d}-\mathrm{mix}}$ and $M_{\mathrm{sp}-\mathrm{mix}}$ the slight increase in the number of iterations that introduces a slight increase in the elapsed time for the iterative loop is swiftly covered by the vast reduction in the preconditioner setup time, especially for the dense mixed preconditioner $M_{\mathrm{d}-\mathrm{mix}}$. Therefore, the mixed arithmetic algorithms outperform the 64-bit ones in terms of overall computing time. By looking at the sparsified variants we observe a considerable reduction in the time per iteration and in the preconditioner setup time induced by the use of the sparse alternatives. Since the use of these variants only introduces a few extra iterations compared to their dense counterparts, this time reduction per iteration is directly reflected in a significant reduction of the total time.

	Total solution time					
# processors	125	216	343	512	729	1000
$M_{\mathrm{d}-64}$	71.0	73.3	75.1	79.4	83.0	86.7
$M_{\mathrm{d}-\mathrm{mix}}$	61.1	65.4	68.4	71.1	74.6	79.2
$M_{\mathrm{sp}-64}$	45.0	48.6	51.9	52.3	54.0	57.7
$M_{\mathrm{sp}-\mathrm{mix}}$	44.6	47.7	49.3	51.8	54.4	57.1
	Time in the iterative loop					
# processors	125	216	343	512	729	1000
$M_{\mathrm{d}-64}$	22.8	25.1	26.9	31.2	34.8	38.5
$M_{\mathrm{d}-\mathrm{mix}}$	21.9	26.2	29.2	31.9	35.4	40.0
$M_{\mathrm{sp}-64}$	12.0	15.6	18.9	19.3	21.0	24.6
$M_{\mathrm{sp}-\mathrm{mix}}$	12.9	16.0	17.6	20.1	22.7	25.4
	# iteration					
# processors	125	216	343	512	729	1000
$M_{\mathrm{d}-64}$	30	33	35	40	44	47
$M_{\mathrm{d}-\mathrm{mix}}$	30	35	39	42	46	50
$M_{\mathrm{sp}-64}$	30	38	46	46	50	56
$M_{\mathrm{sp}-\mathrm{mix}}$	33	41	44	49	54	59

Table 5.11: Parallel elapsed time for the solution of the Poisson problem (sec).

The performances on Problem 2 are displayed in Table 5.12. The results show that the most time consuming part is the iterative loop. We see that the time saved in the preconditioner setup by the use of mixed-precision arithmetic still compensates for a slight increase in the number of iterations. Consequently on the heterogeneous diffusion problem the mixed-precision algorithm outperforms the 64-bit one. If we now look at the sparsified variant, the tremendous reductions in

both the time per iteration (two times faster than the dense one) and the preconditioner setup time (three times faster than the dense one) offset the gap in the number of iterations. Consequently, the sparse alternatives clearly outperform the dense ones. Similar comments can be made for the performances on Problems 3 and Problems 4 as shown in Table 5.13 and Table 5.14. In all the experiments the sparsified versions outperform their dense counterparts and the mixed sparse variant often gives the fastest scheme.

	Total solution time					
# processors	125	216	343	512	729	1000
$M_{\mathrm{d-64}}$	78.6	83.9	86.7	95.8	101.1	108.0
$M_{\mathrm{d-mix}}$	69.8	75.9	79.7	88.6	93.1	100.8
$M_{\mathrm{sp-64}}$	51.0	56.0	57.6	64.1	68.3	72.6
$M_{\mathrm{sp-mix}}$	50.8	54.7	57.3	64.5	68.7	73.0
	Time in the iterative loop					
# processors	125	216	343	512	729	1000
$M_{\mathrm{d-64}}$	30.4	35.7	38.5	47.6	52.9	59.9
$M_{\mathrm{d-mix}}$	30.7	36.8	40.5	49.4	53.9	61.6
$M_{\mathrm{sp-64}}$	18.0	23.0	24.6	31.1	35.3	39.6
$M_{\mathrm{sp-mix}}$	19.1	23.0	25.6	32.8	37.0	41.3
	# iteration					
# processors	125	216	343	512	729	1000
$M_{\mathrm{d-64}}$	40	47	50	61	67	73
$M_{\mathrm{d-mix}}$	42	49	54	65	70	77
$M_{\mathrm{sp-64}}$	45	56	60	74	84	90
$M_{\mathrm{sp-mix}}$	49	59	64	80	88	96

Table 5.12: Parallel elapsed time for the solution of Problem 2 (sec).

	Total solution time					
# processors	125	216	343	512	729	1000
$M_{\mathrm{d-64}}$	102.9	119.6	122.1	134.8	151.7	160.5
$M_{\mathrm{d-mix}}$	94.6	111.9	114.9	126.6	143.9	152.8
$M_{\mathrm{sp-64}}$	63.8	74.8	76.1	83.4	92.2	98.6
$M_{\mathrm{sp-mix}}$	62.9	73.1	74.9	83.4	92.6	98.4
	Time in the iterative loop					
# processors	125	216	343	512	729	1000
$M_{\mathrm{d-64}}$	54.7	71.4	73.9	86.6	103.5	112.3
$M_{\mathrm{d-mix}}$	55.5	72.8	75.8	87.4	104.7	113.6
$M_{\mathrm{sp-64}}$	30.8	41.8	43.0	50.4	59.2	65.6
$M_{\mathrm{sp-mix}}$	31.2	41.3	43.2	51.7	60.9	66.7
	# iteration					
# processors	125	216	343	512	729	1000
$M_{\mathrm{d-64}}$	72	94	96	111	131	137
$M_{\mathrm{d-mix}}$	76	97	101	115	136	142
$M_{\mathrm{sp-64}}$	77	102	105	120	141	149
$M_{\mathrm{sp-mix}}$	80	106	108	126	145	155

Table 5.13: Parallel elapsed time to solve the heterogeneous and anisotropic Problem 3 (sec).

	Total solution time					
# processors	125	216	343	512	729	1000
$M_{\mathrm{d-64}}$	115.8	139.4	150.6	171.4	181.7	198.2
$M_{\mathrm{d-mix}}$	106.3	133.7	144.9	165.3	176.2	189.6
$M_{\mathrm{sp-64}}$	71.4	83.9	93.7	103.2	107.4	118.8
$M_{\mathrm{sp-mix}}$	70.3	82.0	90.5	103.5	110.3	117.7
	Time in the iterative loop					
# processors	125	216	343	512	729	1000
$M_{\mathrm{d-64}}$	67.6	91.2	102.4	123.2	133.5	150.1
$M_{\mathrm{d-mix}}$	67.2	94.5	105.8	126.2	137.1	150.4
$M_{\mathrm{sp-64}}$	38.4	50.8	60.7	70.1	74.3	85.8
$M_{\mathrm{sp-mix}}$	38.6	50.3	58.8	71.8	78.5	86.0
	# iteration					
# processors	125	216	343	512	729	1000
$M_{\mathrm{d-64}}$	89	120	133	158	169	183
$M_{\mathrm{d-mix}}$	92	126	141	166	178	188
$M_{\mathrm{sp-64}}$	96	124	148	167	177	195
$M_{\mathrm{sp-mix}}$	99	129	147	175	187	200

Table 5.14: Parallel elapsed time to solve the heterogeneous and anisotropic Problem 4 (sec).

Subdomain grid size	$M_{\mathrm{d-64}}$	$M_{\mathrm{d-mix}}$	$M_{\mathrm{sp-64}}$	$M_{\mathrm{sp-mix}}$
$20 \times 20 \times 20$	35.8_{MB}	17.9_{MB}	1.8_{MB} (5%)	0.9_{MB} (5%)
$25 \times 25 \times 25$	91.2_{MB}	45.6_{MB}	2.7_{MB} (3%)	1.3_{MB} (3%)
$30 \times 30 \times 30$	194.4_{MB}	97.2_{MB}	3.8_{MB} (2%)	1.6_{MB} (2%)
$35 \times 35 \times 35$	367.2_{MB}	183.6_{MB}	7.3_{MB} (2%)	3.6_{MB} (2%)

Table 5.15: Local data storage for the four preconditioners.

We now examine the four variants from a memory requirement perspective. For that, we depict in Table 5.15 the maximal amount of memory required on each processor of the parallel platform. We report in each column of Table 5.15 the size in megabytes for a preconditioner when the size of the subdomains is varied. For the sparse variants, we give in parenthesis the percentage of retained entries. These figures indicate that both the mixed arithmetic approach and the sparse variant reduce significantly the memory usage. A feature of the sparse variants is that they reduce the memory usage dramatically. The mixed precision strategy cut in half the required data storage, which has a considerable effect in terms of computing system operation and execution time, and also cut in half the time for assembling the local Schur matrix, due to halving the total neighbour to neighbour subdomain communication.

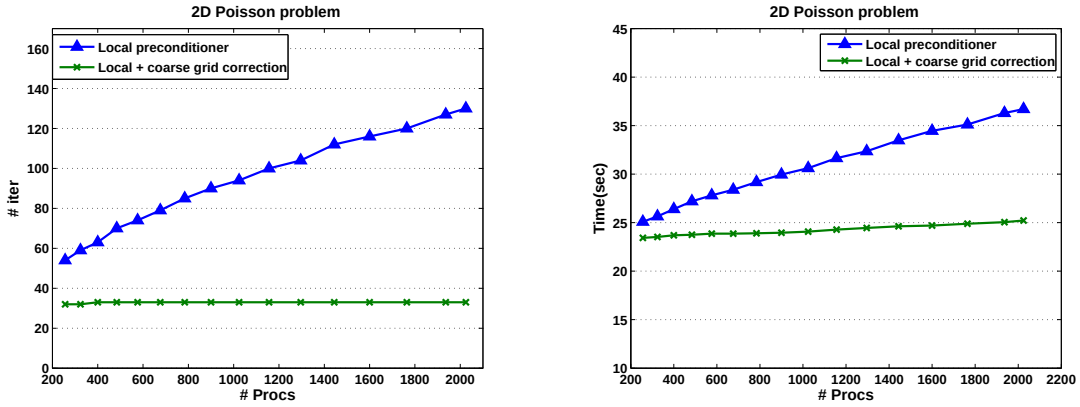
5.4.4 Influence of the coarse component correction

Although these preconditioners are local, consequently not numerically scalable, they exhibit a fairly good parallel time scalability as the relative cost of the setup partially hides the moderate increase in the number of iterations. A possible remedy to overcome this lack of numerical scalability is to introduce a coarse grid component. To illustrate the ability of our preconditioners to act efficiently as the local component of a two-level scheme, we consider a simple two-level preconditioner obtained by adding an additional term to them. For our experiments, the coarse space component extracts one degree of freedom per subdomain as described in Section 3.5 [26]. We start briefly by some experiments for $2D$ problems and show the effect of the coarse grid correction. For these experiments, we consider two model problems:

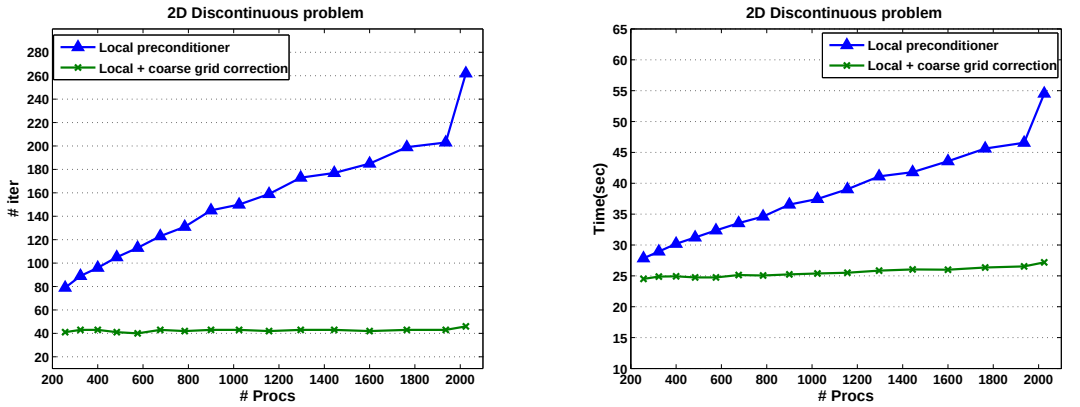
- the Poisson Problem 1,
- the discontinuous Problem 2 described in Section 5.2 where the jump between coefficient is in the xy plan.

	Total solution time							
# processors	256	400	676	1024	1296	1600	1764	2025
$M_{\mathrm{d}-64}$	27.8	30.2	33.5	37.5	41.1	43.6	45.6	54.5
$M_{\mathrm{d}-64}$+*coarse*	24.5	24.9	25.1	25.4	25.8	26.0	26.4	27.2
	Time for coarse setup							
# processors	256	400	676	1024	1296	1600	1764	2025
$M_{\mathrm{d}-64}$	–	–	–	–	–	–	–	–
$M_{\mathrm{d}-64}$+*coarse*	0.44	0.45	0.52	0.57	0.68	0.80	0.89	1.05
	Time in the iterative loop							
# processors	256	400	676	1024	1296	1600	1764	2025
$M_{\mathrm{d}-64}$	8.7	11.0	14.4	18.3	22.0	24.4	26.5	35.4
$M_{\mathrm{d}-64}$+*coarse*	4.9	5.3	5.5	5.7	6.0	6.0	6.3	7.0
	# iterations							
# processors	256	400	676	1024	1296	1600	1764	2025
$M_{\mathrm{d}-64}$	79	96	123	150	173	185	199	262
$M_{\mathrm{d}-64}$+*coarse*	41	43	43	43	43	42	43	46
	Time per iteration							
# processors	256	400	676	1024	1296	1600	1764	2025
$M_{\mathrm{d}-64}$	0.110	0.115	0.117	0.122	0.127	0.132	0.133	0.135
$M_{\mathrm{d}-64}$+*coarse*	0.120	0.124	0.127	0.132	0.140	0.144	0.147	0.152

Table 5.16: Performance of a parallel two-level preconditioner on Problem 2 using a 600×600 subdomain mesh.



(a) Poisson problem.



(b) Discontinuous Problem 2.

Figure 5.6: The number of PCG iterations (left) and the computing time (right) for a 600×600 grid when varying the number of subdomains from 256 to 2025.

In Figure 5.6, we report on the number of preconditioned conjugate gradient iterations (left) and the computing time (right) for each model problem. Its detailed performance on Problem 2 is displayed in Table 5.16. For these tests, we vary the number of subdomains while keeping constant their sizes (i.e., H variable with $\frac{H}{h}$ constant). In these experiments each subdomain is a 600×600 grid and the number of subdomains goes from 256 up to 2025 using a box decomposition; that is 16×16 decomposition up to 45×45 decomposition. Notice that with the increase in the number of subdomains, the overall system size is multiplied by about eight; that is the global system size varies from 92 million dof on 256 processors up to about 729 million dof on 2025 processors. The left graphs show the growth of the number of iterations of the local preconditioner without any coarse grid correction (the blue line with triangular). Ultimately one wants to compare the green line with x with the blue line with triangle. One can see that the performance of the two-level preconditioner (green line with x) comes close to the ideal numerical scalability. That is, the number of iterations stagnates close to 33 for Poisson problem and to 43 for discontinuous Problem 2, whereas for the standalone local preconditioner (the blue line with triangular) the growth in the number of iterations is notable. The graphs on the right gives the global computing time to solve the linear system. The scalability is also observed when the coarse component is introduced, the computing time remains constant when varying the number of subdomains from 256 to 2025. The solution of the coarse problem is negligible compared to the solution of the local Dirichlet problems. Finally for the $2D$ case the method scale well, when the number of processors grows from 256 (to solve a problem with 79 million unknowns) up to 2025 (to solve a problem with 729 million unknowns). The ratios between the total elapsed time expended for running on 2025 and on 256 processors is 1.09.

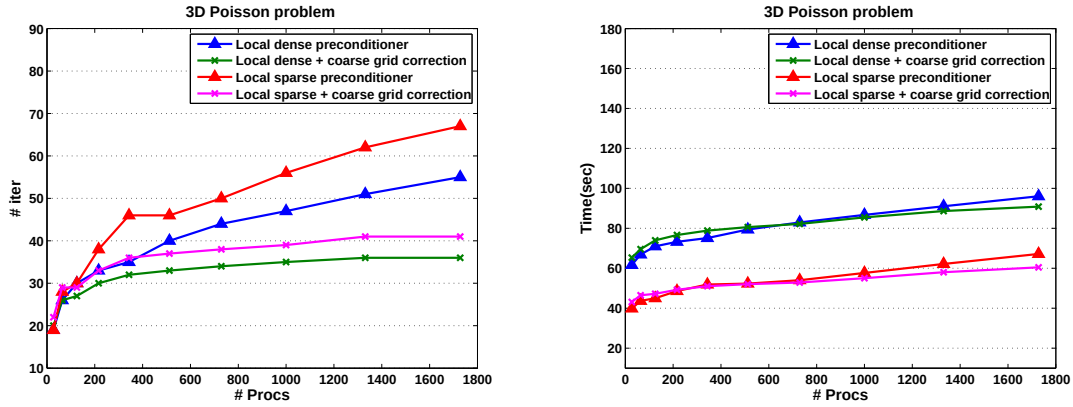
The behaviour is slightly different in $3D$. We consider two similar model problems as for the $2D$ case. Furthermore, we illustrate the effect of the coarse space correction in combination with the sparse preconditioner $M_{\text{sp}-64}$. In Figure 5.7, we report the number of preconditioned conjugate gradient iterations (left) and the computing time (right) for each model problem. Its detailed performance on Problem 2 is displayed in Table 5.17 for the combination with the dense $M_{\text{d}-64}$ preconditioner, and in Table 5.18 for the combination with the sparse $M_{\text{sp}-64}$ preconditioner. For these tests, we vary the number of subdomains while keeping constant their sizes (i.e., H variable with $\frac{H}{h}$ constant). In these experiments each subdomain is a $35 \times 35 \times 35$ grid and the number of subdomains goes from 27 up to 1728. We can notice that the setup time for the coarse space component is larger in $3D$ compared to $2D$ for comparable number of subdomains. This is mainly due to the fact that the local Schur matrices are larger in $3D$ and the number of coarse degree of freedoms that touch a subdomain is also higher; requiring more matrix-vector product to compute $\mathcal{S}_0$. We recall that the overall system size varies from 1.1 million dof on 27 processors up to about 75 million dof on 1728 processors. The left graphs show the growth of the number of iterations of the local preconditioner without any coarse grid correction (the blue and the red line with triangular). We observe that the coarse grid correction significantly alleviates the growth in the number of iterations when the number of subdomains is increased (the green and the magenta line with x). On 1728 processors, almost half the number of iterations are saved. The numbers of iterations with the two-level preconditioner tends to be asymptotic stable for these problems. The coarse component gives rise to preconditioners that are independent of, or weakly dependent on, the number of subdomains. We note that for the $3D$ case the convergence of all the local preconditioners depends slightly on the number of subdomains. In other term the gap in the number of iterations between the local preconditioner and the two-level one is less impressive on the $3D$ problems in comparison with the $2D$ problems for similar number of subdomains. Furthermore, the saving of iterations does not directly translate into time savings (the right graphs). We observe that each iteration becomes marginally more expensive, but the dominating part is clearly spent in the setup.

	Total solution time							
# processors	125	216	343	512	729	1000	1331	1728
$M_{\mathrm{d}-64}$	78.6	83.9	86.7	95.8	101.1	108.0	112.0	123.9
$M_{\mathrm{d}-64}$+*coarse*	80.2	87.2	86.0	91.5	94.2	97.9	100.8	103.5
	Time for coarse setup							
# processors	125	216	343	512	729	1000	1331	1728
$M_{\mathrm{d}-64}$	–	–	–	–	–	–	–	–
$M_{\mathrm{d}-64}$+*coarse*	2.04	2.05	2.20	2.38	2.76	3.68	4.12	4.89
	Time in the iterative loop							
# processors	125	216	343	512	729	1000	1331	1728
$M_{\mathrm{d}-64}$	30.4	35.7	38.5	47.6	52.9	59.9	63.8	75.7
$M_{\mathrm{d}-64}$+*coarse*	29.9	37.0	35.6	41.0	43.2	46.1	48.5	50.4
	# iterations							
# processors	125	216	343	512	729	1000	1331	1728
$M_{\mathrm{d}-64}$	40	47	50	61	67	73	76	87
$M_{\mathrm{d}-64}$+*coarse*	34	42	40	45	47	48	48	48
	Time per iteration							
# processors	125	216	343	512	729	1000	1331	1728
$M_{\mathrm{d}-64}$	0.76	0.76	0.77	0.78	0.79	0.82	0.84	0.87
$M_{\mathrm{d}-64}$+*coarse*	0.88	0.88	0.89	0.91	0.92	0.96	1.01	1.05

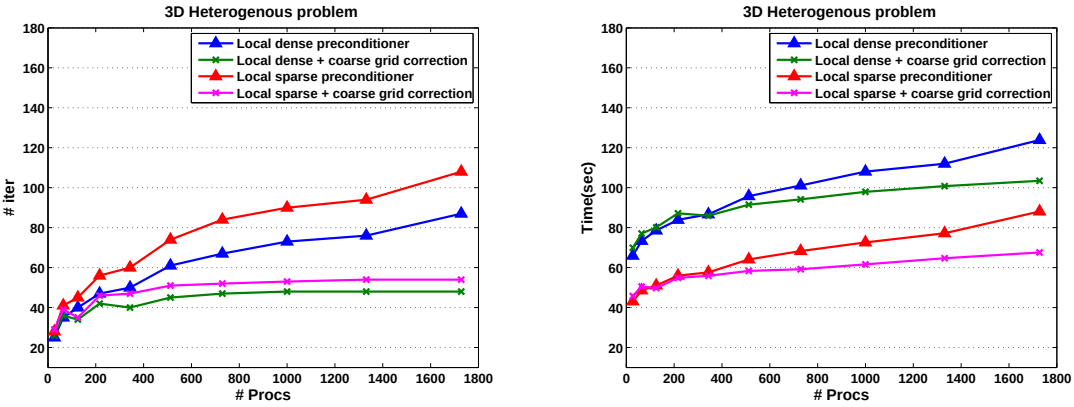
Table 5.17: Performance of a parallel two-level preconditioner on Problem 2 using a $35 \times 35 \times 35$ subdomain mesh for the dense preconditioner.

	Total solution time							
# processors	125	216	343	512	729	1000	1331	1728
$M_{\mathrm{sp}-64}$	51.0	56.0	57.6	64.1	68.3	72.6	77.2	88.1
$M_{\mathrm{sp}-64}$+*coarse*	49.8	54.9	55.9	58.4	59.2	61.6	64.7	67.6
	Time for coarse setup							
# processors	125	216	343	512	729	1000	1331	1728
$M_{\mathrm{sp}-64}$	–	–	–	–	–	–	–	–
$M_{\mathrm{sp}-64}$+*coarse*	2.04	2.05	2.20	2.38	2.76	3.68	4.12	4.89
	Time in the iterative loop							
# processors	125	216	343	512	729	1000	1331	1728
$M_{\mathrm{sp}-64}$	18.0	23.0	24.6	31.1	35.3	39.6	44.2	55.1
$M_{\mathrm{sp}-64}$+*coarse*	14.7	19.8	20.7	22.9	23.4	24.9	27.5	29.7
	# iterations							
# processors	125	216	343	512	729	1000	1331	1728
$M_{\mathrm{sp}-64}$	45	56	60	74	84	90	94	108
$M_{\mathrm{sp}-64}$+*coarse*	35	46	47	51	52	53	54	54
	Time per iteration							
# processors	125	216	343	512	729	1000	1331	1728
$M_{\mathrm{sp}-64}$	0.40	0.41	0.41	0.42	0.42	0.44	0.47	0.51
$M_{\mathrm{sp}-64}$+*coarse*	0.42	0.43	0.44	0.45	0.45	0.47	0.51	0.55

Table 5.18: Performance of a parallel two-level preconditioner on Problem 2 using a $35 \times 35 \times 35$ subdomain mesh for the sparse preconditioner.



(a) Poisson problem.



(b) Discontinuous Problem 2.

Figure 5.7: The number of PCG iterations (left) and the computing time (right) for a $35 \times 35 \times 35$ grid when varying the number of subdomains from 27 to 1728. comparison between the local and the two-level preconditioner for the dense and the sparse variants.

5.5 Concluding remarks

In this chapter, we have studied the numerical and implementation scalability of variants of the additive Schwarz preconditioner in non overlapping domain decomposition techniques for the solution of large 3D academic elliptic problems. The numerical experiments show that the sparse variant enables us to get reasonable numerical behaviour and permits the saving of a significant amount of memory. Although we have not yet any theoretical arguments to establish the backward stability of the mixed precision approach, this technique appears very promising in the context of multi-core heterogeneous massively parallel computers, where some devices (such a the graphic cards) only operate in 32-bit arithmetic. Some works deserve to be undertaken to validate the approach in this computing context as well as theoretical developments to assess their numerical validity.

Although these preconditioners are local, consequently not numerically scalable, they exhibit a fairly good parallel time scalability as the relative cost of the setup partially hides the moderate increase in the number of iterations. In order to compensate the lack of numerical scalability we investigated their numerical behaviour when the local components are used in conjunction with a simple coarse grid correction. This latter component enables to recover the numerical scalability while not penalizing much the time per iteration. In the current experiments, the coarse problems are solved redundantly on each processor. On large massively computers with tens of thousands processors, the size of the coarse problems might require to consider parallel solution. Different variants can be foreseen. The most promising would be to dedicate a subset of processors to the solution of this additive component while the others compute the local parts. Special attention would have to be paid to ensure a good work balance among the various processors that no longer work in a SPMD mode.

Chapter 6

Numerical investigations on convection-diffusion equations

6.1 Introduction

In the previous chapter, we study the numerical and parallel scalability of the algebraic additive Schwarz preconditioners for the solution of symmetric positive definite systems arising from the discretization of self-adjoint elliptic equations. This chapter is devoted to a similar study for $3D$ convection-diffusion problems [48] of the form given by Equation (6.1)

$$\left\{ \begin{array}{rcl} -\epsilon \mathrm{div}(K.\nabla u) + v.\nabla u &=& f \quad \mathrm{in} \quad \Omega, \\ u &=& 0 \quad \mathrm{on} \quad \partial\Omega. \end{array} \right. \tag{6.1}$$

Such problems appear in many mathematical modeling of wide range of scientific and technical phenomena such as heat and mass-transfer, flow and transport in porous media related to petroleum and ground water applications, etc. The matrices resulting from the discretization of these problems are unsymmetric, even if the original elliptic operator was self adjoint due to the convection component.

In this chapter, we consider academic problems associated with the discretization of Equation (6.1) in the unit cube for various diffusion and convection terms in order to study the robustness of the preconditioners.

6.2 Experimental environment

We investigate the parallel scalability of the proposed implementation of the preconditioners. The studies were carried out using the local parallel computing facility at CERFACS. The target computer is the revolutionary IBM eServer Blue Gene/L supercomputer. Blue Gene/L already represents a phenomenal leap in the supercomputer race, with a peak performance of 5.7 TFlops. From a practical point of view, Blue Gene/L is built starting with dual CPU (processor) chips placed in pairs on a compute card together with $2 \times 512 MBytes$ of RAM ($512MB$ for each dual core chip). Blue Gene/L consists of 1024 chips, where each chip has two modified PowerPC 440s running at $700\ MHz$ and each CPU can perform four floating-point operations per cycle, giving a theoretical peak performance of 2.8 GFlops/chip. The CPU used here has a much lower clock frequency than other players in the field such as AMD Opteron, IBM POWER, and Intel Pentium 4. Also, it has not been designed to run server OS's like LINUX or AIX. Thus, the applications that can be run on this supercomputer are of a very specific scientific and technical nature. These chips are connected by three networks:

- $3D$ torus: bandwidth of 2.1 $GBytes/s$, offered latency between 4 and 10 μs, dedicated to one to one MPI communications.
- Tree: offers a bandwidth of 700 $MBytes/s$ and a latency of 5 μs for I/O. The collective network connects all the compute nodes in the shape of a tree; any node can be the tree root (originating point). MPI implementation will use that network each time it happens to be more efficient than the torus network for collective communication.
- The barrier (global interrupt) network: as the number of tasks grows, a simple (software) barrier in MPI costs more and more. On a very large number of nodes, an efficient barrier becomes mandatory. The barrier (global interrupt) network is the third dedicated hardware network Blue Gene/L provides for efficient MPI communication with a latency of 1.3 μs.

All interactions between the Blue Gene/L computing nodes and the outside world are carried through the I/O nodes under the control of the service node. There are 16 nodes dedicated to I/O and two networks connecting the service node to the I/O nodes (a gigabit network and a service network). This platform is equipped by different software and scientific libraries such as:

- IBM compilers: XL Fortran and XL C/C++ for Linux Blue Gene/L versions.
- IBM Engineering and Scientific Subroutine Libraries (ESSL).
- MPI library (MPICH2) V0.971.
- Mathematical Acceleration Subsystem (MASS) libraries.
- GNU Tool-chain (glibc, gcc, binutils, gdb).
- Java Runtime JRE 1.4.1.
- IBM LoadLeveler.
- IBM General Parallel File System (GPFS).

We briefly recall the different problems investigated in the previous chapter to define the diffusion term in Equation (6.1). We then introduce two fields to define the convection terms considered in our numerical simulations. A scalar term is used in front of the diffusion term that enables us to vary the Péclet number so that the robustness with respect to this parameter can be investigated. These various choices of $3D$ model problems are though to be difficult enough and representative for a large class of applications.

We consider for the diffusion coefficient as in the previous chapter, the matrix K in (6.1) as diagonal with piecewise constant function entries defined in the unit cube as depicted in Figure 6.1. The diagonal entries $a(x,y,z)$, $b(x,y,z)$, $c(x,y,z)$ of K are bounded positive functions on Ω enabling us to define heterogeneous and/or anisotropic problems.

To vary the difficulties we consider both discontinuous and anisotropic PDE's where constant diffusion coefficients are defined along vertical beams according to Figure 6.1 pattern. For the sake of completeness we also consider the simple homogeneous diffusion where all the coefficient functions a, b and c are identically one. More precisely we define the following set of problems:

Problem 1: Poisson where $a(\cdot) = b(\cdot) = c(\cdot) = 1$.

Problem 2: heterogeneous diffusion problem based on Pattern 1;

$$a(\cdot) = b(\cdot) = c(\cdot) = \begin{cases} 1 & \text{in} \quad \Omega^1 \cup \Omega^3 \cup \Omega^5, \\ 10^3 & \text{in} \quad \Omega^2 \cup \Omega^4 \cup \Omega^6. \end{cases}$$

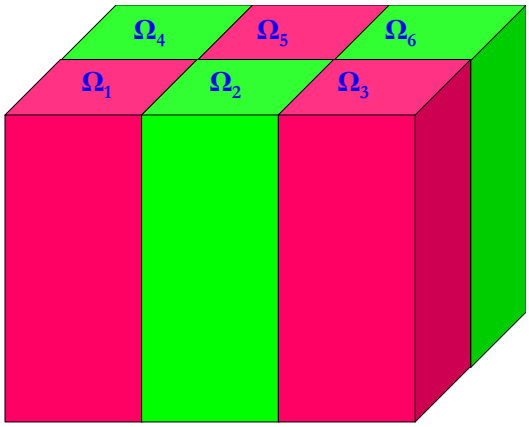


Figure 6.1: Pattern for heterogeneous diffusion: variable coefficient domains.

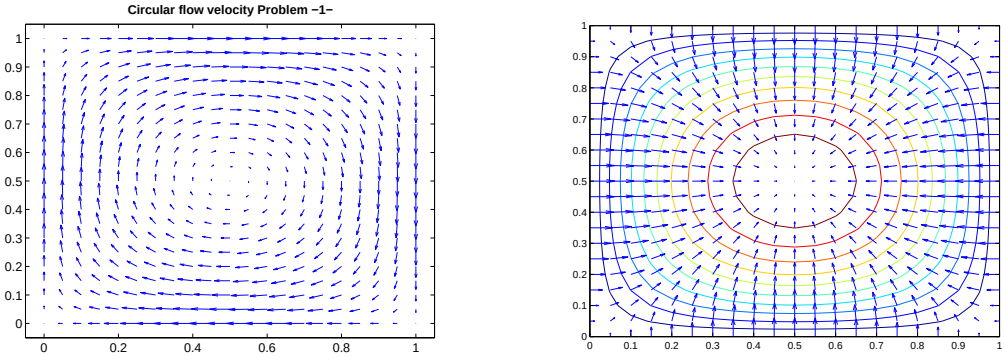
Problem 3: heterogeneous and anisotropic diffusion problem based on Pattern 1; $a(\cdot) = 1$ and

$$b(\cdot) = c(\cdot) = \begin{cases} 1 & \text{in} \quad \Omega^1 \cup \Omega^3 \cup \Omega^5, \\ 10^3 & \text{in} \quad \Omega^2 \cup \Omega^4 \cup \Omega^6. \end{cases}$$

For each of the diffusion problems described above we define a convection term for all the directions. We choose two types of convection problems:

- Convection 1: models a circular flow in the xy direction while a sinusoidal flow in the z direction. Figure 6.2 shows the streamlines of the convection field. This convection field is:

$$\begin{cases} v_x(\cdot) &= (x - x^2)(2y - 1), \\ v_y(\cdot) &= (y - y^2)(2x - 1), \\ v_z(\cdot) &= sin(\pi z). \end{cases}$$



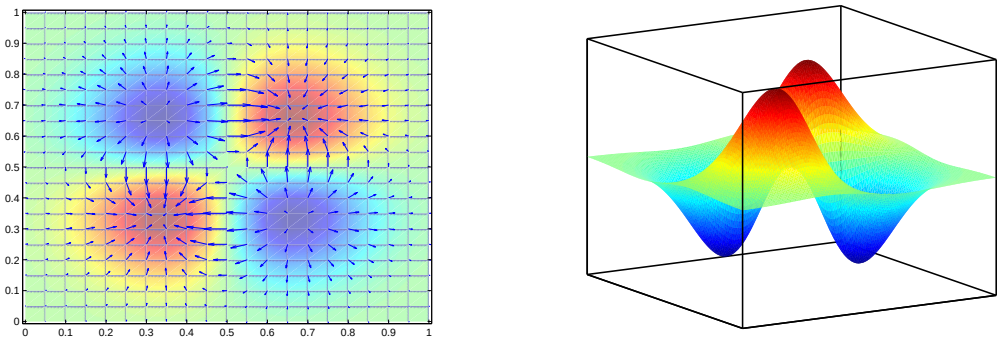
(a) xy plane. (b) yz plane.

Figure 6.2: circular convection flow.

- Convection 2: is an example of a four area sinusoidal flow depicted in Figure 6.3. It is specified

by the convection field:

$$\begin{cases} v_x(\cdot) &= 4sin(y) * e^{-x^2-y^2} * (cos(x) - 2xsin(x)), \\ v_y(\cdot) &= 4sin(x) * e^{-x^2-y^2} * (cos(y) - 2ysin(y)), \\ v_z(\cdot) &= 0. \end{cases}$$



(a) xy plane.

(b) surface of the velocity field.

Figure 6.3: four area sinusoidal convection flow.

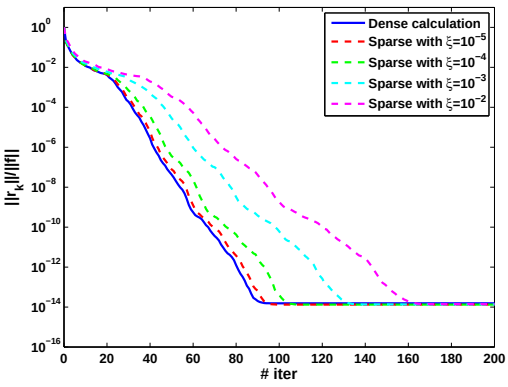
Each problem is discretized on the unit cube using standard second order finite difference discretization with a seven point stencil.

6.3 Numerical performance behaviour

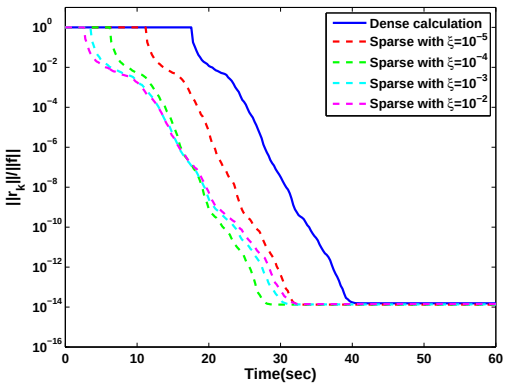
The discretization of the problems that we consider here give rise to linear systems that are unsymmetric, and we therefore have to replace the conjugate gradient solver by a suited Krylov subspace solver. Because of the theoretical results available for GMRES [84] in finite precision calculation we consider this solver and its closely related variants (FGMRES) for our numerical investigations. Furthermore, GMRES in practice has proved quite powerful for a large class of unsymmetric problems. In this section we present the convergence results and the computing time of the sparsified and mixed arithmetic preconditioners. We also compare them to the classical $M_{\mathrm{d}-64}$. Furthermore, we intend to evaluate the sensitivity of the preconditioners to the convection term. For that we analyze the effect of the Péclet number on the convergence rate. As in the previous chapter we consider the convergence history of the normwise backward error on the right-hand side that is defined by $\frac{\|r_k\|}{\|f\|}$ along the iterations. In that expression f denotes the right-hand side of the Schur complement system to be solved and r_k the true residual at the k^{th} iteration (i.e., $r_k = f - \mathcal{S}x_\Gamma^{(k)}$). We remind that only right preconditioner is considered so that the backward error is independent from the preconditioner which enables us to make fair comparison between the various variants.

6.3.1 Influence of the sparsification threshold

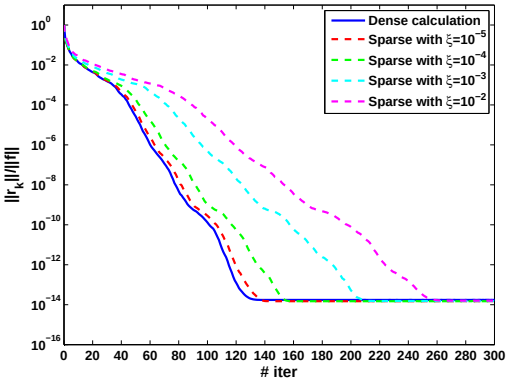
The sparse feature of the preconditioner was originally developed for SPD matrices, for which properties of the resulting preconditioner such as SPD can be proved [25]. However this strategy has been successfully applied in much more general situations as the unsymmetric case here. We presented here several test problems that can help us to determine the behaviour of the sparse preconditioner. For these tests, the size of the system is a decomposition of $300 \times 300 \times 300$



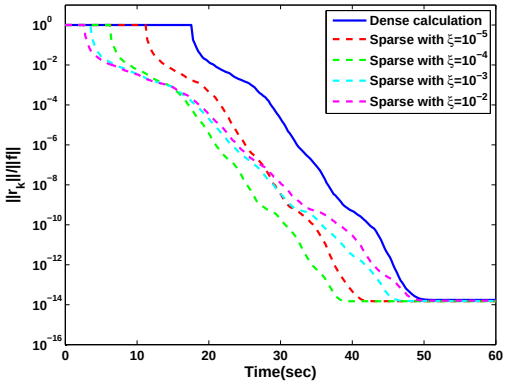
(a) Homogeneous diffusion Problem 1 with Convection 1 (history v.s. iterations).



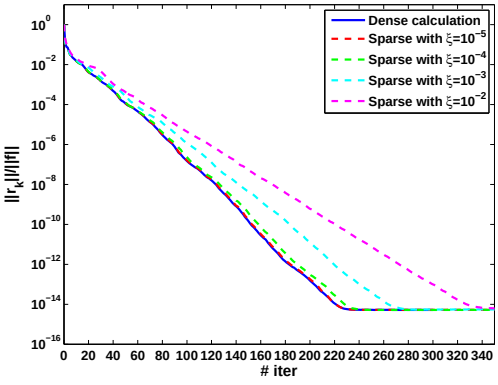
(b) Homogeneous diffusion Problem 1 with Convection 1 (history v.s. time).



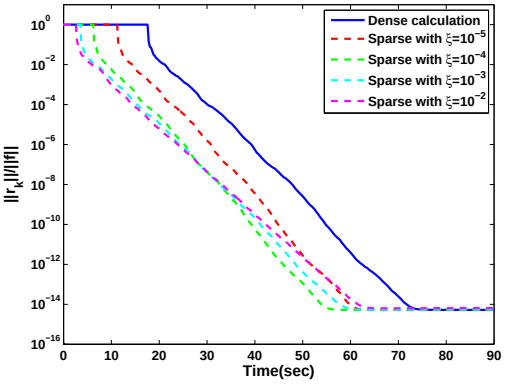
(c) Heterogeneous diffusion Problem 2 with Convection 1 (history v.s. iterations).



(d) Heterogeneous diffusion Problem 2 with Convection 1 (history v.s. time).



(e) Heterogeneous and anisotropic diffusion Problem 3 with Convection 1 (history v.s. iterations).



(f) Heterogeneous and anisotropic diffusion Problem 3 with Convection 1 (history v.s. time).

Figure 6.4: Convergence history for a $300\times300\times300$ mesh mapped onto 1728 processors for various sparsification dropping thresholds (Left: scaled residual versus iterations, Right: scaled residual versus time). The convection term is defined by Convection 1 and low Péclet number ($\varepsilon = 1$).

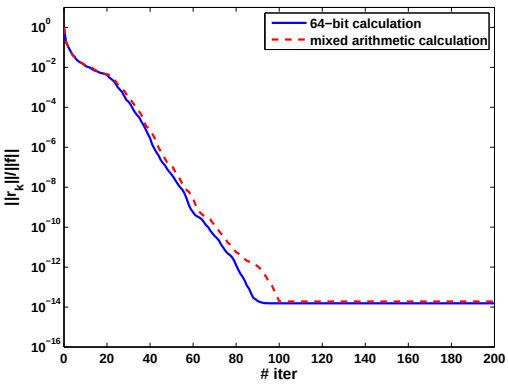
mesh mapped onto 1728 processors. That is, each subdomain has a size of about 15,000 dof. We will briefly compare and show the effect of the sparsification parameter for the different problems mentioned above. We display in Figure 6.4 the convergence history for different choices of the dropping parameter ξ defined in Equation (3.6) and compare them with the dense one. The left graphs of Figure 6.4 show the convergence history as a function of the iterations, whereas the right graphs gives the convergence as a function of the computing time. The shape of these graphs is typical: as ξ is increased the amount of kept entries is decreased, the setup time of the preconditioner (initial plateaus of the graphs) becomes faster but the convergence is deteriorated. For a large sparsification threshold (magenta line), there are not enough nonzero entries in the preconditioner to allow for a convergence behaviour similar to the dense variant. For a small sparsification threshold (red line), the numerical performance of the sparse preconditioner is closer to the dense one and the convergence behaviours are similar. Similarly to what we observed for pure diffusion problem, a nice trade-off between memory and elapsed time is obtained for a sparsification threshold $\xi = 10^{-4}$ (the green line). Even though the sparse variants require more iterations, with respect to time they converge faster as the preconditioner setup is more than twice cheaper and the time per iteration is also smaller. This trend was already observed for the symmetric case in Chapter 5.

6.3.2 Influence of the mixed arithmetic

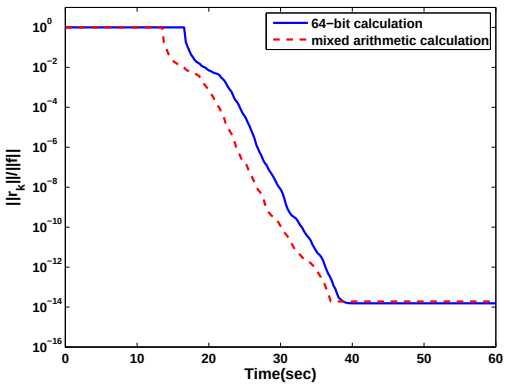
In this section we consider the mixed precision approach [50]. In this framework, only the preconditioning step is performed in 32-bit arithmetic; the rest of the calculation is carried out in 64-bit. In the right preconditioned GMRES context, the backward stability result indicates that it is hopeless to expect convergence at a backward error level smaller than the 32-bit accuracy. To overcome this limitation the preconditioner can be considered as variable along the iterations. At each step the 32-bit preconditioner can be viewed as variable perturbed 64-bit preconditioner. In this context, our choice is to use the flexible GMRES method instead of GMRES.

We focus in this section, in the numerical behaviour of the mixed approach and compare it with a fully 64-bit approach. For this purpose we consider the same example as the previous section, and also the same decomposition. The left graphs of Figure 6.5 show the numerical performance comparison between the mixed precision algorithm, and the 64-bit algorithm for a decomposition of $300 \times 300 \times 300$ mesh mapped onto 1728 processors. These results show that the mixed arithmetic implementation compares favorably with the 64-bit one. We observe for this algorithm, that the number of iterations slightly increases. We notice that the increase induced by the mixed arithmetic is smaller than the one encountered in the previous section when a sparsification is used. Attractive enough, the attainable accuracy of the mixed algorithm compares very closely to the 64-bit one. This feature is illustrated in the graphs of Figure 6.5 where is plotted the backward error associated with the two algorithms. We can clearly see that the mixed algorithm reaches the same accuracy level as the 64-bit algorithm. We also display in the right graphs, the backward error as a function of the elapsed time in sec. Because the computing platform used for these experiments does not allow higher processing rate in 32-bit compared to 64-bit the saving in computing time is less distinctive, than those presented in the Section 5.3.2 of the previous chapter. Larger computing time gains can be achieved by using another platform, such as the ones described in Section 5.2. To summarize, we can omit the effect of the computing gain because it is related to the machine architecture. There are some limitations to the success of this approach, such as when the conditioning of the problem exceeds the reciprocal of the accuracy of the single precision computations.

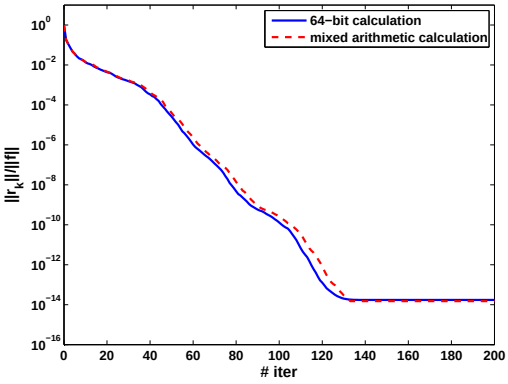
Furthermore, the gain in memory space due to the 32-bit storage of the preconditioner is partially consummated by the extra storage of the Z basis required by the flexible variant of GMRES.



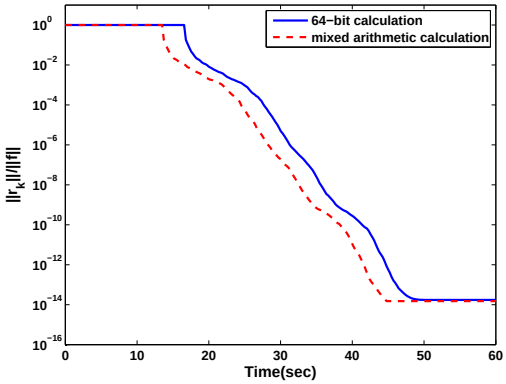
(a) Homogeneous diffusion Problem 1 with Convection 1 (history v.s. iterations).



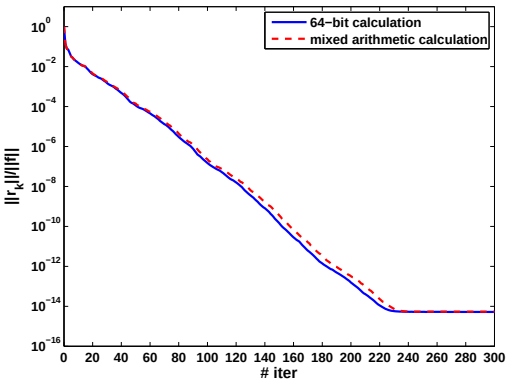
(b) Homogeneous diffusion Problem 1 with Convection 1 (history v.s. time).



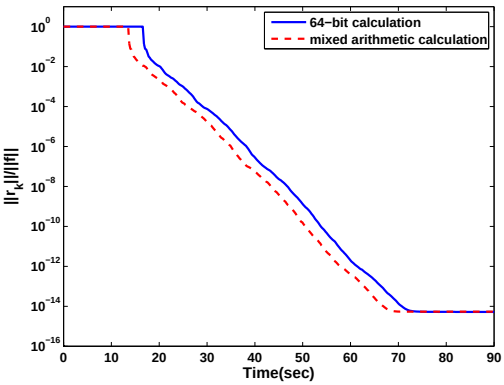
(c) Heterogeneous diffusion Problem 2 with Convection 1 (history v.s. iterations).



(d) Heterogeneous diffusion Problem 2 with Convection 1 (history v.s. time).



(e) Heterogeneous and anisotropic diffusion Problem 3 with Convection 1 (history v.s. iterations).

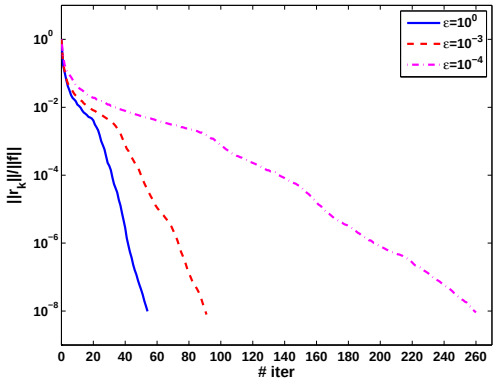


(f) Heterogeneous and anisotropic diffusion Problem 3 with Convection 1 (history v.s. time).

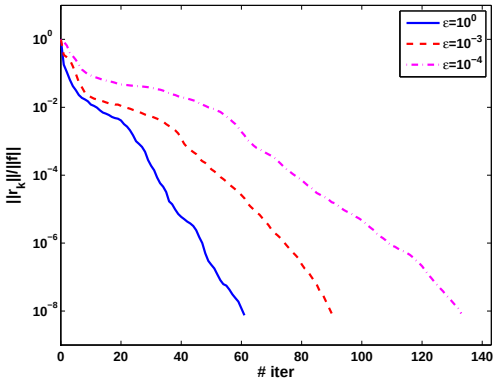
Figure 6.5: Convergence history for a $300 \times 300 \times 300$ mesh mapped onto 1728 processors for various dropping thresholds (Left: scaled residual versus iterations, Right: scaled residual versus time). The convection term is defined by Convection 1 and low Péclet number ($\varepsilon = 1$).

6.3.3 Effect of the Péclet number

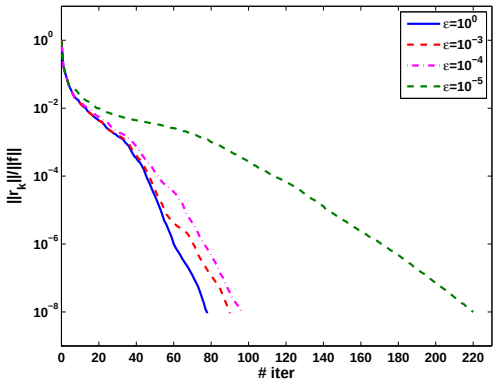
A crucial characteristic of a preconditioner is the way its response to disturbances changes when the system parameters change. For that, we intend to evaluate the sensitivity of the preconditioners to discontinuity with or without anisotropy in the diffusion coefficients, and to convection dominated problem. The convection dominated case which is most difficult to solve is particularly interesting, and can occur in many practical problems. Thus, the diffusion coefficient of Equation (6.1) is supposed to be very small $0 < \epsilon \ll 1$ compared to the norm of the velocity field v which governs the convection. As a consequence, the solution u of Equation (6.1) frequently contains many scales composed of a complex collection of exponential (or regular or parabolic) boundary layers. We should mention that for very small value of ϵ ($\epsilon = 10^{-5}$ or $\epsilon = 10^{-6}$), standard numerical methods such as the finite element method (FEM) or the difference method usually fail since they introduce nonphysical oscillations. One possible remedy involves additional stabilization. The most successful approaches are the streamline upwind Petrov Galerkin method (SUPG), also known as streamline diffusion finite element method (SDFEM), the Galerkin least squares approximation (GLS), and the Douglas-Wang method. We are not concerned in this thesis by the stabilization techniques, we can refer the reader to a extensive literature work of [45, 75, 83, 91, 97].



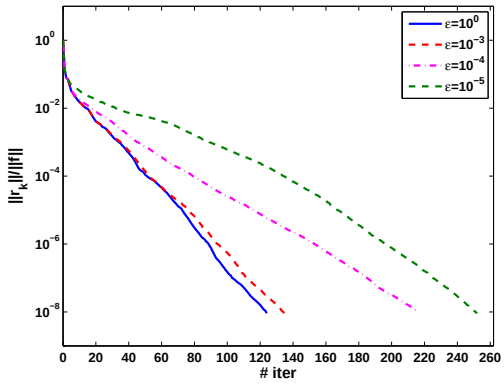
(a) Homogeneous diffusion Problem 1 with Convection 1.



(b) Homogeneous diffusion Problem 1 with Convection 2.



(c) Heterogeneous diffusion Problem 2 with Convection 1.



(d) Heterogeneous and anisotropic diffusion Problem 3 with Convection 1.

Figure 6.6: Convergence history for a $300 \times 300 \times 300$ mesh mapped onto 1728 processors for various convection trend.

In this context, we present here experiments for different value of ϵ, from the easiest example

with low Péclet number ($\epsilon = 1$) to the hardest dominated convection ($\epsilon = 10^{-5}$). These examples allow us to highlight the numerical effect caused by large convection. For these experiments, the mesh size of $300 \times 300 \times 300$ is mapped onto 1728 processors. We present the convergence history plot as a function of the number of iterations, to reduce normwise backward error on the right-hand side below 10^{-8}. Notice that the discontinuity and the anisotropy in the coefficients are considered. We will briefly present here graphs for the dense 64-bit preconditioner and show the effect of the convection parameter for the different problems mentioned above. More detailed experiments results for all variants of the preconditioner and for the different ϵ will be presented in the next two sections. In Figure 6.6, we present results for Problem 1 applying either Convection 1 (see Figure 6.6(a)) or Convection 2 (see Figure 6.6(b)). We then display in Figure 6.6(c) the heterogeneous diffusion Problem 2 with the Convection 1, and the heterogeneous anisotropic Problem 3 applied with Convection 1 is displayed in Figure 6.6(d).

The following examples presented here, underline and confirm the theoretical predictions, that is, increasing the convection term make harder the problem to solve. These results show that the number of iterations required by the GMRES grows with large convection term. This grow in the number of iterations remains reasonable, it depends slightly of the strength of the convection . More precisely, the mesh Péclet number should not exceed a certain value of the finest grid. This upper bound for the Péclet number coincides with the applicability of the central differencing scheme in the discretization. It was found that the case $\epsilon = 10^{-4}$ represents the rigorous test, while larger choice of ϵ exhibits faster convergence. For stronger convection $\epsilon = 10^{-5}$, the convergence does not hold. This case confirms the sensitivity to the nonphysical oscillations and dissipation schema, we omit the study of this case in this work and we propose to stabilize the discretization method and to control the dissipation in stretched regions of the mesh in a future work.

To summarize, the additive Schwarz preconditioner was found to be robust with respect to convection-diffusion equation, the results presented are satisfactory. The experiments with the different alternatives of the dense 64-bit additive Schwarz preconditioner have showed similar behaviour.

6.4 Parallel numerical scalability

The studies in this section present a more detailed look at performance from the point of view of the numerical and parallel scalability and their dependency on the different convection trends. We perform scaled experiments where the global problem size is varied linearly with the number of processors. Such experiments illustrate the ability of parallel computation to perform large simulations (fully exploiting the local memory of the distributed platform) in ideally a constant elapsed time. In the numerical experiments of the following subsections, the iterative method used to solve these problems is the right preconditioned GMRES for all variants and flexible GMRES for the mixed algorithm. We choose the ICGS (Iterative Classical Gram-Schmidt orthogonalization) strategy which is suitable for parallel implementation. The iterations began with a zero initial guess and were stopped when the normwise backward error becomes smaller than 10^{-8} or when 1000 steps are taken.

6.4.1 Numerical scalability on massively parallel platforms

We intend to present, evaluate and analyze the effect on the convergence rate of the different preconditioners considered on various model problems and various convection trends. Various results are presented in Tables 6.1 and 6.2. We divide the discussion into two steps:

- we illustrate the numerical scalability of the Krylov solver when the number of subdomains increases,
- we present performance based on the increase of the convection term.

Experiments when increasing the size of the subdomain have been performed, we observe that the subdomain size has only a slight effect on the convergence rate. For the sake of readability, we omit to present these results and only report on experiments with $25 \times 25 \times 25$ subdomain size. The preconditioners tested are the 64-bit dense additive Schwarz preconditioner $M_{\mathrm{d}-64}$, the mixed variant $M_{\mathrm{d}-\mathrm{mix}}$ and the sparse alternative one $M_{\mathrm{sp}-64}$. We first comment on the numerical scalability of the preconditioners when the number of subdomains is varied while the Péclet number is constant. This behaviour can be observed in the results displayed in Tables 6.1 and 6.2 by reading these tables by row. By looking at the number of iterations when the number of subdomains increases from 27 to 1728, that is, when increasing the overall problem size from 0.4 million dof up to 27 million dof. It can be seen that the increase in the number of iterations is moderate. When we multiply the number of subdomains by 64, the number of iterations increases between 3 to 4 times. For the characteristics of the problems and the associated difficulties, we can say that, the preconditioner which exploits the local information available on each subdomain, performs quite well. For example let us look at the Table 6.1, for the heterogeneous diffusion Problem 2 combined with the Convection 1. For $\epsilon = 10^{-3}$, when we increase the number of subdomains from 27 to 1728, we see that, the number of iterations of $M_{\mathrm{d}-64}$ increases from 26 to 90; that is, a 300% increase of iterations for a problem size that is multiplied by 64.

Moreover, we study the effect of changing the dropping threshold for the sparse variant of the preconditioner. As explained in Section 6.3.1, as this threshold increases, the sparsity of the preconditioner increases, and the preconditioner behaves poorly. For example in Table 6.1, when the diffusion term associated with Problem 2 is considered. We observe that gap between $M_{\mathrm{d}-64}$ and $M_{\mathrm{sp}-64}$ with $\xi = 10^{-3}$ is significant; more than 60 iterations (66%) on 1728 subdomains.

Furthermore, we see that, when we increase the number of subdomains, the sparser the preconditioner, the larger the number of iterations is. The gap is larger when the Péclet number is increased. Similarly to the pure diffusion case presented in Chapter 5, it appears that the choice $\xi = 10^{-4}$ provides us with the best trade-off between memory and solution time saving. Finally, we see that the mixed preconditioner $M_{\mathrm{d}-\mathrm{mix}}$ performs very similarly to the 64-bit one.

Regarding the behaviour of the preconditioners for convection dominated problems, although those problems are more difficult to solve the preconditioners are still effective. We recall that the preconditioners do not exploit any specific information about the problem (e.g., direction of flow). From a numerical point of view, if we read the tables column-wise, we can observe the effect of the increase of the Péclet number on the difficulties for the iterative scheme to solve the resulting linear systems. The good news, is that, even with this increase, the preconditioners perform reasonably well. This robustness is illustrated by the fact that the solution is tractable even for large Péclet numbers.

6.4.2 Parallel performance scalability on massively parallel platforms

In this subsection, we attempt to analyze the features of the preconditioners from a computational point of view. In that respect, we look at the main three steps that compose the solver. As described in the Section 5.4.3 of Chapter 5, the main parts of the method are:

- the initialization phase which is the same for all the variants of the preconditioners (mainly factorization of the local Dirichlet problems and calculation of the local Schur complements);
- the preconditioner setup phase which differs from one variant to another;
- the iterative loop which is related to convergence rate.

In this chapter the parallel computer is different from the one considered in Chapter 5; consequently the elapsed times of the initialization phase are different even though the size of the local subdomains are the same. In Tables 6.3 are depicted the elapsed time to factorize the local Dirichlet problem of the matrix associated with each subdomain, and to construct the local Schur

		# subdomains ≡ # processors									
subdomain grid size		27	64	125	216	343	512	729	1000	1331	1728
Homogeneous diffusion term											
$\epsilon = 1$	$M_{\mathrm{d-64}}$	18	25	27	31	35	40	44	47	50	54
	$M_{\mathrm{d-mix}}$	19	26	29	33	37	42	46	50	53	57
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-4})$	19	26	30	35	40	44	49	54	58	62
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-3})$	22	30	37	43	48	55	61	67	72	78
$\epsilon = 10^{-3}$	$M_{\mathrm{d-64}}$	23	35	39	49	54	60	67	74	82	91
	$M_{\mathrm{d-mix}}$	24	38	41	51	59	66	76	85	95	106
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-4})$	24	37	41	55	61	74	84	96	106	118
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-3})$	31	49	56	74	84	100	114	130	143	159
$\epsilon = 10^{-4}$	$M_{\mathrm{d-64}}$	74	100	111	139	154	182	195	219	238	260
	$M_{\mathrm{d-mix}}$	84	120	137	173	194	225	247	273	297	322
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-4})$	74	100	113	140	157	183	198	223	243	264
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-3})$	76	101	117	147	162	195	213	243	265	289
Diffusion term defined by Problem 2											
$\epsilon = 1$	$M_{\mathrm{d-64}}$	23	32	36	43	45	55	61	67	69	78
	$M_{\mathrm{d-mix}}$	24	34	38	45	47	58	64	70	72	82
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-4})$	25	34	39	49	52	64	71	77	79	92
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-3})$	33	44	50	66	68	86	97	104	104	125
$\epsilon = 10^{-3}$	$M_{\mathrm{d-64}}$	26	36	41	47	53	63	70	77	83	90
	$M_{\mathrm{d-mix}}$	27	37	43	49	56	65	72	80	86	93
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-4})$	29	39	46	56	63	74	83	91	98	109
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-3})$	38	50	58	77	82	103	116	126	130	153
$\epsilon = 10^{-4}$	$M_{\mathrm{d-64}}$	32	41	49	52	63	69	76	84	94	97
	$M_{\mathrm{d-mix}}$	33	43	50	54	65	72	78	86	96	100
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-4})$	36	45	54	62	72	82	92	100	111	118
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-3})$	43	58	68	83	92	111	124	137	142	162
Diffusion term defined by Problem 3											
$\epsilon = 1$	$M_{\mathrm{d-64}}$	33	46	54	71	71	81	91	97	101	124
	$M_{\mathrm{d-mix}}$	33	47	55	73	74	84	94	100	105	128
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-4})$	33	47	55	73	74	84	93	102	105	127
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-3})$	38	54	61	83	82	97	107	116	116	142
$\epsilon = 10^{-3}$	$M_{\mathrm{d-64}}$	38	48	58	75	79	91	107	108	114	135
	$M_{\mathrm{d-mix}}$	39	50	60	78	82	94	110	112	117	139
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-4})$	39	50	61	79	83	97	112	116	121	141
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-3})$	42	58	70	90	96	114	127	138	141	164
$\epsilon = 10^{-4}$	$M_{\mathrm{d-64}}$	46	61	76	93	110	129	152	168	190	216
	$M_{\mathrm{d-mix}}$	48	62	78	96	112	132	156	173	195	222
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-4})$	47	65	82	102	117	140	167	185	206	237
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-3})$	57	82	99	129	145	176	214	235	256	303

Table 6.1: Number of preconditioned GMRES iterations for various diffusion terms combined with Convection 1 when the number of subdomains and the Péclet number are varied.

		# subdomains ≡ # processors									
subdomain grid size		27	64	125	216	343	512	729	1000	1331	1728
Homogeneous diffusion term											
$\epsilon = 1$	$M_{\mathrm{d-64}}$	21	27	29	34	39	44	48	52	57	61
	$M_{\mathrm{d-mix}}$	22	28	32	36	41	46	51	55	60	64
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-4})$	22	28	33	40	45	49	55	61	65	70
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-3})$	26	34	41	49	56	63	70	77	81	88
$\epsilon = 10^{-3}$	$M_{\mathrm{d-64}}$	32	28	29	58	38	46	76	54	62	90
	$M_{\mathrm{d-mix}}$	34	29	29	61	40	48	79	56	64	93
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-4})$	33	28	30	61	43	51	81	63	72	98
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-3})$	35	34	36	73	57	68	106	83	96	127
$\epsilon = 10^{-4}$	$M_{\mathrm{d-64}}$	263	435	403	96	69	71	107	69	73	133
	$M_{\mathrm{d-mix}}$	375	0	0	101	70	72	110	70	74	136
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-4})$	271	451	381	95	71	73	110	73	81	138
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-3})$	283	494	363	95	73	79	132	88	99	167
Diffusion term defined by Problem 2											
$\epsilon = 1$	$M_{\mathrm{d-64}}$	23	32	36	43	46	56	62	68	69	79
	$M_{\mathrm{d-mix}}$	24	34	38	45	49	58	64	70	73	82
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-4})$	26	34	39	49	53	64	71	77	80	92
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-3})$	35	46	51	67	70	87	98	105	106	127
$\epsilon = 10^{-3}$	$M_{\mathrm{d-64}}$	27	37	43	49	56	66	72	79	87	93
	$M_{\mathrm{d-mix}}$	27	39	45	51	59	69	75	83	90	96
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-4})$	29	40	48	58	66	77	85	94	101	111
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-3})$	38	54	60	79	85	106	118	127	131	154
$\epsilon = 10^{-4}$	$M_{\mathrm{d-64}}$	28	62	56	60	71	80	86	98	112	116
	$M_{\mathrm{d-mix}}$	28	65	57	62	73	82	88	100	115	118
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-4})$	29	62	60	67	81	92	102	114	130	139
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-3})$	38	70	74	87	100	121	134	148	167	183
Diffusion term defined by Problem 3											
$\epsilon = 1$	$M_{\mathrm{d-64}}$	36	45	54	71	71	81	95	97	101	124
	$M_{\mathrm{d-mix}}$	38	47	55	74	74	85	97	100	105	128
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-4})$	37	47	55	73	74	84	96	102	105	127
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-3})$	42	54	61	84	82	97	107	116	116	143
$\epsilon = 10^{-3}$	$M_{\mathrm{d-64}}$	44	56	75	100	114	132	169	181	196	235
	$M_{\mathrm{d-mix}}$	45	58	78	104	118	136	175	186	203	242
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-4})$	47	59	79	108	123	144	180	197	211	251
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-3})$	59	75	95	135	152	183	218	248	257	308
$\epsilon = 10^{-4}$	$M_{\mathrm{d-64}}$	203	222	281	147	107	128	158	179	213	230
	$M_{\mathrm{d-mix}}$	221	237	308	170	110	131	162	182	218	236
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-4})$	206	231	288	168	120	147	175	204	234	255
	$M_{\mathrm{sp-64}}$ $(\xi = 10^{-3})$	221	277	331	205	164	200	227	275	322	343

Table 6.2: Number of preconditioned GMRES iterations for various diffusion terms combined with Convection 2 when the number of subdomains and the Péclet number are varied.

Subdomain grid size	$20 \times 20 \times 20$	$25 \times 25 \times 25$	$30 \times 30 \times 30$
Time	5.3	20.27	37.7

Table 6.3: Initialization time (sec).

complement using the MUMPS package, and that for different problem sizes. The results illustrate again the nonlinear cost of the direct solver with respect to the problem size.

Regarding the computational time to build the preconditioner which is the second step of the method. This cost includes the time to assemble the local Schur complement, and to factorize the assembled dense local Schur for $M_{\mathrm{d}-64}$ and $M_{\mathrm{d}-\mathrm{mix}}$, or to sparsify and then to factorize the resulting sparse assembled local Schur for $M_{\mathrm{sp}-64}$. The elapsed time of this step is independent from the number of subdomains, and depends only on two factors. They are, first, the size of the local problem (the leading size of the interface of this subdomain), and second the number of neighbour subdomains (which is equal to 26 for an internal domain). We report in Table 6.4 the

Subdomain grid size	$M_{\mathrm{d}-64}$	$M_{\mathrm{d}-\mathrm{mix}}$	$M_{\mathrm{sp}-64}$ with $\xi = 10^{-4}$	$M_{\mathrm{sp}-64}$ with $\xi = 10^{-3}$
$20 \times 20 \times 20$	4.1	3.4	2.4	1.2
$25 \times 25 \times 25$	17.5	14.1	6.4	3.5
$30 \times 30 \times 30$	40.1	33.0	10.5	4.2

Table 6.4: Preconditioner setup time (sec).

setup time for the different variants of the preconditioner and for different sizes of the subdomains. Referring to Table 6.4, one can observe that the performance of the sparse preconditioner compared to the dense one, is more than 3 time faster. In the case of the mixed arithmetic algorithm, and as it could have been expected on that platform, no significant computational speedup can be observed because the 64-bit calculation is as fast as the 32-bit one. The small improvement can be due to higher cache hit in 32-bit since the algorithm still only uses half of the memory space.

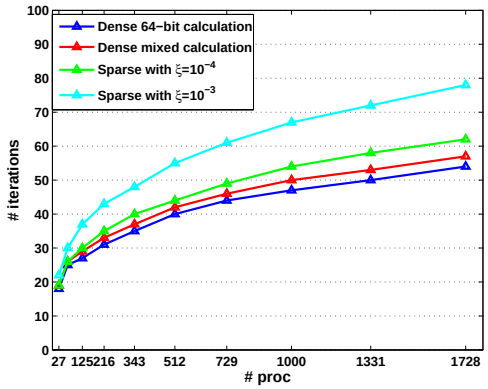
In order to study the performance of the iterative loop, we report in Table 6.5 the average time per iteration. In contrast with the CG situation, the time per iteration in GMRES/FGMRES does depend on the iteration number as a crucial step is the orthogonalization of the Krylov basis. We present in Table 6.5, the average time per iteration of the iterative loop for a fixed number of iterations equal to 300, for a fixed problem size, when increasing the number of subdomains. Thus, we give the average time of one iteration for subdomains of size $25 \times 25 \times 25$ (15,625 dof). It can

# processors	125	216	343	512	729	1000	1331	1728
$M_{\mathrm{d}-64}$	0.200	0.205	0.211	0.216	0.235	0.235	0.236	0.245
$M_{\mathrm{d}-\mathrm{mix}}$	0.189	0.191	0.198	0.217	0.216	0.216	0.220	0.225
$M_{\mathrm{sp}-64}$ ($\xi = 10^{-4}$)	0.176	0.177	0.177	0.191	0.195	0.197	0.198	0.209
$M_{\mathrm{sp}-64}$ ($\xi = 10^{-3}$)	0.169	0.170	0.174	0.187	0.194	0.195	0.196	0.205

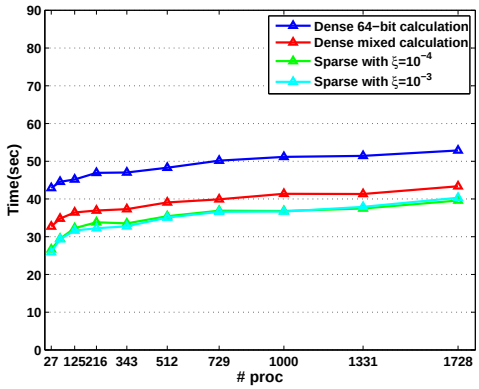
Table 6.5: Parallel average elapsed time for one iteration of the GMRES/FGMRES (sec).

be seen that the average elapsed time per iteration is nearly constant and does not depend much on the number of processors. For example increasing the number of processors from 125 to 1728, the time goes from 0.2 seconds up to 0.24 seconds for $M_{\mathrm{d}-64}$, which gives rise to a efficient parallel implementation of the iterative solver. This very nice scalability is mainly due to the network available on the Blue Gene computer dedicated to the reductions. The second observation is that the sparse alternative $M_{\mathrm{sp}-64}$ leads to a smaller average time per iteration in comparison with the dense one. This is due to the fact that the the time to apply the preconditioner is more than twice faster.

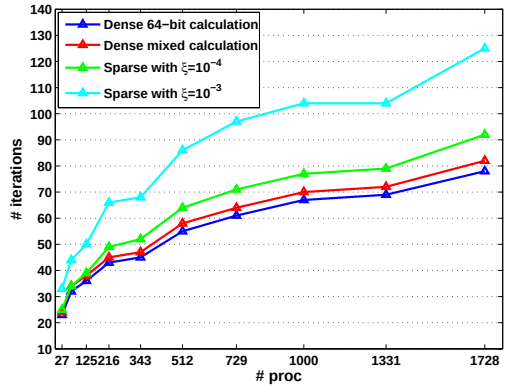
In the sequel, we consider the overall computing time with the aim of analyzing the parallel scalability of the complete algorithms. We display in the left graphs of Figures 6.7-6.9 the number of iterations required to solve the linear systems. On the right graphs we display the corresponding elapsed time of the overall solution. For each of these tests, we recall that the subdomains are $25 \times 25 \times 25$ grid mesh with 15,625 dof. As expected, even if the number of iterations to converge



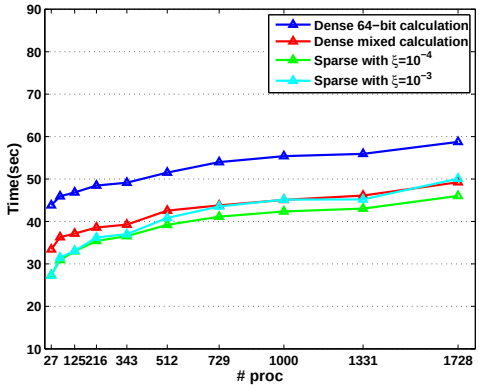
(a) Homogeneous diffusion Problem 1 with Convection 1.



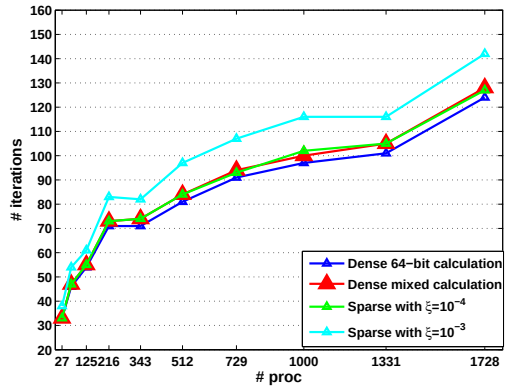
(b) Homogeneous diffusion Problem 1 with Convection 1.



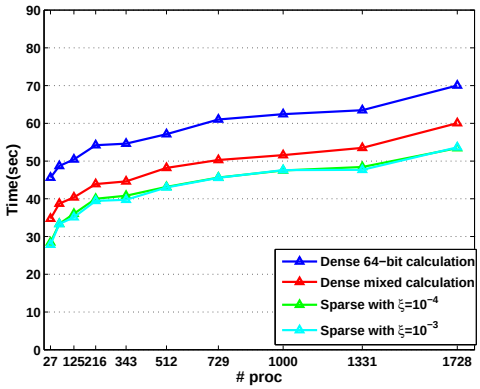
(c) Heterogeneous diffusion Problem 2 with Convection 1.



(d) Heterogeneous diffusion Problem 2 with Convection 1.

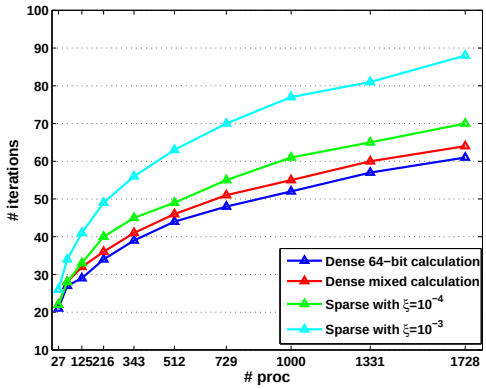


(e) Heterogeneous and anisotropic diffusion Problem 3 with Convection 1.

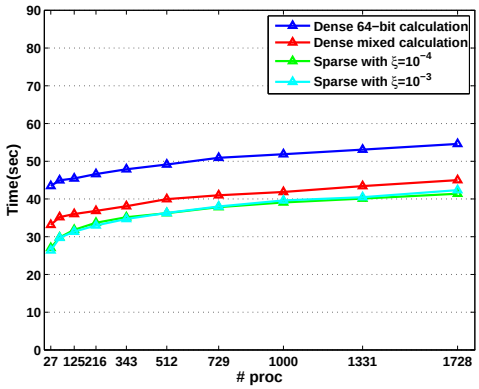


(f) Heterogeneous and anisotropic diffusion Problem 3 with Convection 1.

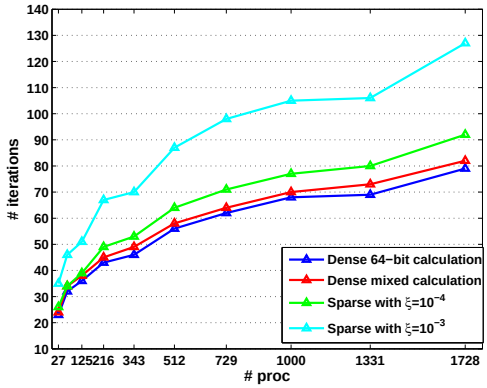
Figure 6.7: Parallel scalability of the three test problems, when varying the number of processors from 27 up to 1728. The convection term is defined by Convection 1 and low Péclet number ($\varepsilon = 1$). (Left: number of iterations, Right: overall computing time for the solution).



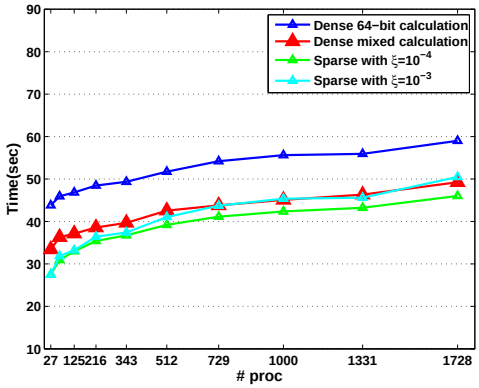
(a) Homogeneous diffusion Problem 1 with Convection 2.



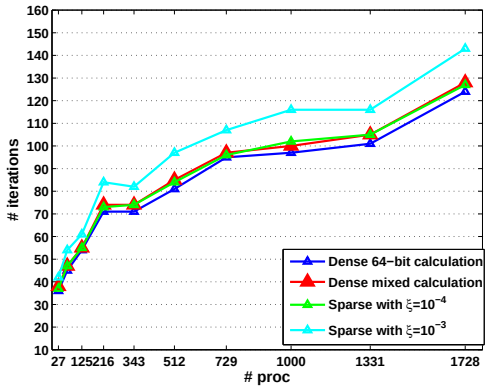
(b) Homogeneous diffusion Problem 1 with Convection 2.



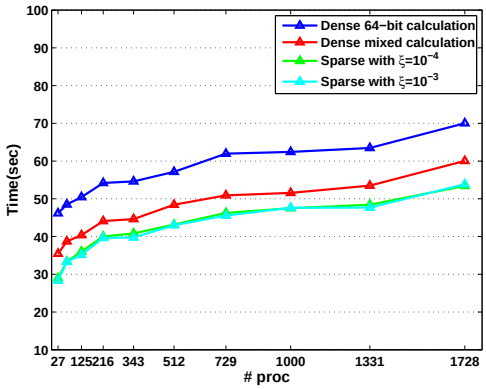
(c) Heterogeneous diffusion Problem 2 with Convection 2.



(d) Heterogeneous diffusion Problem 2 with Convection 2.

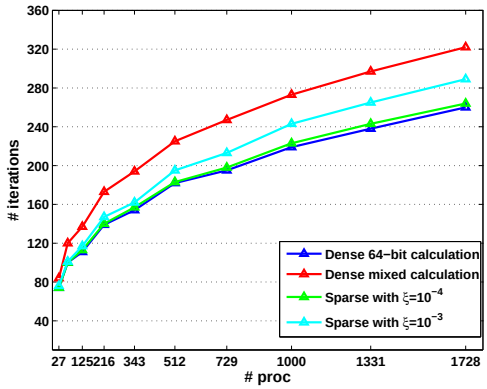


(e) Heterogeneous and anisotropic diffusion Problem 3 with Convection 2.

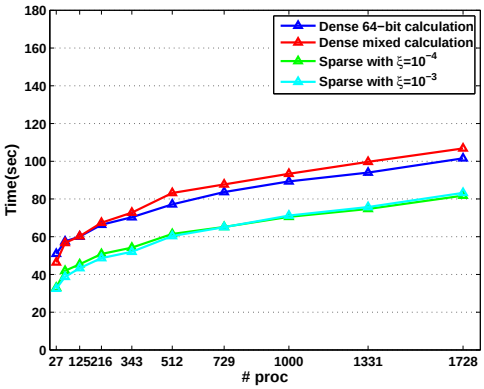


(f) Heterogeneous and anisotropic diffusion Problem 3 with Convection 2.

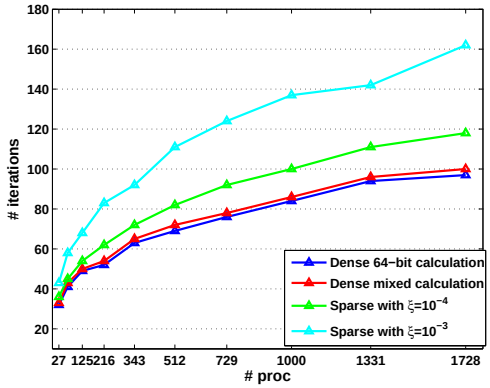
Figure 6.8: Parallel scalability of the three test problems, when varying the number of processors from 27 up to 1728. The convection term is defined by Convection 2 and low Péclet number ($\varepsilon = 1$). (Left: number of iterations, Right: overall computing time for the solution).



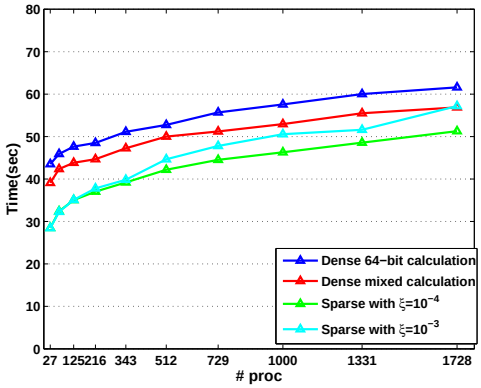
(a) Homogeneous diffusion Problem 1 with Convection 1 and $\varepsilon = 10^{-4}$.



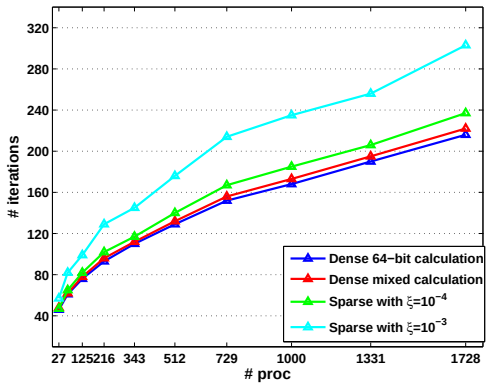
(b) Homogeneous diffusion Problem 1 with Convection 1 and $\varepsilon = 10^{-4}$.



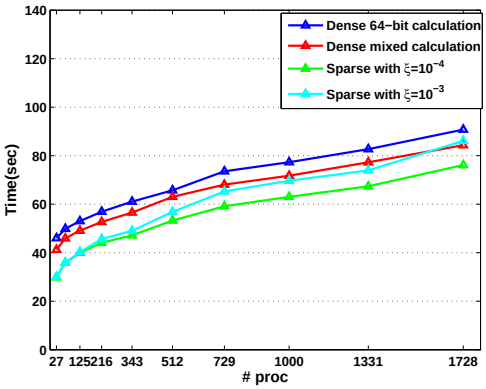
(c) Heterogeneous diffusion Problem 2 with Convection 1 and $\varepsilon = 10^{-4}$.



(d) Heterogeneous diffusion Problem 2 with Convection 1 and $\varepsilon = 10^{-4}$.



(e) Heterogeneous and anisotropic diffusion Problem 3 with Convection 1 and $\varepsilon = 10^{-4}$.



(f) Heterogeneous and anisotropic diffusion Problem 3 with Convection 1 and $\varepsilon = 10^{-4}$.

Figure 6.9: Parallel scalability of the three test problems, when varying the number of processors from 27 up to 1728. The convection term is defined by Convection 1 and high Péclet number ($\varepsilon = 10^{-4}$). (Left: number of iterations, Right: overall computing time for the solution).

increases as the number of subdomains increases, the growth in the solution time is rather moderate for all variants. The growth in the number of iterations as the number of subdomains increases is rather pronounced, whereas it is rather moderate for the global solution time as the initialization step represents a significant part of the overall calculation.

Similarly, to what we observed in the previous chapter, the sparse preconditioner $M_{\mathrm{sp}-64}$ performs at its best when the dropping parameter $\xi = 10^{-4}$. With this variant, the reduction in the setup of the preconditioner is significant; this gain is large enough to compensate for the few additional iterations required to converge. Dropping more entries by using $\xi = 10^{-3}$ often lead to an increase of the elapsed time as the number of iterations grows significantly.

6.5 Concluding remarks

In this chapter, we have studied the numerical and parallel scalability of various variants of our preconditioner for the solution of unsymmetric problems arising from the discretization of academic 3D convection diffusion problems. Similarly to what was observed for symmetric positive definite problems in the previous chapter, the variants based on sparse approximations of the assembled local Schur complement exhibit attractive features both in term of computation (memory and CPU saving) but also numerically in term of convergence rate compared to their dense counterparts. On all our experiments they are the most efficient and reduce the solution time and the memory space. For those problems, the behaviour with respect to the dropping threshold is quite smooth.

For unsymmetric problems, the use of mixed arithmetic preconditioners requires to use the flexible variant of GMRES if a high accuracy is expected. In that context, the theoretical backward stability result of GMRES indicates that the backward error cannot be lower than the 32-bit machine precision; this limitation does not seem to exist for FGMRES even though no theoretical result exists yet. Such a theoretical study would deserve to be undertaken possibly following the pioneer work [6]. Because of the parallel platform used for the experiments has similar computing speed in 32 and 64-bit, the potential benefit in time has not been illustrated. Nevertheless it would have been observed on computers as the System-X or on the Cray considered in the previous chapter.

For those problems with dominated convection it is known that the numerical scalability cannot be recovered thanks to the use of a coarse grid mechanism. One alternative to avoid losing computing power (due to the increase of the number of iterations) when the number of processors is increased would be to dedicate more than one processor per subdomain. This possibility was not considered on those problems but will be investigated in the next two chapters related to real life applications.

III

Part III

Study of parallel scalability on large real application problems

Part III: résumé

La validation de notre approche sur des cas réels est enfin réalisée sur des problèmes de mécanique des structures en maillages non-structurés (en collaboration avec la société SAMTECH-Chapitre 7) et en imagerie sismique (en collaboration avec le consortium SEISCOPE - Chapitre 8). Dans ce dernier cas, on s'intéresse à la résolution des équations de Helmholtz en régime fréquentiel. Plusieurs simulations sur des cas réels 2D et 3D ont été réalisés. L'objectif est d'évaluer la robustesse et la performance de notre méthode hybride pour la solution des ces problèmes "grands challenge" qui sont classiquement résolus par des méthodes directes.

Pour ces applications, la décomposition (partitioning en Anglais) jour un rÃťle central. C'est un sujet important pour les simulations concernant les applications réelles et industrielles. Nous illustrons l'influence des stratégies de décompositions sur la performance de l'algorithme de résolution dans le Chapitre 7.

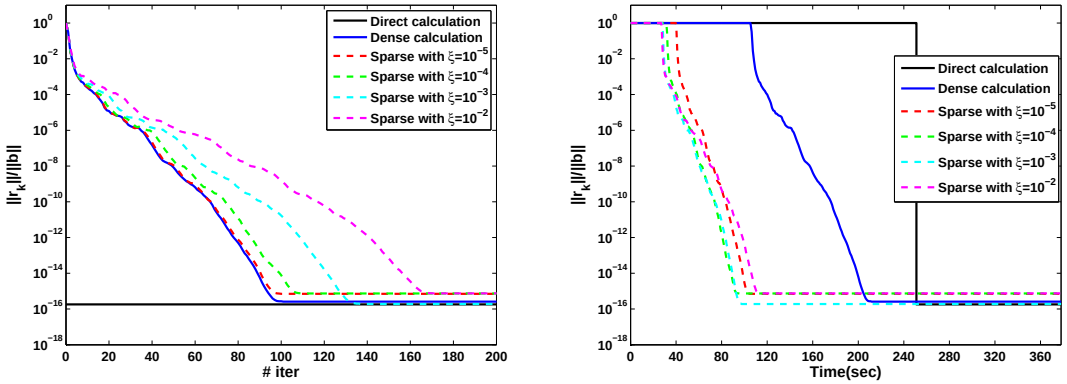


Figure 6.10: Comportement numérique de la variante creuse.

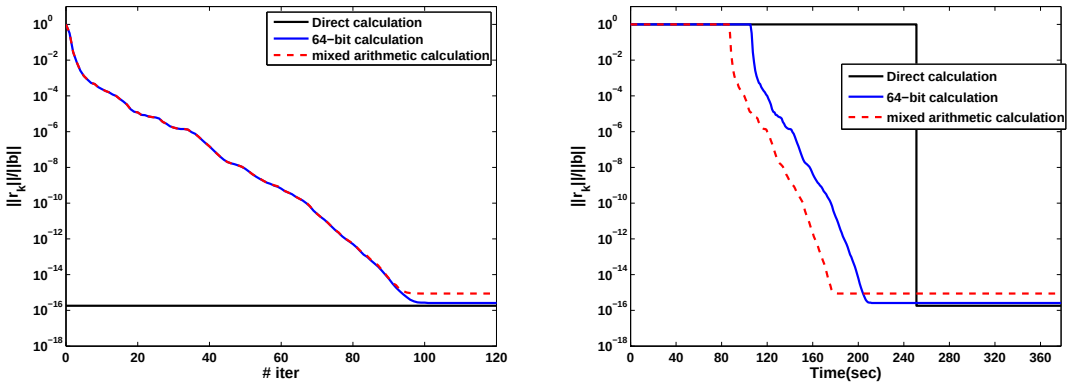


Figure 6.11: Comportement numérique de la variante précision mixte.

Une étude similaire aux chapitres précédents, sur l'influence de la sparsification (Figure 6.10) et de la précision mixte (Figure 6.11) a été effectuée. Les résultats observés sont prometteurs. Pour des valeurs optimales du paramètre de seuil, on observe un gain significatif en mémoire et en temps de calcul ce qui rend la variante creuse du préconditionneur très intéressante. De même pour la

variante mixte, surtout que ces applications nécessitent un espace de stockage énorme ; dans ce contexte économiser la moitié de cet espace de stockage ainsi qu'un gain en temps de calcul est aussi très appréciable.

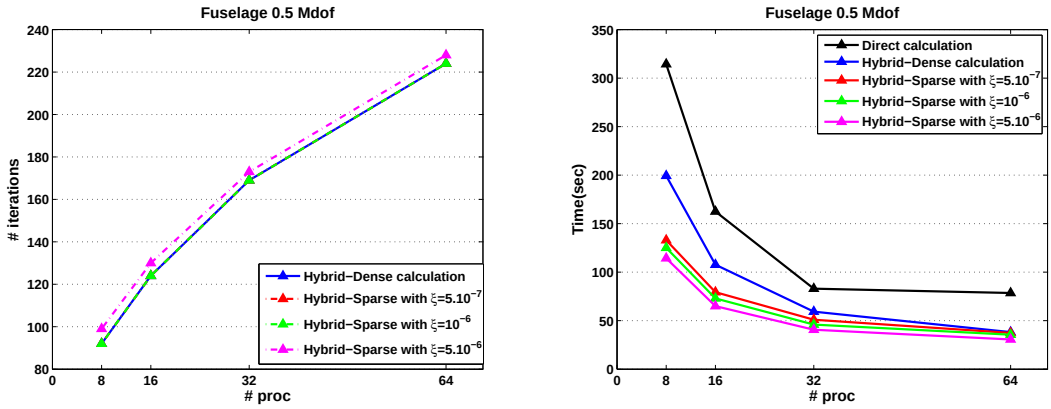


Figure 6.12: scalabilité numérique et paralléle performance.

Ensuite des études de performances numériques et parallèles ont été effectuées. Par exemple pour un problème de mécanique des structures, la Figure 6.12 illustre à gauche le nombre d'itérations lorsqu'on augmente le nombre de processeurs, tandis que celle de droite illustre le temps de calcul correspondant. On remarque l'avantage de la méthode hybride, la simulation peut être réalisée deux fois plus vite qu'une approche directe. Ainsi on peut aussi remarquer l'amélioration de la variante creuse du préconditionneur.

Pour des problèmes en imagerie séismique, on note que les méthodes directes ne peuvent pas être utilisées pour réaliser des grands simulations $3D$ à cause du stockage mémoire qui est prohibitif ; l'approche hybride offre une alternative prometteuse.

L'augmentation en nombre d'itérations et la taille mémoire nécessaire pour ces applications nous a incité à développer une approche exploitant deux niveaux de parallélisme. Nous présentons les détails des performances parallèles de notre approche exploitant deux nivaux de parallélisme.

La Figure 6.13 montre une comparaison entre un algorithme parallèle classique (*1-level*) et un algorithme utilisant le deux niveaux de parallélisme (*2-levels*). On note l'amélioration de la performance parallèle ; dans cette configuration l'algorithme à deux niveaux de parallélisme est presque deux fois plus rapide que les algorithmes parallèles classiques. Par ailleurs, le tableau 6.6 montre l'effet numérique des deux niveaux de parallélisme pour des applications d'imagerie séismique. On remarque qu'au lieu d'augmenter le nombre de sous-domaines lorsqu'on augmente le nombre de processeurs, il est préférable d'allouer plusieurs processeurs par sous-domaine en limitant le nombre de sous-domaines.

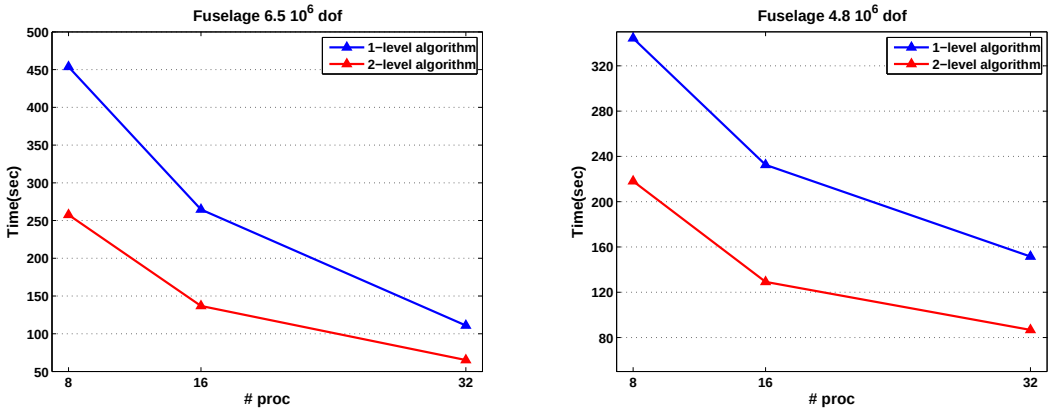


Figure 6.13: Performance paralléle de la méthode de deux niveaux de parallélisme.

Frequency equal to 7 Hz						
Available processors	Algo	# subdomains	Processors/ subdomain	# iter	Iterative loop	One right-hand side
$\approx$ 200 processors	*1-level parallel*	192	1	235	79.0	85.8
	2-level parallel	96	2	119	38.2	45.1
	2-level parallel	48	4	105	42.9	51.1
	2-level parallel	50	4	81	28.1	35.5
$\approx$ 100 processors	*1-level parallel*	96	1	119	57.0	61.1
	1-level parallel	98	1	148	66.7	66.7
	2-level parallel	48	2	105	62.1	67.8
	2-level parallel	50	2	81	39.1	45.1

Table 6.6: Performance numérique des deux niveaux de parallélisme (*2-level parallel* method) pour l'application $3D$ Overthrust SEG/EAGE.

Chapter 7

Preliminary investigations on structural mechanics problems

7.1 Introduction

Large-scale scientific applications and industrial numerical simulations are nowadays fully integrated in many engineering areas such as aeronautical modeling, structural mechanics, electrical simulation and so on. Those simulations often involve the discretization of linear or nonlinear PDE on very large meshes leading to systems of equations with millions of unknowns. The use of large high performance computers is mandatory to solve these problems.

In this chapter, we focus on a specific engineering area, the structural mechanics, where large problems have to be solved. Our purpose it to evaluate the robustness and possibly the performance of our preconditioner on the solution of the challenging linear systems that are often solved using direct solvers. In that respect we consider two different classes of problems.

The first one, is related to the solution of the linear elasticity equations with constraints such as rigid bodies and cyclic conditions. These constraints are handled using Lagrange multipliers, that give rise to symmetric indefinite augmented systems. Such linear systems are preferably solved using the MINRES [78] Krylov subspace method, that can be implemented using only a few vectors thanks to the symmetry property that enables the use of short recurrences. In our study, because we intend to perform comparisons in term of computing performance and also in term of accuracy, we preferred using GMRES that is proved backward stable.

The second class of problems, is still related to linear elasticity equations. The linear systems involved in such simulations are symmetric positive definite linear systems and solved using the conjugate gradient.

All the problems presented in this chapter have been generated using the Samcef V12.1-02 finite element software for nonlinear analysis, Mecano developed by Samtech http://www.samcef.com/.

7.2 Experimental framework

7.2.1 Model problems

In this chapter we consider a few real life problems from structural mechanics applications. Those examples are generated using the Samcef tool called Samcef-Mecano V12.1-02.

Samcef-Mecano is a general purpose finite element software that solves nonlinear structural and mechanical problems. It minimizes the potential energy using the displacement (translations and/or rotations) as unknowns. For each kinematic constraint (linear constraint or kinematic joint), a Lagrange multiplier is automatically generated. In order to have a good numerical behaviour,

an augmented Lagrangian method is used. The modified problem consists in finding the minimum of the potential F^* :

$$F^*(q) = F(q) + k\lambda\phi + \frac{p}{2}\Phi^T\Phi$$

where q is the degrees of freedom vector (translations and/or rotations), k is the kinematic constraint scaling factor, p is the kinematic constraint penalty factor, ϕ is the kinematic constraint vector and λ is the Lagrange multiplier vector.

The equations of motion of the structure discretized by finite elements take the general form

$$M\ddot{q} + f^{\text{int}} = f^{\text{ext}}$$

where the notation is simplified by including in the internal forces f^{int} the contribution of the kinematic constraints and that of the elastic, plastic, damping, friction, ... forces. Three types of analysis can be performed:

1. Static analysis;
2. Kinematic or quasi-static analysis;
3. Dynamic analysis.

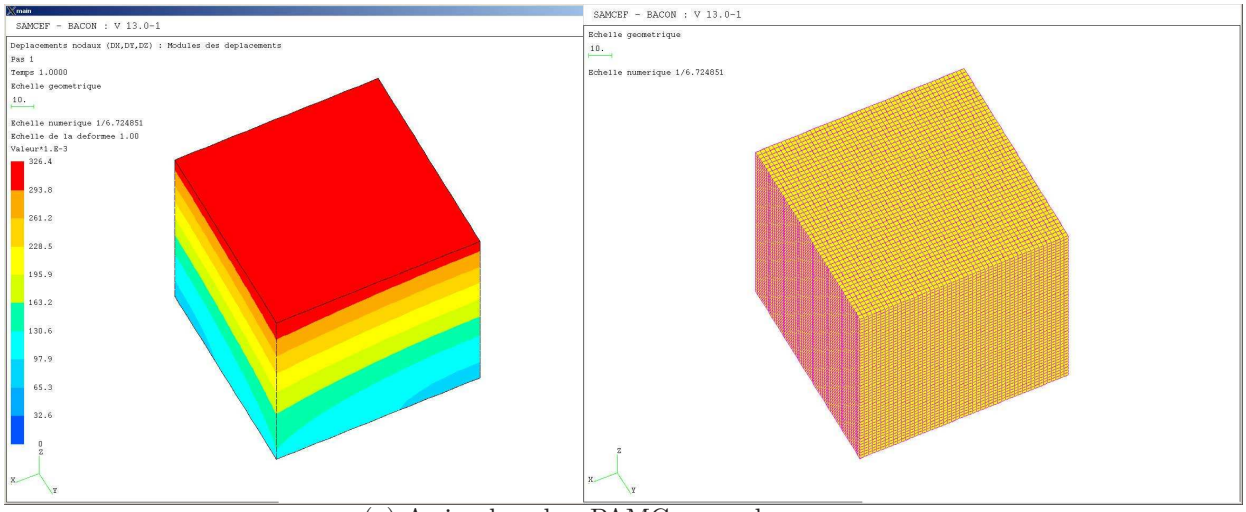
At each time step, a set of nonlinear equations has to be solved and a Newton-Raphson scheme is used in order to solve this nonlinear problem. These equations express the equilibrium of the system at a given time. In a static analysis, these equations take stiffness effects (linear or not) into account. In a kinematic analysis, the effects due to the kinematic velocities are added to the effects taken into account by the static analysis. Finally, the dynamic analysis takes all the effects of the kinematic analysis into account including also inertia effects, that do not appear in the static and the kinematic analysis.

For the numerical examples considered here, a static computation is performed. The materials are elastic: the relation between the stress and the strain is $\sigma = H\epsilon$ where H is the Hooks matrix. For each test case we run our solver on the matrix generated during the first iteration of the first time step.

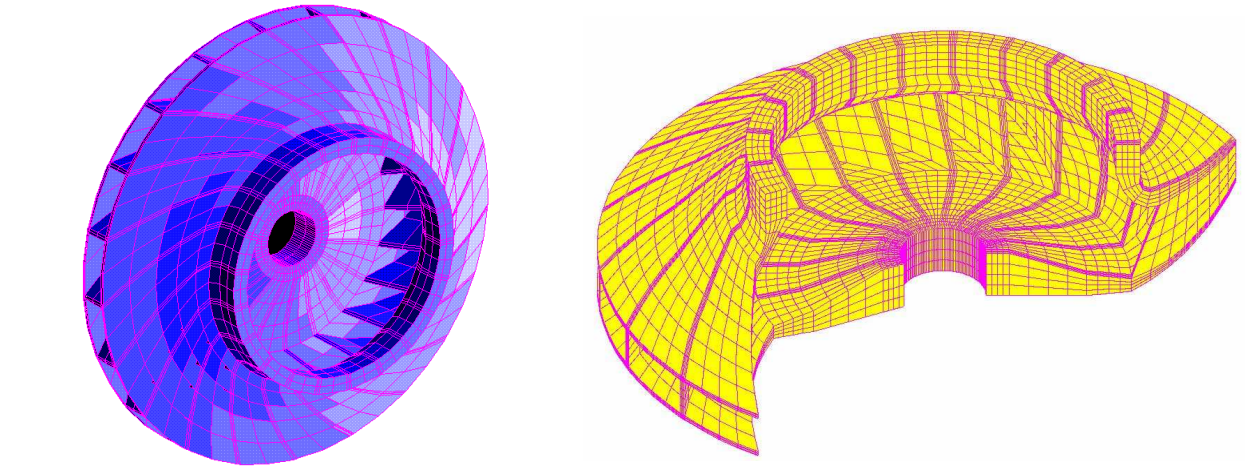
The geometry of the examples are displayed in Figure 7.1. The first corresponds to a simple cube (Figure 7.1 (a)) where no constraints are imposed. There are three unknowns per nodes that are the translations. Fixations are added in the planes $x = 0, y = 0$ and $z = 0$ on displacement coordinates x, y and z respectively. A uniform displacement is applied on the top of the cube. This test example is referred to as PAMC. The associated linear systems are symmetric positive definite.

A more realistic problem is displayed in Figure 7.1 (b), that is an impeller. This case represents a 90 degrees sector of an impeller. It is composed of 3D volume elements. Cyclic conditions are added using elements that link displacements of the slaves nodes on one side of the sector, to master facets on the other side of the sector. These conditions are taking into account using elements with 3 Lagrange multipliers. Angular velocities are introduced on the complete structure and centrifugal loads are computed on the basis of the angular velocities and of the mass representation. This example is called Rouet in the sequel, the associated linear system is symmetric indefinite.

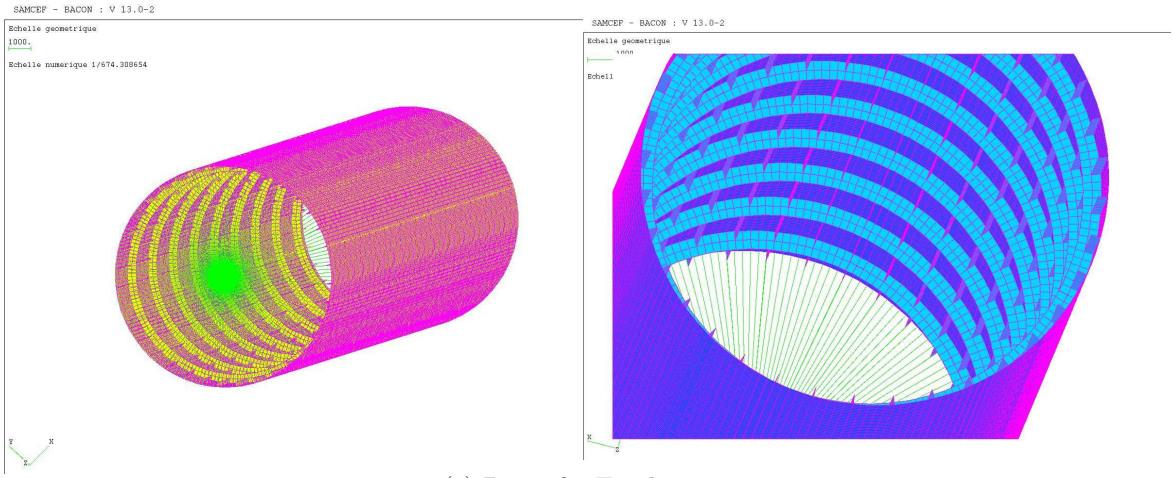
Lastly, a parameterized barrel (section of a fuselage) is depicted in Figure 7.1 (c). It is composed of its skin, stringers (longitudinal) and frames (circumferential, in light blue on Figure 7.1 (c)). Midlinn shell elements are used: each node has 6 unknowns (3 translations and 3 rotations). On one extremity of the fuselage all the degrees of freedom are fixed. On the other extremity a rigid body element is added: all the degrees of freedom of the nodes are linked to the displacement of the master node of the element. In order to represent this dependency Lagrange multipliers are added. A force perpendicular to the axis of the fuselage is applied on the master node. This last test example is referred to as Fuselage and the associated linear system is symmetric indefinite. The Fuselage example, although defined in 3D is more 2.5D rather than full 3D as the complete



(a) A simple cube: PAMC example.



(b) A wheel: Rouet example.



(c) Part of a Fuselage.

Figure 7.1: various structural mechanics meshes.

volume is not meshed.

In Table 7.1 we display for the different mesh geometries the various sizes of the problems we have experimented. For each problem, we give the number of finite elements and the number of degrees of freedom.

PAMC example		
# elements	# degrees of freedoms	# of Lagrange equations
PAMC50 125,000	$0.8 \cdot 10^6$	0
PAMC80 512,000	$3.2 \cdot 10^6$	0
Rouet example		
# elements	# degrees of freedoms	# of Lagrange equations
337,000	$1.3 \cdot 10^6$	13,383
Fuselage example		
# elements	# degrees of freedoms	# of Lagrange equations
500,000	$3.3 \cdot 10^6$	2,160
750,000	$4.8 \cdot 10^6$	2,592
1,000,000	$6.5 \cdot 10^6$	3,024

Table 7.1: Characteristics of the various structural mechanics problems.

7.2.2 Parallel platforms

Our target parallel machine is an IBM JS21 supercomputer installed at CERFACS to address diverse applications in science and engineering. It works currently with a peak computing performance of 2.2 TeraFlops. This is a 4-core blade server for applications requiring 64-bit computation. It is ideal for computer-intensive applications and transactional Internet servers.

This paragraph provides more detailed information about the IBM PowerPC 970MP microprocessor, that is the processor of the BladeCenter JS21.

- The BladeCenter JS21 leverages the high-performance, low-power 64-bit IBM PowerPC 970MP microprocessor.
- The 4-core configuration comprises two dual-core PowerPC 970MP processors running at 2.5 GHz.
- Each processor core includes 32/64 KB L1 (data/instruction) and 1 MB (non-shared) L2 cache.
- Each node is equipped with 8 GBytes of main memory.
- The AltiVec is an extension to the IBM PowerPC Architecture. It defines additional registers and instructions to support single-instruction multiple-data (SIMD) operations that accelerate data-intensive tasks.

The BladeCenter JS21 is supported by the AIX 5L, Red Hat Enterprise Linux, and SUSE Linux Enterprise Server (SLES) operating systems. This latter is installed on our experimental JS21. Distributed memory parallel applications might require the installation of a high-performance, low-latency interconnection network between BladeCenter JS21s. This requirement is supported through the use of Myrinet2000 network offering a bandwidth of 838 MBytes/sec between nodes and a latency of 3.2 μs. This platform is equipped by different software and scientific libraries such as:

- IBM compilers: XL Fortran and XL C/C++.

- IBM Engineering and Scientific Subroutine Libraries (ESSL4.2).
- MPI library (MPICH2 V0.971), LAM V7.1.1-3, and OpenMp V1.1.1-2.
- Mathematical Acceleration Subsystem (MASS) libraries.
- GNU Tool-chain (glibc, gcc, binutils, gdb).
- Java Runtime JRE 1.4.1.
- IBM LoadLeveler.
- IBM General Parallel File System (GPFS).
- FFTW 3.1.
- HDF5.
- NETCDF 3.6.1.

7.3 Partitioning strategies

When dealing with large sparse linear systems arising from the discretization of PDE's on large 3D meshes, most of the parallel numerical techniques rely on a partition of the underlying mesh. For finite element approaches, the partitioning is performed on the set of elements. In that respect, the dual graph of the mesh is split. In this graph, the vertices represent the elements and there is a edge between two vertices if the elements associated with these vertices share an node of the mesh. When the work per element is independent of the element, a good partitioning should aim at splitting the graph into subgraphs having comparable numbers of vertices while minimizing the size of the interfaces between the subgraphs. This latter constraint is often seen as a way to reduce the amount of communication between the subgraphs/subdomains; in our case it mainly means reducing the size of the Schur complement system to be solved. In that framework, a good partitioning should attempt to balance and minimize the sizes of the complete interface associated with each subdomain; i.e., balance the size of the local Schur complement matrices, as the preconditioner cost (setup and application) and parallel efficiency mainly depend on a good balance of them.

For linear systems involving Lagrange multipliers an additional constraint should be taken into account. If the mesh is decomposed without considering the Lagrange multipliers we might end-up with a splitting of the mesh for which Lagrange multipliers coupled unknowns that are on the interface while the Lagrange multiplier is considered as an "internal" unknown. In such a situation, the matrix associated with the internal unknowns has a zero row/column and is consequently structurally singular. The Schur complement does not exist and the hybrid technique breaks down. A simple and systematic way to fix this weakness it to enforce the Lagrange multipliers to be moved into the interface. If the partitioner has produced balanced subgraphs with minimal interfaces, moving the Lagrange multipliers into the interfaces significantly deteriorates the quality of the partition. A good partitioning strategy should then, balance the size of the subgraphs while minimizing and balancing the interface sizes but also balance the distribution of the Lagrange multipliers among the subgraphs. In that respect, when the Lagrange multipliers are moved into the interfaces, the interfaces remain balanced.

In order to achieve this, we do not apply a graph partitioner on the dual graph of the mesh but add some weights to the vertices. The mesh partitioner we use is Metis [62] routine metis_partgraphVKway that enables us to consider two weights per vertex, one associated with the workload ($weight\_vertex$) and the other with the amount of communication ($weight\_comm$). In order to balance the Lagrange multipliers among the subgraphs, for the elements with Lagrange multipliers we relax the weight associated with the communication (i.e. communication weight set to zero) and penalize

their workload by setting their work weight to a large value. For the other elements, the work weights are set proportional to the number of degrees of freedom associated with them (*ndof*), while the communication weight is set to the number of adjacent elements (degree of the vertices associated with the element in the dual graph *nadj*). Among the numerous weighted variants that we have experimented with, this following was the best we found:

$$\begin{cases} weight\_comm(element) = 0 & if \text{ element } contains\ Lagrange\ multipliers, \\ weight\_comm(element) = nadj & otherwise, \end{cases}$$

$$\begin{cases} weight\_vertex(element) = large\ value & if \text{ element } contains\ Lagrange\ multipliers, \\ weight\_vertex(element) = ndof & otherwise. \end{cases}$$

Thus, the objective is to try to fit the maximum of unknowns associated with Lagrange equations into the interface and to minimize the number of these equations belonging to one subdomain. With this strategy, we first attempt to have as much Lagrange unknowns as possible in the interface. Secondly, we try to have the remaining ones equitably distributed among the subdomains. This latter feature enables us to preserve a good balance of the interface even after these Lagrange unknowns are moved into the subdomain interfaces as treatment of the possible singularity.

		Maximal interface size	Init time	Preconditioner setup	# iter	Iterative loop	Total time
PAMC50 0.8 Mdof	Poor	21187	86	631	55	479	1196
16 subdomains	Better	14751	91	200	40	70	361
PAMC50 0.8 Mdof	Poor	12559	23	135	69	106	264
32 subdomains	Better	11119	22	83	50	53	158
ROUET 1.3 Mdof	Poor	13492	141	255	76	77	473
16 subdomains	Better	10953	67	137	79	61	264
ROUET 1.3 Mdof	Poor	11176	21	141	108	80	242
32 subdomains	Better	7404	23	44	106	45	111
Fuselage 6.5 Mdof	Poor	11739	35	168	162	144	347
32 subdomains	Better	6420	36	30	176	58	124
Fuselage 6.5 Mdof	Poor	10446	15	120	217	170	305
64 subdomains	Better	4950	13	15	226	54	82

Table 7.2: Partitioning effect for various structured/unstructured problems when decomposed differently.

The effect of the partitioning quality on the efficiency of the parallel hybrid solver is illustrated in Table 7.2. In that table “poor partitioning" corresponds to un-weighted graph partitioning for the problems with Lagrange multipliers; “better partitioning" refers to weighted graph partition using Metis and its communication volume minimization option. For the PAMC example, where there are no Lagrange multipliers, “poor partitioning" corresponds to a splitting of the mesh using the Metis option based on the edges-cut minimization heuristic, while “better partitioning" corresponds to Metis splitting based on the volume communication minimization heuristic. This latter approach often generates partition with smaller interface sizes. As shown in Table 7.2, poor load balance causes inadequate performance of the algorithm. The main drawback is that the number of elements assigned to each processor as well as the local interface sizes vary uncommonly. Thus the local Schur complement sizes highly vary causing unbalanced preconditioner setup where the fastest processors should wait the slowest one before starting the iterative step. It also induces large idle time at each global synchronization points implemented for the calculation of the dot-product (global reduction) in the iterative process. Moreover, the computing and memory cost of the preconditioner is related to the interface size, thus large subdomain interfaces imply inefficiency

in both computing time and memory required to store the preconditioner. The importance of this latter is clearly exposed in Table 7.2. On the first hand, we can see that for large subdomain sizes, the preconditioner setup time, which cost increases as function of $O(n^3)$, becomes very expensive. On the other hand, the iterative loop time depends closely on the matrix-vector product and from the preconditioner application costs. For large interfaces (large local Schur complements), both the matrix-vector calculation and the preconditioner application become expensive (because unbalanced), thus increasing the overall computing time and requiring a large amount of data storage.

As consequence, it is imperative to strive for a very balanced load. We have resorted these problems by the use of multiple constraints graphs partitioning. In our numerical experiments, the partitioning is applied with weighted vertices based on a combination of the characteristics described above.

7.4 Indefinite symmetric linear systems in structural mechanics

7.4.1 Numerical behaviour of the sparsification

While the primary purpose of this section is to focus on the way the numerical performance of the sparse preconditioner can be stated, it also gives us tips for describing both computational benefits and data storage of the sparse algorithm. In this section we first investigate the advantages in term of setup cost and memory storage associated with the use of the sparsification strategy for the preconditioner. Then we focus on the numerical behaviour of the resulting preconditioners.

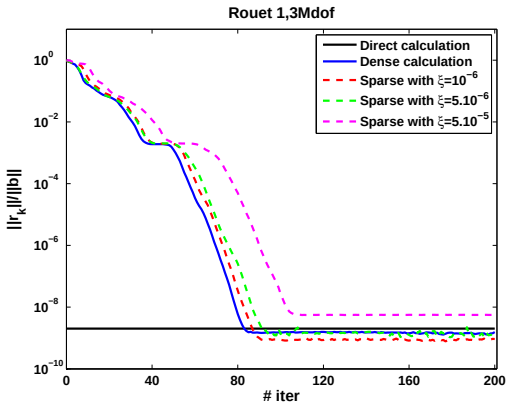
We report results for an unstructured mesh with 1 million finite elements (6.5 million dofs) for the Fuselage test case, and on a unstructured mesh of 340,000 finite elements (1.3 million dofs) for the Rouet test case. We display in Table 7.3, the memory space and the computing time required by the preconditioner on each processor for different values of the sparsification dropping parameter ξ. The results presented in this table are for both test cases with 16 subdomains. The maximal local subdomain interface size for the Fuselage problem on this decomposition has 9444 unknowns whereas for the Rouet problem it has of 10953 unknowns. It can be seen that a lot of storage can be saved. This also gives rise to a great time saving in the preconditioner setup phase.

ξ	0	5.10^{-7}	10^{-6}	5.10^{-6}	10^{-5}	5.10^{-5}
Rouet problem with 1.3 Mdof						
Memory	960_{MB}	384_{MB}	297_{MB}	153_{MB}	105_{MB}	48_{MB}
Kept percentage	100%	40%	31%	16%	11%	5%
Preconditioner setup	137	145	96	37	26	11
Fuselage problem with 6.5 Mdof						
Memory	710_{MB}	122_{MB}	92.7_{MB}	46.3_{MB}	35.6_{MB}	17.8_{MB}
Kept percentage	100%	17%	13%	7%	5%	2.5%
Preconditioner setup	89	26	19.5	10.8	8.8	5.8

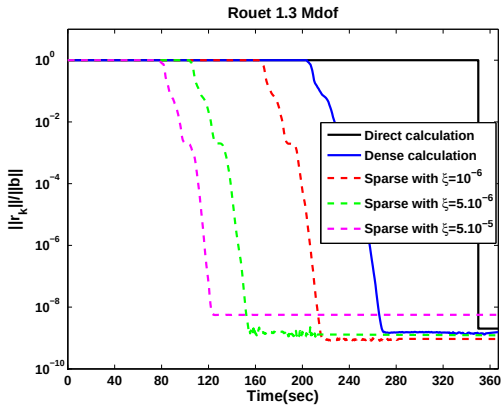
Table 7.3: Preconditioner computing time (*sec*) and amount of memory (MB) in $M_{\text{sp}-64}$ v.s. $M_{\text{d}-64}$ for various choices of the dropping parameter, when the problems are mapped onto 16 processors.

The cost of the preconditioner is not the only component to consider for assessing its interest. We should analyze its numerical performance. For that purpose, we report in Figure 7.2, the convergence history for various choices of ξ depicted in Table 7.3, for both the Fuselage and the

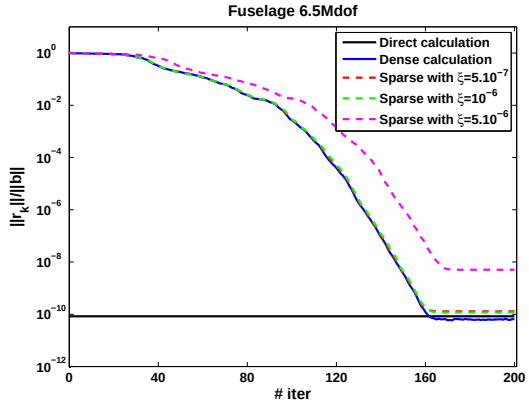
Rouet problem. For both test cases, we depict on the left-hand side of the Figure the convergence history as a function of the iterations, and on the right-hand side, the convergence history as a function of the computing time. For the sake of completeness, we also report on the performance of a parallel sparse direct solution, that can be considered for these sizes of problems. It is clear that the sparsified variant outperforms its dense counterpart. However, for these real engineering problems, the sparse preconditioner has to retain more information about the Schur complement than for the academic cases. For these problems, in order to preserve the numerical quality we need to keep more than 10% of the Schur entries whereas 2% in the academic case were sufficient.



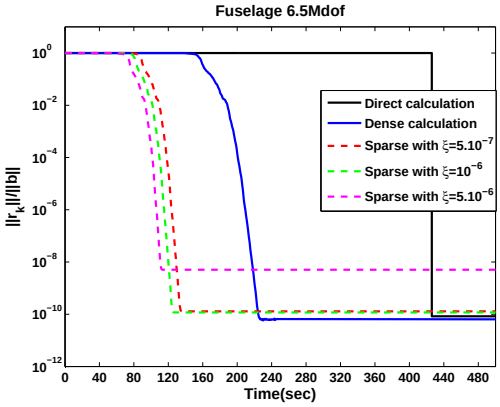
(a) Rouet 1.3Mdof (history v.s. iterations).



(b) Rouet 1.3Mdof (history v.s. time).



(a) Fuselage 6.5Mdof (history v.s. iterations).



(b) Fuselage 6.5Mdof (history v.s. time).

Figure 7.2: Convergence history of full-GMRES for Fuselage and Rouet problems mapped onto 16 processors, of the direct, the Hybrid-dense ($M_{\mathrm{d}-64}$) and the Hybrid-sparse ($M_{\mathrm{sp}-64}$) solvers for various sparsification dropping thresholds (Left: scaled residual versus iterations, Right: scaled residual versus time).

One can observe that the attainable η_b accuracies are different for the various choices of ξ. The largest effect can be seen on the Fuselage test case (bottom graphs). The explanation is as follows. The normwise backward stability of GMRES is established for $\eta_{AM,b}(y) = \frac{\|r^k\|}{\|AM\|\|y\|+\|b\|}$ that can become of the order of the level of machine precision. Consequently, if this quantity was plotted it would be the same for all the preconditioners. They would exhibit a plateau at the same level. In our experiments, because $\|AM\|$ was expensive to compute, we only display $\eta_b(y)$. Consequently, the attainable accuracy (i.e., the level of the plateau) depends on $\|AM\|$

that varies with ξ. To conclude on this accuracy aspect, we mention that the normwise backward error $\eta_b(x)$ for the solution computed with the direct solver is comparable to the one obtain with the hybrid techniques. We point out that the componentwise backward error is much smaller (of order $5.39 \cdot 10^{-14}$) as expected for Gaussian elimination with partial pivoting.

In term of computing time, we can see that the sparse preconditioner setup is more than 3 times faster than the dense one. For example if we look at Table 7.3 for the Fuselage example, we can see that for $\xi = 10^{-6}$ it is 4 times faster than the dense one (19.5 v.s. 89 seconds). In term of global computing time it can be seen that the sparse algorithm is about twice faster. The very few extra iterations introduced by the sparse variant are compensated by a faster iteration due to the reduced floating-point operations associated with it.

For the Rouet test case the sparse variants behave comparably as for the Fuselage problem. For very small values of ξ (for example $\xi = 5.10^{-7}$), the sparse preconditioner retains a large amount of entries, more than 30% of the elements. The sparse computational cost to setup the preconditioner becomes expensive, even more expensive than the dense one. For example, see Table 7.11, the cost of the sparse preconditioner for $\xi = 5.10^{-7}$ on 8 or 16 subdomains is respectively 283 seconds and 145 seconds whereas it is respectively 235 seconds and 137 seconds for the dense preconditioner. Furthermore, we can point that the gap in the number of iterations between very small choices of ξ (for example $\xi = 5.10^{-7}$ on the Rouet Table 7.11) and a reasonable choice of ξ (for example $\xi = 5.10^{-6}$ on the same Rouet example) is already small. A good tuning of ξ (for example $\xi = 5.10^{-6}$ on the same Rouet problem), can lead to the best ratio between computational cost and numerical solution performance. On the Rouet example in Table 7.11, for the decomposition into 16 subdomains, the number of iterations for $\xi = 5.10^{-7}$ is 80 and it is 87 for $\xi = 5.10^{-6}$, while the global computational cost is respectively 262 seconds and 151 seconds. Finally, we should mention that the componentwise backward error associated with the solution computed by the direct method for this example is about $9.17 \cdot 10^{-11}$.

7.4.2 Parallel performance

In this section we report on parallel experiments. For indefinite systems we choose full-GMRES as Krylov solver and consider the ICGS (Iterative Classical Gram-Schmidt) orthogonalization variant The initial guess is always the zero vector and convergence is detected when the normwise backward error becomes less than 10^{-8} or when 300 steps have been unsuccessfully performed.

7.4.2.1 Numerical scalability on parallel platforms

In this subsection we describe how both preconditioners ($M_{\mathrm{d}-64}$ and $M_{\mathrm{sp}-64}$) affect the convergence rate of the iterative hybrid solver and what numerical performance is achieved. Hence, we report test cases for different sizes of the Fuselage problem; only one Rouet problem size is considered.

In Table 7.4 we display the number of iterations obtained for different problem sizes of the Fuselage test case. Each original problem is split into 4, 8, 16, 32, and 64 subdomains expect for the problem with about 1 million elements that does not fit into the memory available on 4 processors. First we test the quality of the sparse preconditioner $M_{\mathrm{sp}-64}$ generated by varying the sparsification threshold ξ and compare the results to the dense preconditioner $M_{\mathrm{d}-64}$. We also indicate in Table 7.4 the percentage of kept entries in the sparse preconditioner for each value of ξ and for the different decompositions, and for different Fuselage problem sizes. The results show that the sparse preconditioner convergence is similar to the one observed using the dense preconditioner. The Fuselage is a relatively difficult problem; when increasing the number of subdomains, the reduced system (global interface system) becomes larger with high heterogeneity. The values of the entries varies by more than 15 order of magnitude. It is much more difficult to compute the solution, thus the small increase in the number of iterations when increasing the number of subdomains is not surprising. The attractive feature is that both preconditioners still

# processors		4	8	16	32	64
$500{\cdot}10^3 elements$ $3.3{\cdot}10^6 dof$	$M_{\mathrm{d}-64}$	38	92	124	169	224
	$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-7}$)	39 (17%)	92 (18%)	124 (22%)	169 (29%)	224 (38%)
	$M_{\mathrm{sp}-64}$ ($\xi = 10^{-6}$)	40 (13%)	92 (14%)	124 (17%)	169 23%()	224 (31%)
	$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-6}$)	51 (6%)	99 (7%)	130 (9%)	173 (13%)	228 (18%)
$800{\cdot}10^3 elements$ $4.8{\cdot}10^6 dof$	$M_{\mathrm{d}-64}$	20	94	122	168	232
	$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-7}$)	24 (17%)	94 (15%)	117 (20%)	168 (25%)	232 (33%)
	$M_{\mathrm{sp}-64}$ ($\xi = 10^{-6}$)	25 (12%)	94 (11%)	123 (15%)	168 (20%)	232 (26%)
	$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-6}$)	41 (6%)	104 (6%)	134 (8%)	177 (11%)	242 (15%)
$10^6 elements$ $6.5{\cdot}10^6 dof$	$M_{\mathrm{d}-64}$	-	98	147	176	226
	$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-7}$)	-	99 (13%)	147 (17%)	176 (22%)	226 (30%)
	$M_{\mathrm{sp}-64}$ ($\xi = 10^{-6}$)	-	101 (10%)	148 (13%)	177 (18%)	226 (24%)
	$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-6}$)	-	121 (5%)	166 (7%)	194 (9%)	252 (13%)

Table 7.4: Number of preconditioned GMRES iterations and percentage of kept entries for the different Fuselage problems. The number of processors is varied for the various variants of the preconditioner and for various choice of ξ. "-" means that the result is not available.

# processors		8	16	32	64
$337{\cdot}10^3 elements$ $1.3{\cdot}10^6 dof$	$M_{\mathrm{d}-64}$	59	79	106	156
	$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-7}$)	59 (31%)	80 (40%)	107 (45%)	156 (56%)
	$M_{\mathrm{sp}-64}$ ($\xi = 10^{-6}$)	59 (24%)	83 (31%)	108 (39%)	157 (47%)
	$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-6}$)	60 (11%)	87 (16%)	114 (21%)	162 (27%)
	$M_{\mathrm{sp}-64}$ ($\xi = 10^{-5}$)	63 (7%)	89 (11%)	116 (15%)	166 (20%)
	$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-5}$)	70 (3%)	103 (5%)	131 (7%)	191 (9%)

Table 7.5: Number of preconditioned GMRES iterations and percentage of kept entries for the Rouet problem with about 500,000 elements and 1.3 Mdof. The number of processors is varied for the various variants of the preconditioner and for various choices of ξ. "-" means that the result is not available.

achieve the same backward error level even for large number of subdomains. Moreover, we study the effect of increasing the size of the mesh, that leads to increasing the subdomain size for a fixed number of processors (vertical reading in the Table 7.4). The growth in the subdomain size keeping fixed the number of processors only has a very slight effect on the convergence rate. For example, we can observe that when increasing the size of the Fuselage problem from 500,000 elements to one million elements for a fixed number of subdomains let say 32 subdomains, the numbers of iterations remain around 170 iterations for both preconditioners.

A similar analysis was performed for the Rouet test case, investigating the effect of domain decomposition. We report in Table 7.5, results when varying the number of subdomains from 8 to 64 for a fixed mesh size of 337,000 elements. As expected, the sparse preconditioner performs as well as the dense preconditioner $M_{\mathrm{d}-64}$, for the different decomposition considered here. In Table 7.5, we display the percentage of kept entries in the sparse preconditioner for each value of ξ and for each decomposition. We can observe that for percentages between 10% to 20%, the sparse preconditioner behaves closely to the dense one for all decompositions. The gap in the number of iterations is between 1 and 5 for the most difficult cases, whereas the sparse variants save a lot of computing resources as described in the next subsection. Regarding the number of iterations when increasing the number of processors, we can still observe, as in the Fuselage test case, a slight growth in the iteration numbers. For example when we increase the number of subdomains 8 times, the iteration number is multiplied by 2.8.

To conclude on this aspect, we would like to underline the fact that either the dense preconditioner $M_{\mathrm{d}-64}$, or the sparse variant $M_{\mathrm{sp}-64}$ are able to ensure fast convergence of the Krylov solvers on our test cases of structural mechanical applications, and even when increasing the number of subdomains. More precisely some tests on the Fuselage with one million elements were performed on more than 64 processors. The results show that the preconditioner still guarantee reasonable numerical performance (for example 275 iterations on 96 processors).

7.4.2.2 Parallel performance scalability

This subsection is devoted to the presentation and analysis of the parallel performance of both preconditioners. A brief comparison with a direct method solution is also given. It is believed that parallel performance is the most important means of reducing turn around time and computational cost of real applications. In this context, we consider experiments where we increase the number of processors while the size of the initial linear system (i.e., mesh size) is kept constant. Such experiments mainly emphasize the interest of parallel computation in reducing the elapsed time to solve a problem of a prescribed size.

For the sake of completeness, we report in Table 7.8 to Table 7.11 a detailed description of the computing time for all problems described above, for both preconditioners, and for different choices of the dropping parameter ξ. We also report the solution time using the parallel sparse direct solver, where neither the associated distribution or redistribution of the matrix entries, nor the time for the symbolic analysis are taken into account in our time measurements. The main aim of this detailed presentation is to evaluate the performance of the three mains phases of the hybrid solver in a very comprehensive way. We recall the main three phases of the method:

- *Phase1:* the initialization phase that is the same for all the variants of the preconditioners. It consists into the factorization of the local internal problem and the computation of the Schur complement. It depends only on the size of the local subdomains and on the size of the local Schur complements;
- *Phase2:* the preconditioner setup phase that differs between the dense and the sparse variants. It depends also on the size of the local Schur complements, and on the dropping parameter ξ for the sparse variants;
- *Phase3:* the iterative loop which is related to the convergence rate and to the time per iteration. This latter depends on the efficiency of the matrix-vector product kernel (explicit v.s.

implicit), and on the preconditioner application, that is the forward/backward substitutions (dense v.s. sparse).

Initialization *phase1*

The results of the parallel efficiency of the initialization phase (*Phase1*) of all test cases when increasing the number of processors are given in Table 7.8 to Table 7.11. It consists in the factorization time of the local internal problem associated with each subdomain, and on the computing time of the local Schur complement using the MUMPS package. Because the problem (mesh) size is fixed, when we increase the number of processors, the subproblems become smaller and the initialization times decrease in a superlinear manner. We can observe this superlinear speedup especially for very large problems as Fuselage with 1 million elements. For this test case the initialization time decreases from 223 seconds on 8 processors down to 13 seconds on 64 processors. Although less important, the same trend was observed on the other test cases.

We note that to compute our preconditioner we need an explicit calculation of the local Schur complement. Among the few available parallel distributed direct solvers, MUMPS offers a unique feature, which is the possibility to compute the Schur complements defined in equation (7.1) using efficient sparse calculation techniques:

$$\mathcal{S}_i = \mathcal{A}_{\Gamma_i\Gamma_i} - \mathcal{A}_{\Gamma_i\mathcal{I}_i}\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}^{-1}\mathcal{A}_{\mathcal{I}_i\Gamma_i}. \tag{7.1}$$

There are several advantages of this approach. It enable us to construct our preconditioner that is based on the explicit form of the Schur complement. It is also easy to build either mixed precision or sparsified preconditioner. In addition, in the explicit case, the matrix-vector product needed at each iteration step of the Krylov solver is performed by a call to the high performance BLAS-2 DGEMV routine whereas it needs two sparse triangular solves in the implicit case. On the other hand, there are also some drawbacks for this method. First, the factorization step takes more time as we have more floating-point operations to perform in order to compute the Schur complement. This method also requires some additional storage to hold the local Schur complement as a dense matrix.

In order to compare the two approaches, we report in Table 7.6, the computing time to factorize the $\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}$ matrix with or without Schur computation. Those results correspond to the Fuselage test cases, for the different decompositions.

# processors		8	16	32	64
Fuselage $4.8{\cdot}10^6 dof$	Interface size	11004	10212	8022	4338
	Interior size	595914	278850	133122	73056
	explicit ($\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}$ + Schur)	131	51	22	9
	implicit ($\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}$)	53	19	9	4
Fuselage $6.5{\cdot}10^6 dof$	Interface size	12420	9444	6420	4950
	Interior size	806712	381804	201336	99060
	explicit ($\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}$ + Schur)	218	48	35	12
	implicit ($\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}$)	95	30	13	6

Table 7.6: Parallel elapsed time (sec) for the factorization of $\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}$ with or without Schur complement.

If we compare the factorization of $\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}$, with or without Schur, it is easy to see that, even if the factorization step takes more time, only a small number of Krylov iterations is usually enough to make the explicit method more efficient than the implicit one as illustrated in the next paragraph related to the time per iteration.

Preconditioner setup *phase2*

In order to study the preconditioner costs of our algorithms, we report in Table 7.8 to Table 7.11 the required time to build the preconditioner. This cost includes the time to assemble the local

Schur complement, and to factorize the assembled dense Schur for $M_{\mathrm{d}-64}$, or to sparsify and then to factorize the resulting sparse assembled Schur for $M_{\mathrm{sp}-64}$. To assemble the preconditioner, neighbour to neighbour communication is performed to exchange informations with processors owning neighbouring regions. The key is that this communication cost is relatively small compared to the factorization of the preconditioner. The assembling part of the preconditioner setup has little effect even when increasing the number of processors.

Regarding the factorization time, we can again see the advantage of the sparse variants of the preconditioner. This is especially noticeable for small numbers of subdomains where the preconditioner size is large. In this case, it is interesting to note that the sparse variant can build the preconditioner more than 3 times faster than the dense $M_{\mathrm{d}-64}$ counterpart. For example in Table 7.8, the sparse preconditioner setup with $\xi = 5.10^{-6}$, is five times faster as the dense algorithm on 8 or 16 processors. However, the performance is related to the percentage of kept entries in the preconditioner. So for very small values of the dropping parameter ξ a lot of entries are kept, the sparse factorization of the preconditioner becomes expensive, sometimes more expensive than the dense $M_{\mathrm{d}-64}$. This can be observed for example in Table 7.11, where, with 8 processors, the setup costs 283 seconds for $\xi = 5.10^{-7}$ about 220 seconds for $\xi = 10^{-6}$, whereas its cost is 235 seconds for the dense $M_{\mathrm{d}-64}$. For the same case, it costs 43 seconds for $\xi = 10^{-5}$, the resulting sparse preconditioner converges with only 4 extra iterations compared to the dense preconditioner. From a speedup point of view, it can be seen that as more and more processors are added, the size of the preconditioner components becomes smaller and thus the cost of the computation becomes faster. Because LAPACK algorithms are used to factorize the dense preconditioner $M_{\mathrm{d}-64}$, it is not surprising that the computing cost of the preconditioner has a superlinear speedup when increasing the number of subdomains. It is well known that the number of floating-point operations of a dense LU factorization is in order of $O(n^3)$, thus decreasing the size of n leads to a superlinear speedup. For similar reasons, superlinear speedups are also observed for the sparse preconditioners.

Iterative loop *phase3*

We study now the performance of the iterative loop. We report the average of the time per iteration required by each of the problem described above to converge when increasing the number of processors. The iterative kernel is divided into three steps: the matrix-vector product, the preconditioner application, and the dot-product calculation. For the matrix-vector product, each processor computes the matrix-vector product and updates the results of only the interface in its region. So, communication here is performed each step only between processors sharing an interface. The matrix-vector product $\mathcal{S}x_{\Gamma}^{(k)}$ is a common step to all variants. It can be performed explicitly, if local Schur complement matrices are explicitly built and stored in memory; an implicit calculation can be implemented otherwise. The Schur complement is defined by

$$\mathcal{S}_i = \mathcal{A}_{\Gamma_i\Gamma_i} - \mathcal{A}_{\Gamma_i\mathcal{I}_i}\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}^{-1}\mathcal{A}_{\mathcal{I}_i\Gamma_i}. \tag{7.2}$$

In the implicit approach, the factors of $\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}$ are used to perform the local matrix-vector products for the local Schur complement defined by equation (7.2). This is done via a sequence of sparse linear algebra computations, namely a sparse matrix-vector product by $\mathcal{A}_{\mathcal{I}_i\Gamma_i}$, then sparse forward/backward substitutions using the computed factors of $\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}$, and finally a sparse matrix-vector product by $\mathcal{A}_{\Gamma_i\mathcal{I}_i}$.

In the explicit case, it consists in a single call to DGEMV, the dense level 2 BLAS subroutine that implements a dense matrix-vector product. In this case, the number of floating-point operations might be smaller and the access to the memory is more regular (i.e., dense versus sparse calculation), this explains the large decrease of computing time observed when using the explicit matrix-vector product. In all our experiments we consider the use of the explicit approach.

In order to compare the two approaches, we report in Table 7.7, the time spent in the matrix-vector product for both explicit and implicit cases, for all experiments. For the Fuselage example (that is 2.5D rather than full 3D), it is clear that the use of the explicit approach is the fastest. In the implicit case, the core of the matrix-vector product needs two sparse triangular solves on the internal unknowns of each subdomain. So it is related first, to the ratio between the number

of unknowns belonging to the interior and the number of unknowns belonging to the interface of the subdomains. For example on the Fuselage of $6.5 \cdot 10^6 dof$, the ratio $\frac{interior}{interface}$ is around 65 on 8 subdomains and 20 on 64 subdomains; that is, the number of interior unknowns is very large compared to those on the interfaces. Similar ratio can be observed on the other Fuselage test cases. As results, the growth in the matrix-vector product time, when the size of the local subdomain increases, is rather pronounced for the implicit case, whereas it is rather moderate for the explicit case. On those examples, the explicit calculation clearly outperforms the implicit one; for the Fuselage test with 1 million elements, on 8 processors, the explicit variant is 5 times faster than the implicit one. For the overall iterative loop it enables us to reduce the time from 270 seconds for implicit calculation down to 94,1 seconds for the explicit one.

For the Rouet problem, the local subdomain sizes are smaller compared with the Fuselage test problems. The ratio between the interior and the interface unknowns is smaller. It is around 10 on 8 subdomains and around 3.7 on 32 and 64 subdomains. In this case, the number of interior unknowns is comparable with the number on the interfaces. Consequently the backward/forward substitutions perform comparably to a dense matrix-vector product kernel on the interface. As a result the gap between explicit and implicit is reduced, but the explicit approach still outperforms the implicit one.

Regarding the preconditioning step, it is still clear that the sparsified variant is of great interest as it reduces considerably the time to apply the preconditioner, which leads to a significant reduction of the time per iteration compared to the dense counterpart.

At each iteration, the third step also performs global reduction. It was observed that, the relative cost of this reduction is negligible compared to the other steps of the algorithm, it increases by less than 2.10^{-3} seconds when increasing the number of processors from 8 to 64.

Thus, by looking at the time per iteration, we might conclude that the extra number of iterations cost introduced when increasing the number of subdomains is almost (in most cases) compensated by the cheaper cost of the resulting time per iteration. Regarding the sparse variants, we notice a significant gain in computing time for suited dropping thresholds. We should mention that dropping too many entries often lead to a significant increase of the iterative loop time as the number of iterations grows notably. For example in Table 7.8 for $\xi = 5.10^{-6}$. On the other side, only dropping a very few entries (very small ξ) also leads to higher time per iteration than a dense approach. For example in Table 7.11 for $\xi = 5.10^{-7}$. A good trade-off between numerical robustness and fast calculation should be found to ensure the best performance of the sparsified approach.

Finally, we compare in Figure 7.3 and Figure 7.4, the two variants of the preconditioners and

# processors		4	8	16	32	64
Rouet $1.3{\cdot}10^6 dof$	*explicit*	-	0.43	0.30	0.16	0.09
	implicit	-	1.06	0.54	0.27	0.16
Fuselage $3.3{\cdot}10^6 dof$	*explicit*	0.40	0.28	0.16	0.06	0.04
	implicit	2.12	1.00	0.49	0.31	0.17
Fuselage $4.8{\cdot}10^6 dof$	*explicit*	-	0.32	0.29	0.20	0.05
	implicit	-	1.51	0.71	0.42	0.21
Fuselage $6.5{\cdot}10^6 dof$	*explicit*	-	0.42	0.30	0.13	0.08
	implicit	-	2.21	1.01	0.61	0.32

Table 7.7: Parallel elapsed time (sec) for one matrix-vector product step. "-" means that the result is not available.

	Total solution time			
# processors	8	16	32	64
Direct	655.5	330.0	201.4	146.3
$M_{\mathrm{d}-64}$	525.1	217.2	124.1	82.2
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-7}$)	338.0	129.0	94.2	70.2
$M_{\mathrm{sp}-64}$ ($\xi = \ 10^{-6}$)	322.8	120.1	87.9	65.1
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-6}$)	309.8	110.9	82.8	63.2
	Time in the iterative loop			
# processors	8	16	32	64
$M_{\mathrm{d}-64}$	94.1	77.9	58.1	54.2
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-7}$)	59.4	52.6	42.2	42.9
$M_{\mathrm{sp}-64}$ ($\xi = \ 10^{-6}$)	57.6	50.3	38.9	40.7
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-6}$)	60.5	49.8	40.7	45.4
	# iteration			
# processors	8	16	32	64
$M_{\mathrm{d}-64}$	98	147	176	226
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-7}$)	99	147	176	226
$M_{\mathrm{sp}-64}$ ($\xi = \ 10^{-6}$)	101	148	177	226
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-6}$)	121	166	194	252
	Time per iteration			
# processors	8	16	32	64
$M_{\mathrm{d}-64}$	0.96	0.53	0.33	0.24
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-7}$)	0.60	0.36	0.24	0.19
$M_{\mathrm{sp}-64}$ ($\xi = \ 10^{-6}$)	0.57	0.34	0.22	0.18
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-6}$)	0.50	0.30	0.21	0.18
	Preconditioner setup time			
# processors	8	16	32	64
$M_{\mathrm{d}-64}$	208.0	89.0	30.0	15.0
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-7}$)	55.6	26.1	16.0	14.3
$M_{\mathrm{sp}-64}$ ($\xi = \ 10^{-6}$)	42.2	19.5	13.0	11.4
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-6}$)	26.3	10.8	6.1	4.8
	Max of the local Schur size			
# processors	8	16	32	64
All preconditioners	12420	9444	6420	4950
	Initialization time			
# processors	8	16	32	64
All preconditioners	223.0	50.3	36.0	13.0

Table 7.8: Detailed performance for the Fuselage problem with about one million elements and 6.5 Mdof when the number of processors is varied for the various variants of the preconditioner and for various choices of ξ. We also report the "factorization+solve" time using the parallel sparse direct solver.

	Total solution time			
# processors	8	16	32	64
Direct	376.0	283.9	171.3	98.8
$M_{\mathrm{d}-64}$	344.4	232.5	151.6	62.1
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-7}$)	214.9	143.7	84.6	55.1
$M_{\mathrm{sp}-64}$ ($\xi = 10^{-6}$)	203.4	129.2	80.6	51.6
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-6}$)	189.2	115.6	78.1	44.5
$M_{\mathrm{sp}-64}$ ($\xi = 10^{-5}$)	190.7	120.9	79.7	no cvg
	Time in the iterative loop			
# processors	8	16	32	64
$M_{\mathrm{d}-64}$	65.9	69.8	72.7	42.0
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-7}$)	41.2	61.5	47.4	34.6
$M_{\mathrm{sp}-64}$ ($\xi = 10^{-6}$)	39.5	58.7	46.4	33.6
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-6}$)	38.7	52.8	47.6	30.5
$M_{\mathrm{sp}-64}$ ($\xi = 10^{-5}$)	43.8	58.5	50.7	no cvg
	# iteration			
# processors	8	16	32	64
$M_{\mathrm{d}-64}$	94	122	168	232
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-7}$)	94	122	168	232
$M_{\mathrm{sp}-64}$ ($\xi = 10^{-6}$)	94	123	168	232
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-6}$)	104	134	177	242
$M_{\mathrm{sp}-64}$ ($\xi = 10^{-5}$)	123	158	201	no cvg
	Time per iteration			
# processors	8	16	32	64
$M_{\mathrm{d}-64}$	0.70	0.57	0.43	0.18
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-7}$)	0.44	0.50	0.28	0.15
$M_{\mathrm{sp}-64}$ ($\xi = 10^{-6}$)	0.42	0.48	0.28	0.14
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-6}$)	0.37	0.39	0.27	0.13
$M_{\mathrm{sp}-64}$ ($\xi = 10^{-5}$)	0.36	0.37	0.25	no cvg
	Preconditioner setup time			
# processors	8	16	32	64
$M_{\mathrm{d}-64}$	142.8	110.0	55.9	10.4
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-7}$)	38.0	29.5	14.3	10.8
$M_{\mathrm{sp}-64}$ ($\xi = 10^{-6}$)	28.2	17.8	11.3	8.3
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-6}$)	14.8	10.1	7.5	4.2
$M_{\mathrm{sp}-64}$ ($\xi = 10^{-5}$)	11.2	9.7	6.1	3.2
	Max of the local Schur size			
# processors	8	16	32	64
All preconditioners	11004	10212	8022	4338
	Initialization time			
# processors	8	16	32	64
All preconditioners	135.7	52.7	23.0	9.7

Table 7.9: Detailed performance for the Fuselage problem with about 0.8 million elements and 4.8 Mdof when the number of processors is varied for the various variants of the preconditioner and for various choices of ξ. We also report the "factorization+solve" time using the parallel sparse direct solver.

	Total solution time				
# processors	4	8	16	32	64
Direct	-	314.4	162.5	83.0	78.5
$M_{\mathrm{d-64}}$	411.1	199.4	107.7	59.3	38.2
$M_{\mathrm{sp-64}}$ ($\xi = 5.10^{-7}$)	334.2	133.0	79.4	51.0	37.3
$M_{\mathrm{sp-64}}$ ($\xi = 10^{-6}$)	311.6	125.1	72.9	46.0	35.6
$M_{\mathrm{sp-64}}$ ($\xi = 5.10^{-6}$)	282.8	114.3	65.0	40.7	30.6
$M_{\mathrm{sp-64}}$ ($\xi = 10^{-5}$)	282.3	114.1	67.3	39.5	29.7
	Time in the iterative loop				
# processors	4	8	16	32	64
$M_{\mathrm{d-64}}$	32.5	46.2	37.2	32.3	26.7
$M_{\mathrm{sp-64}}$ ($\xi = 5.10^{-7}$)	23.2	31.6	27.5	24.3	23.3
$M_{\mathrm{sp-64}}$ ($\xi = 10^{-6}$)	22.5	30.3	25.9	23.3	23.1
$M_{\mathrm{sp-64}}$ ($\xi = 5.10^{-6}$)	25.1	29.5	24.8	22.5	21.0
$M_{\mathrm{sp-64}}$ ($\xi = 10^{-5}$)	29.5	31.9	28.8	22.7	20.8
	# iteration				
# processors	4	8	16	32	64
$M_{\mathrm{d-64}}$	38	92	124	169	224
$M_{\mathrm{sp-64}}$ ($\xi = 5.10^{-7}$)	39	92	124	169	224
$M_{\mathrm{sp-64}}$ ($\xi = 10^{-6}$)	40	92	124	169	224
$M_{\mathrm{sp-64}}$ ($\xi = 5.10^{-6}$)	51	99	130	173	228
$M_{\mathrm{sp-64}}$ ($\xi = 10^{-5}$)	64	114	155	191	248
	Time per iteration				
# processors	4	8	16	32	64
$M_{\mathrm{d-64}}$	0.85	0.50	0.30	0.19	0.12
$M_{\mathrm{sp-64}}$ ($\xi = 5.10^{-7}$)	0.60	0.34	0.22	0.14	0.10
$M_{\mathrm{sp-64}}$ ($\xi = 10^{-6}$)	0.56	0.33	0.21	0.14	0.10
$M_{\mathrm{sp-64}}$ ($\xi = 5.10^{-6}$)	0.49	0.30	0.19	0.13	0.09
$M_{\mathrm{sp-64}}$ ($\xi = 10^{-5}$)	0.46	0.28	0.19	0.12	0.08
	Preconditioner setup time				
# processors	4	8	16	32	64
$M_{\mathrm{d-64}}$	182.0	79.1	37.9	13.8	5.3
$M_{\mathrm{sp-64}}$ ($\xi = 5.10^{-7}$)	114.3	27.4	19.2	13.4	7.8
$M_{\mathrm{sp-64}}$ ($\xi = 10^{-6}$)	92.5	20.8	14.4	9.4	6.3
$M_{\mathrm{sp-64}}$ ($\xi = 5.10^{-6}$)	61.0	10.8	7.5	4.9	3.4
$M_{\mathrm{sp-64}}$ ($\xi = 10^{-5}$)	56.2	8.2	5.8	3.5	2.7
	Iterative system unknowns				
# processors	4	8	16	32	64
All preconditioners	17568	28644	43914	62928	88863
	Max of the local Schur size				
# processors	4	8	16	32	64
All preconditioners	11766	8886	7032	4908	3468
	Initialization time				
# processors	4	8	16	32	64
All preconditioners	196.7	74.0	32.7	13.2	6.2

Table 7.10: Detailed performance for the Fuselage problem with about 0.5 million elements and 3.3 Mdof when the number of processors is varied for the various variants of the preconditioner and for various choices of ξ. We also report the "factorization+solve" time using the parallel sparse direct solver. "-" means that the result is not available because of the memory requirement.

	Total solution time			
# processors	8	16	32	64
Direct	435.1	350.0	210.7	182.5
$M_{\mathrm{d}-64}$	453.7	264.6	110.9	70.1
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-7}$)	499.4	262.2	143.6	64.4
$M_{\mathrm{sp}-64}$ ($\xi = 10^{-6}$)	433.4	212.6	124.6	59.7
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-6}$)	277.5	151.7	86.5	51.4
$M_{\mathrm{sp}-64}$ ($\xi = 10^{-5}$)	246.7	134.6	70.5	47.2
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-5}$)	214.0	122.1	63.4	46.2
	Time in the iterative loop			
# processors	8	16	32	64
$M_{\mathrm{d}-64}$	57.2	60.8	44.5	42.1
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-7}$)	54.9	50.4	39.6	37.4
$M_{\mathrm{sp}-64}$ ($\xi = 10^{-6}$)	51.9	49.8	36.7	37.7
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-6}$)	42.0	47.9	37.6	35.6
$M_{\mathrm{sp}-64}$ ($\xi = 10^{-5}$)	42.2	41.8	30.2	33.2
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-5}$)	38.5	44.3	34.1	34.4
	# iteration			
# processors	8	16	32	64
$M_{\mathrm{d}-64}$	59	79	106	156
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-7}$)	59	80	107	156
$M_{\mathrm{sp}-64}$ ($\xi = 10^{-6}$)	59	83	108	157
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-6}$)	60	87	114	162
$M_{\mathrm{sp}-64}$ ($\xi = 10^{-5}$)	63	89	116	166
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-5}$)	70	103	131	191
	Time per iteration			
# processors	8	16	32	64
$M_{\mathrm{d}-64}$	0.97	0.77	0.42	0.27
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-7}$)	0.93	0.63	0.37	0.24
$M_{\mathrm{sp}-64}$ ($\xi = 10^{-6}$)	0.88	0.60	0.34	0.24
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-6}$)	0.70	0.55	0.33	0.22
$M_{\mathrm{sp}-64}$ ($\xi = 10^{-5}$)	0.67	0.47	0.26	0.20
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-5}$)	0.55	0.43	0.26	0.18
	Preconditioner setup time			
# processors	8	16	32	64
$M_{\mathrm{d}-64}$	235.0	137.0	43.5	19.0
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-7}$)	283.0	145.0	81.2	18.0
$M_{\mathrm{sp}-64}$ ($\xi = 10^{-6}$)	220.0	96.0	65.0	13.0
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-6}$)	74.0	37.0	26.0	6.8
$M_{\mathrm{sp}-64}$ ($\xi = 10^{-5}$)	43.0	26.0	17.5	5.0
$M_{\mathrm{sp}-64}$ ($\xi = 5.10^{-5}$)	14.0	11.0	6.5	2.8
	Max of the local Schur size			
# processors	8	16	32	64
All preconditioners	13296	10953	7404	5544
	Initialization time			
# processors	8	16	32	64
All preconditioners	161.5	66.8	22.9	9.0

Table 7.11: Detailed performance for the Rouet problem with about 0.33 million elements and 1.3 Mdof when the number of processors is varied for the various variants of the preconditioner and for various choices of ξ. We also report the "factorization+solve" time using the parallel sparse direct solver.

a direct solver. The direct solver is used in the context of general symmetric matrices, where with the default parameters automatically set by the MUMPS package. In term of permutation, we have performed some experiments to compare between the nested dissection Metis routine of ordering and the Approximate Minimum Degree (AMD) ordering. The best performance was observed when we used the Metis routine. We performed runs using assembled and distributed matrix entries. We mention that neither the associated distribution or redistribution of the matrix entries, nor the time for the symbolic analysis are taken into account in our time measurements. For the direct method, we only report the minimum elapsed time for the factorization and for the backward/forward substitutions. We illustrate the corresponding computing time when increasing the number of processors, for all the tests cases. Figure 7.3 presents the three Fuselage test cases, whereas Figure 7.4 describes the Rouet one. The graphs on the left of these figures give the number of iterations required by each variant, whereas the right graphs summarize the overall computing time and compare them with the time required by the direct solver. Compared with the direct method, our hybrid approach gives always the fastest scheme. Over the course of a long simulation, where each step requires the solution of a linear system, our approach represents a significant saving in computing resources; this observation is especially valid for the attractive sparse variant.

We now investigate the analysis in term of memory requirement. We depict in Table 7.12 the maximal peak of memory required on the subdomains to compute the factorization and either the dense preconditioner for the $hybrid - M_{\mathrm{d}-64}$ method or the sparse preconditioner with $\xi = 5.10^{-6}$ for the $hybrid - M_{\mathrm{sp}-64}$ method. We report also the average memory required by the direct method.

For each test case, we report in each row of Table 7.12 the amount of memory storage required (in MB) for the different decomposition described in this subsection. This amount is huge for small number of subdomains. This is due to the fact that the size of the local Schur complements is extremely large. Furthermore the large number of unknowns associated with the interior of each subdomain leads to local factorizations that are memory consuming. A feature of the sparse variants is that they reduce the preconditioner memory usage.

# processors		4	8	16	32	64
Rouet $1.3{\cdot}10^6 dof$	*Direct*	-	5368	2978	1841	980
	$Hybrid - M_{\mathrm{d}-64}$	-	5255	3206	1414	739
	$Hybrid - M_{\mathrm{sp}-64}$	-	3996	2400	1068	560
Fuselage $3.3{\cdot}10^6 dof$	*Direct*	5024	3567	2167	990	669
	$Hybrid - M_{\mathrm{d}-64}$	6210	3142	1714	846	399
	$Hybrid - M_{\mathrm{sp}-64}$	5167	2556	1355	678	320
Fuselage $4.8{\cdot}10^6 dof$	*Direct*	9450	6757	3222	1707	1030
	$Hybrid - M_{\mathrm{d}-64}$	8886	4914	2994	1672	623
	$Hybrid - M_{\mathrm{sp}-64}$	7470	4002	2224	1212	495
Fuselage $6.5{\cdot}10^6 dof$	*Direct*	-	8379	5327	2148	1503
	$Hybrid - M_{\mathrm{d}-64}$	-	6605	3289	1652	831
	$Hybrid - M_{\mathrm{sp}-64}$	-	5432	2625	1352	660

Table 7.12: Comparison of the maximal local peak of the data storage (MB) needed by the hybrid and the direct method. "-" means that the result is not available because of the memory requirement.

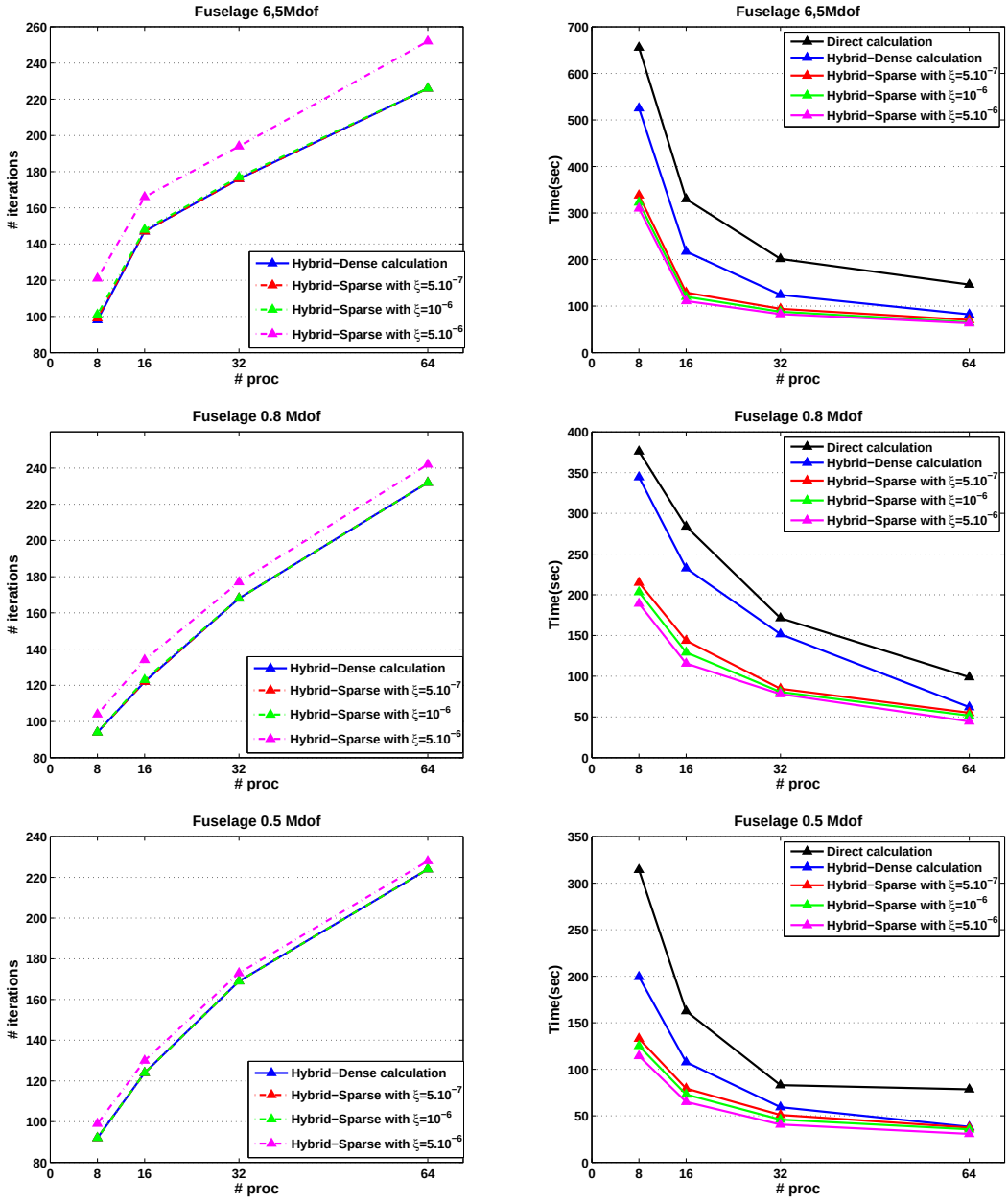


Figure 7.3: Parallel performance for the Fuselage test cases, when increasing the number of processors. Left graphs display the numerical behaviour (number of iterations), whereas the right graphs display a comparison of the computing time between the hybrid solver and the direct solver. Moreover, the results for both preconditioners and for different values of ξ are reported.

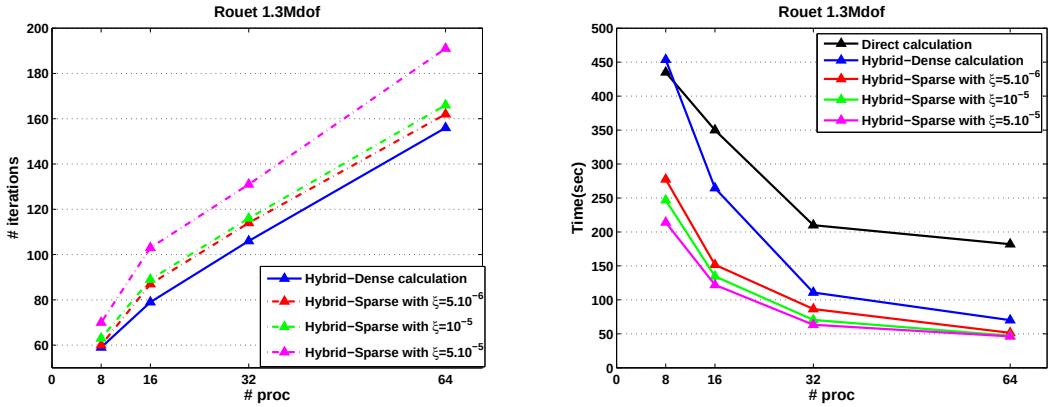


Figure 7.4: Parallel performance for the Rouet (1.3 Mdof) test case, when increasing the number of processors. Left graphs display the numerical behaviour (number of iterations), whereas the right graphs display a comparison of the computing time between the hybrid solver and the direct solver. Moreover, the results for both preconditioners and for different values of ξ are reported.

7.5 Symmetric positive definite linear systems in structural mechanics

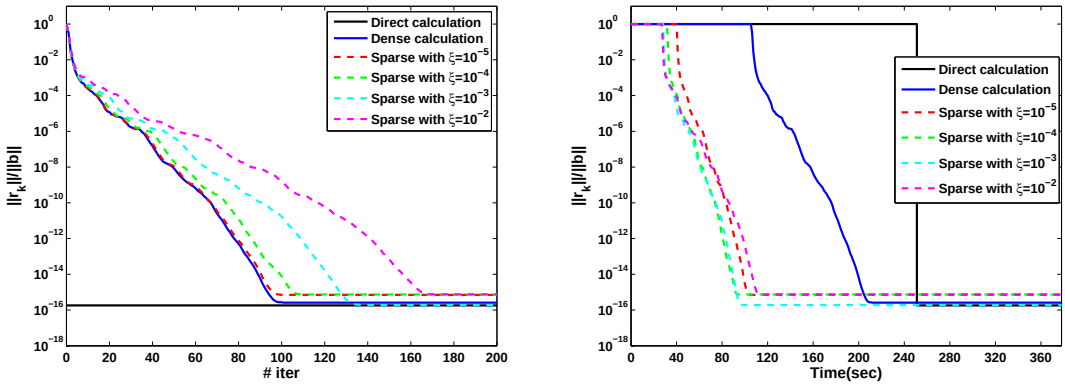
7.5.1 Numerical behaviour

As discussed in the previous chapters, we study in this section the numerical behaviour of the preconditioners. For that purpose, we compare the numerical performance of the sparsified preconditioner $M_{\mathrm{sp}-64}$ and compare it to the classical $M_{\mathrm{d}-64}$. We also perform a comparison with the mixed arithmetic preconditioner $M_{\mathrm{d}-\mathrm{mix}}$. To be exhaustive, we also consider a direct solver. We note that for these problems, the discretization gives rise to linear systems that are symmetric positive definete. Therefore we use the conjugate gradient Krylov solver in the iterative phase. The performance and robustness of the preconditioners are evaluated for the PAMC50 test problem.

7.5.1.1 Influence of the sparsification threshold

In order to study the effect of the sparse preconditioner on the convergence rate we display in Figure 7.5 the convergence history for various choices of the dropping parameter ξ involved in the definition of $M_{\mathrm{sp}-64}$ in Equation (3.6). We also compare them to the convergence history of the dense $M_{\mathrm{d}-64}$ and to a direct solution method. On the left-hand side we display the convergence history as a function of the iterations. On the right-hand side, the convergence is given as a function of the computing time. The results presented here are for a mesh with 125,000 finite elements (0.8 million dof) on the PAMC50 test case mapped onto 32 processors.

These results show again the attractive features of the sparse variant. They illustrate the main advantage of the sparse preconditioners that is, their very low costs both in memory and in computing time compared to the dense preconditioner or to the direct method. Again, the preconditioning quality is not significantly degraded, the sparse approach behaves very closely to the dense one for suited choices of ξ (for example 10^{-5} or 10^{-4}). We report in Table 7.13, the memory space and the computing time required to build the preconditioner on each processor. The maximal subdomain interface in this test case is 11119 unknowns. The results demonstrate the effectiveness of the sparsified preconditioners. When considering both memory and computational aspects of the sparse approach we believe that it exhibits many advantages, in particular when dealing with large subdomain interfaces.



(a) PAMC50 0.8 Mdof (history v.s. iterations). (b) PAMC50 0.8 Mdof (history v.s. time).

Figure 7.5: Convergence history for PAMC50 problem mapped onto 32 processors, of the direct, the Hybrid-dense ($M_{\mathrm{d}-64}$) and the Hybrid-sparse ($M_{\mathrm{sp}-64}$) solvers for various sparsification dropping thresholds (Left: scaled residual versus iterations, Right: scaled residual versus time).

ξ	0	10^{-5}	10^{-4}	10^{-3}	10^{-2}
PAMC50 problem with 0.8 Mdof					
Memory	943_{MB}	160_{MB}	38_{MB}	11_{MB}	3_{MB}
Kept percentage	100%	17.0%	4.1%	1.1%	0.3%
Preconditioner setup	83.3	17.7	9.5	5.9	5.0

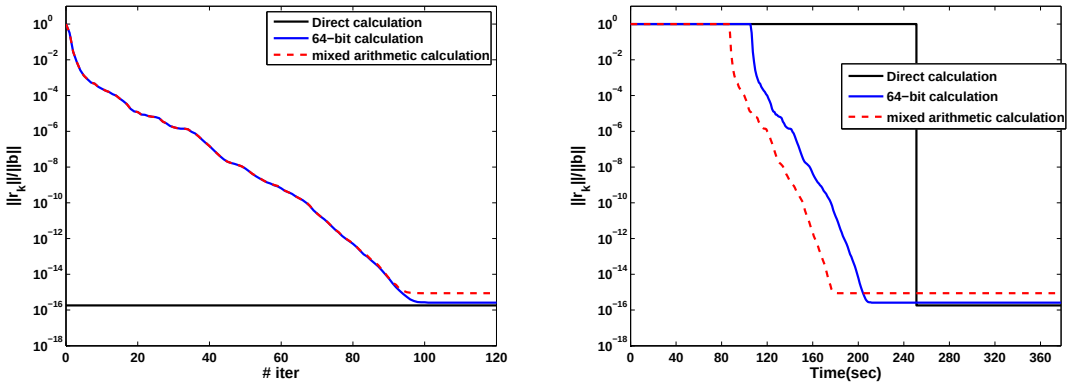
Table 7.13: Preconditioner computing time(*sec*) and amount of memory(MB) in $M_{\mathrm{sp}-64}$ v.s. $M_{\mathrm{d}-64}$ for various choices of the dropping parameter.

7.5.1.2 Influence of the mixed arithmetic

We focus in this subsection on the numerical behaviour of the mixed arithmetic approach [50] and compare it with the full 64-bit and with a direct solution method. In this respect, we consider the same example as in the previous subsection. We plot the convergence history for the PAMC50 test problem when it is decomposed into 32 subdomains. As previously, the performance and the robustness of the mixed arithmetic preconditioner are evaluated. On the left graph of Figure 7.6, the convergence history is a function of the iterations and on the right one the convergence history is a function of time. Again, it can be observed that for this type of $3D$ problems, the mixed precision algorithm behaves very closely to the 64-bit algorithm. It does not delay the convergence. As expected the two preconditioners reach the same accuracy as the direct method. That is, at the level of 64-bit arithmetic. When looking at the right graph, one observes that the saving in computing time is not as significative as it was in Section 5.3.2. We should mention that the computing platform that we use here does not allow higher 32-bit processing speed compared with 64-bit. The main advantage of this approach is that it gives rise to similar behaviour as the 64-bit algorithm with low cost in both memory and marginately less computing time.

7.5.2 Parallel performance experiments

For the sake of completeness, we would like to illustrate the performance of these implementations for both the numerical and the computing time point of view. We require that the normwise backward error becomes smaller than 10^{-8}. We perform experiments for a fixed mesh size, when decomposed into different number of subdomains. In the first subsection below, we analyze the



(a) PAMC50 0.8 Mdof (history v.s. iterations). (b) PAMC50 0.8 Mdof (history v.s. time).

Figure 7.6: Convergence history for PAMC50 problem mapped onto 32 processors, of the direct, the Hybrid-64 bit ($M_{\mathrm{d}-64}$) and the Hybrid-32 bit ($M_{\mathrm{d}-\mathrm{mix}}$) solvers (Left: scaled residual versus iterations, Right: scaled residual versus time).

numerical performance of the three preconditioners, whereas the next subsection is devoted to the study of their parallel features and efficiency.

7.5.2.1 Numerical scalability

We now illustrate the numerical behaviour of the conjugate gradient Krylov solver, when the number of subdomains increases. The preconditioner tested are the dense 64-bit additive Schwarz preconditioner $M_{\mathrm{d}-64}$, the sparse alternative $M_{\mathrm{sp}-64}$ and the mixed arithmetic variant $M_{\mathrm{d}-\mathrm{mix}}$. These preconditioners were presented in Section 3.2, Section 3.3 and Section 3.4. The numerical experiments are performed on the PAMC50 (0.8 Mdof) test problem, when decomposed into 16, 32, 64 and 96 subdomains. Another comparison is presented on the PAMC80 (3.2 Mdof) mapped onto 192 processors. We note that this latter simulation cannot be performed using a direct method neither on 96 nor on 192 processors. The main drawback is that it typically requires more than 400 GBytes on 96 processors and 507 GBytes on 192 processors. This amount of memory is not available on our test platform that has only 384 GBytes on 192 processors.

Table 7.14 presents the number of iterations. The results obtained with different choices of the dropping parameter ξ of $M_{\mathrm{sp}-64}$ are also given. In addition, we report in Table 7.14, the percentage of kept entries for the sparsification strategies. We see that, except in one case, the choice of the preconditioner does not really influence the convergence of the iterative scheme. The only degradation is observed with the sparse preconditioner $M_{\mathrm{sp}-64}$ for the very small value $\xi = 10^{-2}$. When multiplying the number of subdomains by 6, the number of iterations is multiplied by less than 2 for all variants. This trend is similar to the one observed on academic examples in Chapter 5.

On a larger problem, we present in Table 7.15 the number of iterations required on the PAMC80 problem mapped onto 96 and 192 processors. The preconditioner performs as well as in the previous simulations for this bigger test example. Using 96 processors, the convergence is reached in 76 iterations for $M_{\mathrm{d}-64}$, while 73 iterations were required by PAMC50 on the same number of processors. Also PAMC80 requires 89 iterations with $M_{\mathrm{sp}-64}$ using $\xi = 10^{-4}$, while PAMC50 needs 76. This is a promising behaviour, that shows that when increasing the overall size of the problem four times (from $0.8 \cdot 10^6$ dofs to $3.2 \cdot 10^6$ dofs), for the same decomposition (96 subdomains) only 3 extra iterations are needed by $M_{\mathrm{d}-64}$, and 13 extra iterations are required by $M_{\mathrm{sp}-64}$ using $\xi = 10^{-4}$.

7.5.2.2 Parallel performance scalability

We devote this subsection to the discussion of the parallel performance and analysis of the preconditioners. We also report a comparison of all the preconditioners with a direct method. Similarly to the indefinite case, we report in Table 7.18, the detailed computing time for the PAMC50 test problem.

For a fixed mesh size, we vary the number of subdomains from 16, 32, 64 up to 96. For each of this partition, we report the maximal size of the local subdomain interface, and the computing time needed by each of the three main phases of our hybrid method in addition to the required time per iteration. We would like to underline the fact that, whatever the preconditioner is, a very significant decrease in the computing time can be observed when we increase the number of processors. We may remark that the behaviour of the preconditioners is similar to what was described above as well as in Chapter 5 and Chapter 6. The sparse preconditioners perform more than twice faster than $M_{\mathrm{d}-64}$ whereas the mixed precision preconditioner $M_{\mathrm{d}-\mathrm{mix}}$ is still reducing the overall computing time even on this platform where 32 and 64-bit calculation are performed at the same speed. For the sparse techniques, we can remark that, here the choice of $\xi = 10^{-4}$ gives us the best parallel performance and thus by looking in Table 7.14, we can conclude that for this type of problem, it is sufficient to retain between 3% to 10% of the local Schur complement entries.

In order to be exhaustive, we also report in Table 7.16 the elapsed time to factorize the $\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}$ matrix with or without Schur computation. Finally in Table 7.17 we display the elapsed time to perform the matrix-vector product using both explicit or implicit approaches described in Subsection 7.4.2.2. The results of Table 7.16 indicate that when the size of the interface (Schur complement) is comparable to the number of interior unknowns, the partial factorization that builds the Schur complement (explicit method) becomes much more expensive than the factorization of the local problem. By looking in Table 7.17, it can be seen that, in that context the implicit approach slightly outperforms the explicit one. This behaviour is clearly observed on the PAMC50 where the ratio $\frac{interior}{interface}$ is 2.7 on 16 subdomains and 0.8 on 96 subdomains. That is, the number of interior unknowns is very closed to the number of unknowns on the interface. This ratio is clearly much smaller than for the Fuselage problem considered in Section 7.4.

# processors		16	32	64	96
$125{\cdot}10^3 elements$ $0.8{\cdot}10^6 dof$	$M_{\mathrm{d}-64}$	40	50	69	73
	$M_{\mathrm{d}-\mathrm{mix}}$	40	50	66	74
	$M_{\mathrm{sp}-64}$ $(\xi = 10^{-4})$	47 (3%)	55 (4%)	70 (6%)	76 (8%)
	$M_{\mathrm{sp}-64}$ $(\xi = 10^{-3})$	58 (0.7%)	65 (1.0%)	83 (1.6%)	87 (2.0%)
	$M_{\mathrm{sp}-64}$ $(\xi = 10^{-2})$	73 (0.2%)	86 (0.3%)	103 (0.5%)	113 (0.6%)

Table 7.14: Number of preconditioned conjugate gradient iterations and percentage of kept entries in the sparse preconditioner for the PAMC50 problem with about 125,000 elements and 0.8 Mdof The number of processors is varied for the various variants of the preconditioner using various choices of ξ .

$PAMC80$ $3.2{\cdot}10^6 dof$	$M_{\mathrm{d}-64}$	$M_{\mathrm{d}-\mathrm{mix}}$	$M_{\mathrm{sp}-64}$ with $\xi = 10^{-4}$	$M_{\mathrm{sp}-64}$ with $\xi = 10^{-3}$	$M_{\mathrm{sp}-64}$ with $\xi = 10^{-2}$
96 *processors*	76	-	89	-	-
192 *processors*	96	103	106	126	158

Table 7.15: Number of preconditioned conjugate gradient iterations for the PAMC80 problem with about 512,000 elements and 3.2 Mdof when the number of processors is varied for the various variants of the preconditioner and for various choices of ξ . "-" means that the run is not available.

# processors		16	32	64	96
$PAMC50\ 0.8{\cdot}10^6 dof$	Interface size	14751	11119	8084	6239
	Interior size	40012	19009	8666	5365
	explicit ($\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}$ + Schur)	90.1	21.5	7.1	4.0
	implicit ($\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}$)	15.0	4.1	1.5	0.8

Table 7.16: Parallel elapsed time (sec) for the factorization of $\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}$ with or without Schur complement.

# processors		16	32	64	96
$PAMC50\ 0.8{\cdot}10^6 dof$	*explicit*	0.65	0.32	0.18	0.10
	implicit	0.52	0.29	0.13	0.09

Table 7.17: Parallel elapsed time (sec) for one matrix-vector product step.

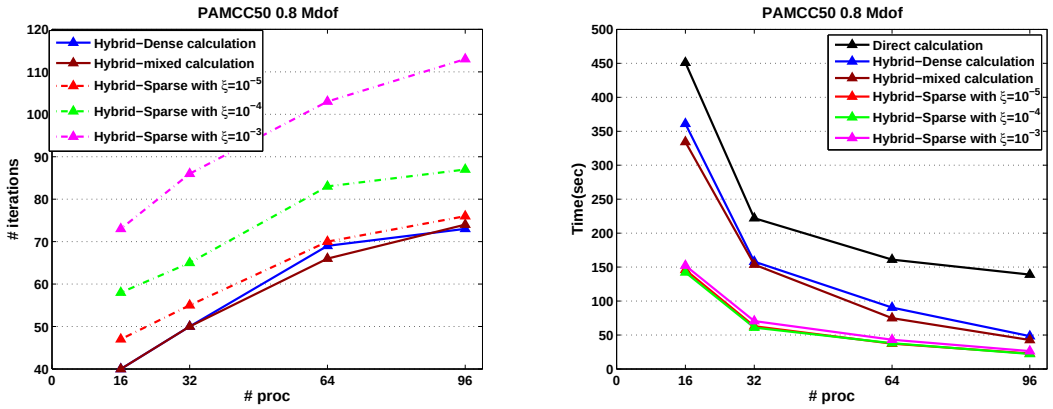


Figure 7.7: Parallel performance for the PAMC50 (0.8 Mdof) test case, when increasing the number of processors.

	Total solution time			
# processors	16	32	64	96
Direct	451.0	222.5	161.1	139.4
$M_{\mathrm{d-64}}$	361.0	158.0	90.4	48.3
$M_{\mathrm{d-mix}}$	334.2	153.7	74.8	42.7
$M_{\mathrm{sp-64}}$ ($\xi = 10^{-4}$)	146.0	62.9	37.2	22.9
$M_{\mathrm{sp-64}}$ ($\xi = 10^{-3}$)	142.2	60.9	37.9	22.2
$M_{\mathrm{sp-64}}$ ($\xi = 10^{-2}$)	152.2	70.5	43.1	26.3
	Time in the iterative loop			
# processors	16	32	64	96
$M_{\mathrm{d-64}}$	70.0	52.7	49.6	26.2
$M_{\mathrm{d-mix}}$	43.2	48.4	34.0	20.6
$M_{\mathrm{sp-64}}$ ($\xi = 10^{-4}$)	38.7	31.5	26.7	16.3
$M_{\mathrm{sp-64}}$ ($\xi = 10^{-3}$)	43.0	33.0	28.4	16.7
$M_{\mathrm{sp-64}}$ ($\xi = 10^{-2}$)	54.5	43.5	34.0	20.9
	# iteration			
# processors	16	32	64	96
$M_{\mathrm{d-64}}$	40	50	69	73
$M_{\mathrm{d-mix}}$	40	50	66	74
$M_{\mathrm{sp-64}}$ ($\xi = 10^{-4}$)	47	55	70	76
$M_{\mathrm{sp-64}}$ ($\xi = 10^{-3}$)	58	65	83	87
$M_{\mathrm{sp-64}}$ ($\xi = 10^{-2}$)	73	86	103	113
	Time per iteration			
# processors	16	32	64	96
$M_{\mathrm{d-64}}$	1.75	1.05	0.72	0.36
$M_{\mathrm{d-mix}}$	1.08	0.97	0.52	0.28
$M_{\mathrm{sp-64}}$ ($\xi = 10^{-4}$)	0.82	0.57	0.38	0.21
$M_{\mathrm{sp-64}}$ ($\xi = 10^{-3}$)	0.74	0.51	0.34	0.19
$M_{\mathrm{sp-64}}$ ($\xi = 10^{-2}$)	0.75	0.51	0.33	0.18
	Preconditioner setup time			
# processors	16	32	64	96
$M_{\mathrm{d-64}}$	200.0	83.3	33.6	18.0
$M_{\mathrm{d-mix}}$	171.0	64.3	30.0	14.6
$M_{\mathrm{sp-64}}$ ($\xi = 10^{-4}$)	16.3	9.4	3.3	2.5
$M_{\mathrm{sp-64}}$ ($\xi = 10^{-3}$)	8.2	5.9	2.3	1.4
$M_{\mathrm{sp-64}}$ ($\xi = 10^{-2}$)	6.7	5.0	1.9	1.2
	Max of the local Schur size			
# processors	16	32	64	96
All preconditioners	14751	11119	8084	6239
	Initialization time			
# processors	16	32	64	96
All preconditioners	91.0	22.0	7.2	4.1

Table 7.18: Detailed performance for the PAMC50 problem with about 125,000 elements and 0.8 Mdof when the number of processors is varied for the various variants of the preconditioner and for various choices of ξ. We also report the "factorization+solve" time using the parallel sparse direct solver.

In order to summarize and compare with the direct method, we plot in Figure 7.7 the performance of the solution techniques on the PAMC50 test case. On the left plot, we display the number of iterations when the number of subdomains is varied. On the right, we depict the overall solution time. In that latter plot, we also report on the parallel performance of a sparse direct solution (again we do not take into account the symbolic analysis time and the distribution time of the matrix entries). On that example, we can see that the hybrid approaches outperform the sparse direct solution technique.

We now look at the memory requirement, for that, we depict in Table 7.19 the maximal peak of memory required on one processor for the hybrid method with either the dense preconditioner ($hybrid - M_{\mathrm{d}-64}$), the mixed preconditioner ($hybrid - M_{\mathrm{d}-\mathrm{mix}}$), and the sparse preconditioner ($hybrid - M_{\mathrm{sp}-64}$) with $\xi = 10^{-3}$. We report also the average of the memory required by the direct method. We report the size in megabytes of the memory storage required for the different decompositions described in this subsection. This amount is very large for small number of subdomains, especially due to the fact that the size of the local Schur complements is large. A feature of the sparse variants is that they reduce the preconditioner memory usage, which has also a considerable effect in the execution time as it reduces the number of floating-point operations required by the factorization.

# processors		16	32	64	96
$PAMC50$ of $0.8{\cdot}10^6 dof$	$Direct$	4175	2265	1507	1027
	$Hybrid - M_{\mathrm{d}-64}$	5318	2688	1282	802
	$Hybrid - M_{\mathrm{d}-\mathrm{mix}}$	4280	2114	986	623
	$Hybrid - M_{\mathrm{sp}-64}$	3422	1630	733	474

Table 7.19: Comparison of the maximal local peak of the data storage needed by the direct and the hybrid method for the different studied preconditioners.

7.6 Exploiting *2-levels* of parallelism

7.6.1 Motivations

Classical parallel implementations of domain decomposition techniques assign one subdomain per processor. Such an approach has two main drawbacks:

1. For many applications, increasing the number of subdomains often leads to increasing the number of iterations to converge. If no efficient numerical mechanism, such as coarse space correction for elliptic problems, is available the solution of very large problems might become ineffective.

2. It implies that the memory required to handle each subdomain is available on each processor. On SMP (Symmetric Multi-Processors) node this constraint can be relaxed as we might only use a subset of the available processors to allow each processor to access more memory. Although such a solution enables an optimal use of the memory some processors are "wasted".

One possible alternative to cure those weaknesses is to consider parallel implementations that exploit *2-levels* of parallelism [49]. Those implementations consist in using parallel numerical linear algebra kernels to handle each subdomain.

In the next sections we study the numerical benefits and parallel performance advantages of the *2-level parallel* approach in the context of structural mechanical simulations. We draw the attention of the reader on the fact that in those sections the number of processors and the number of subdomains are most of the time different. A *2-level parallel* implementation will be effective for hybrid solver if the three main phases of these numerical techniques can be efficiently performed

in parallel. Due to some features of the version of the parallel sparse direct solver MUMPS the first phase that consists in factorizing the local internal problems and computing the local Schur complement was still perform sequential. This limitation should disappear in the next release of the solver that would enable us to better assess the advantages and weaknesses of the *2-level parallel* scheme.

7.6.2 Numerical benefits

The numerical attractive feature of the *2-level parallel* approach is that increasing the number of processors to speedup the solution of large linear systems does not imply increasing the number of iterations to converge as it is often the case with the *1-level parallel* approach.

We report in Table 7.20 both the number of iterations and the parallel computing time spent in the iterative loop, for the problems depicted in Section 7.4. For each problem, we choose a fixed number of processors and vary the number of subdomains that are allocated to different number of processors.

In this table it can be seen that decreasing the number of subdomains reduces the number of iterations. The parallel implementations of the numerical kernels involved in the iterative loop is efficient enough to speedup the solution time. On the largest Fuselage example, when we have 32 processors, standard (*1-level parallel*) implementation partitions the mesh into 32 subdomains requiring 176 iterations to convergence and consuming 58.1 seconds. With the *2-level parallel* implementation, either 16 or 8 subdomains can be used. The 16 subdomain partition requires 147 iterations performed in 44.8 seconds and the 8 subdomain calculation needs 98 iterations performed in 32.5 seconds. This example illustrates the advantage of the *2-level parallel* implementation from a numerical viewpoint. We should mention that using the *2-levels* of parallelism leads also to decrease the computing time needed to setup the preconditioners. This will be explained in details in the next section. Whereas the performance of the initialization phase is momently omitted due to the version of the direct solver we used.

7.6.3 Parallel performance benefits

When running large simulations that need all the memory available on the nodes of an SMP-machine, standard parallel codes (*1-level parallel*) are enforced to use only one processor per node, thus leaving the remaining processors idle. In this context, the goal of the *2-level parallel* method, is to exploit the computing facilities of the remaining processors and allows them to contribute to the computation.

We report in Table 7.21, the performance results of the *2-level parallel* method for the Fuselage with 6.5 million degrees of freedom. This is the analogous to Table 7.8 of Subsection 7.4.2.2. We use the *2-level parallel* algorithm only for the simulations that left idle processors when the standard (*1-level parallel*) algorithm was run due to memory constraints. In that case, the 8 or 16 subdomain decompositions require respectively 7 GBytes and 5 GBytes of memory; so that the *1-level parallel* implementation can only exploits one of the four SMP processors. That means that the 8 subdomain simulation using the *1-level parallel* approach requires the use of 8 SMP nodes, where only one processor per node is used; its leaves 24 idle processors. Even worse, the 16 subdomain simulation requires 16 SMP nodes where still only one processors per node is used. It leaves 48 idle processors. In such a context the benefit of the *2-level parallel* approach is clear. The parallel performance of the $M_{\mathrm{d}-64}$ and $M_{\mathrm{sp}-64}$ are reported in this table and similar results are given for the other test problems (Table 7.22 for the Fuselage with 3.3 millions degrees of freedom and Table 7.23 for the Rouet with 1.3 million degrees of freedom)

To study the parallel behaviour of the *2-level parallel* implementation we discuss the efficiency of the three main steps of the algorithm.

- We recall, that these preliminary experiments were performed with a version of the sparse direct solver that does not enable us to perform efficiently the factorization of the local

# total processors	Algo	# subdomains	# processors/ subdomain	# iter	iterative loop time
Fuselage 1 million elements with 6.5 10^6 dof					
16 processors	*1-level parallel*	16	1	147	77.9
	2-level parallel	8	2	98	51.4
32 processors	*1-level parallel*	32	1	176	58.1
	2-level parallel	16	2	147	44.8
	2-level parallel	8	4	98	32.5
64 processors	*1-level parallel*	64	1	226	54.2
	2-level parallel	32	2	176	40.1
	2-level parallel	16	4	147	31.3
	2-level parallel	8	8	98	27.4
Fuselage 0.5 million elements with 3.3 10^6 dof					
8 processors	*1-level parallel*	8	1	92	46.2
	2-level parallel	4	2	38	18.6
16 processors	*1-level parallel*	16	1	124	37.2
	2-level parallel	8	2	92	25.9
	2-level parallel	4	4	38	10.1
32 processors	*1-level parallel*	32	1	169	32.3
	2-level parallel	16	2	124	22.1
	2-level parallel	8	4	92	14.3
	2-level parallel	4	8	38	11.8
Rouet 0.33 million elements with 1.3 10^6 dof					
16 processors	*1-level parallel*	16	1	79	60.8
	2-level parallel	8	2	59	34.1
32 processors	*1-level parallel*	32	1	106	44.5
	2-level parallel	16	2	79	38.6
	2-level parallel	8	4	59	21.1
64 processors	*1-level parallel*	64	1	156	42.1
	2-level parallel	32	2	106	22.4
	2-level parallel	16	4	79	26.2
	2-level parallel	8	8	59	25.5

Table 7.20: Numerical performance and advantage of the *2-level parallel* method compared to the standard *1-level parallel* method for the Fuselage and Rouet problems and for different decomposition proposed.

problem and the calculation of the local Schur complement efficiently in parallel. Consequently we preferred to keep this calculation sequential and do not discuss *2-level parallel* implementation of this step.

- Preconditioner setup *phase2*: This phase includes two steps, the assembling the local Schur complement, and the factorization of either $M_{\mathrm{d}-64}$ or $M_{\mathrm{sp}-64}$ depending on the selected variant. The factorization of $M_{\mathrm{d}-64}$ is performed using Scalapack, while $M_{\mathrm{sp}-64}$ is factorize using a parallel instance of MUMPS.

  The results reported in Tables 7.21- 7.23 highlight the advantage of the *2-level parallel* algorithm. For the dense preconditioner $M_{\mathrm{d}-64}$, we can observe that even if we not take advantage of all the working nodes and use only 2 of the 4 available processors to perform the preconditioner setup phase, the benefit is considerable. The computing time is divided by around 1.8 for all test cases. For the sparse preconditioner, the gain is also important, it varies between 1.5 to 2.

  The objective of the *2-level parallel* method is to take advantage of all the available resources to complete the simulation. That is, take advantage of all the processors of the nodes. We emphasize that the discussion in this section should pertain to the use of the 4 processors available on each node, which constitutes the main objective of our *2-level parallel* investigation. The first observation highlights the success of the *2-level parallel* algorithm on the achieved performance. For the dense $M_{\mathrm{d}-64}$ preconditioner, the calculation is performed around 3 times faster thanks to the efficiency of the parallel dense linear algebra kernels of ScaLAPACK. For the sparse preconditioner, the speedups vary between 1.5 to 3. This speedup is pronounced for small values of ξ ($\xi = 5.10^{-7}$ for the Fuselage tests and $\xi = 5.10^{-6}$ for the Rouet test) and for large interface sizes, whereas it is rather moderate for very large values of ξ, and for small interface sizes.

  Finally, it can be noticed that, the best execution times are obtained using the *2-level parallel* method for all test problems. The *2-level parallel* method is needed to attain the best parallel performance.

- The *phase3* of the method is the iterative loop, that mainly involves 3 numerical kernels that are: the matrix-vector product implemented using PBLAS routines; the preconditioner application that relies either on SCALAPACK kernels for $M_{\mathrm{d}-64}$ or MUMPS for $M_{\mathrm{sp}-64}$ and a global reduction for the dot-product calculation. The results reported in these tables, show similar speedups as the ones observed for *phase2* (preconditioner setup). For the dense preconditioner the execution of the iterative loop is 2 to 3 times faster than for the *1-level parallel* algorithm. Also, for the sparse preconditioner, the convergence is achieved 1.7 to 3 times faster.

To visualize the time saving enabled by the *2-level parallel* implementation we display in Figure 7.8 the global computing time for the different decompositions. For the sake of readability, we only display the results for the dense preconditioner. Those curves illustrate the benefit of the *2-level parallel* implementation that enables us to get much better computing throughput out of the SMP nodes.

7.7 Concluding remarks

In this chapter, we have investigated the numerical behaviour of our preconditioner for the solution of linear systems arising in three dimensional structural mechanics problems representative of difficulties encountered in this application area. In order to avoid the possible singularities related to the splitting of the Lagrange multiplier equations we propose a first solution that can surely be improved. In particular, other partitioning strategies would deserved to be studied and investigated. Some work in that direction would deserve to be undertaken prior the possible integration

# subdomains or SMP-nodes	**8**			**16**			**32**		
# processors per subdomain	*1-level parallel*	*2-levels parallel*		*1-level parallel*	*2-levels parallel*		*1-level parallel*	*2-levels parallel*	
	1	2	4	1	2	4	1	2	4
	Total solution time								
$M_{\mathrm{d}-64}$	525.1	399.1	326.4	217.2	147.8	112.0	124.1	97.2	71.7
$\xi = 5.10^{-7}$	338.0	291.4	265.5	129.0	103.8	90.7	94.2	83.8	65.3
$\xi = 10^{-6}$	322.8	279.4	260.3	120.1	95.3	84.0	87.9	79.7	62.1
$\xi = 5.10^{-6}$	309.8	270.4	251.1	110.9	87.8	79.7	82.8	71.6	56.0
	Time in the iterative loop								
$M_{\mathrm{d}-64}$	94.1	51.5	32.5	77.9	44.8	31.3	58.1	40.8	22.7
$\xi = 5.10^{-7}$	59.4	36.8	20.6	52.6	32.9	24.8	42.2	32.0	18.8
$\xi = 10^{-6}$	57.6	34.3	19.9	50.3	29.6	21.6	38.9	31.2	18.2
$\xi = 5.10^{-6}$	60.5	35.8	20.1	49.8	29.5	23.2	40.7	28.7	16.1
	# iterations								
$M_{\mathrm{d}-64}$	98			147			176		
$\xi = 5.10^{-7}$	99			147			176		
$\xi = 10^{-6}$	101			148			177		
$\xi = 5.10^{-6}$	121			166			194		
	Time per iteration								
$M_{\mathrm{d}-64}$	0.96	0.53	0.33	0.53	0.30	0.21	0.33	0.23	0.13
$\xi = 5.10^{-7}$	0.60	0.37	0.21	0.36	0.22	0.17	0.24	0.18	0.11
$\xi = 10^{-6}$	0.57	0.34	0.20	0.34	0.20	0.15	0.22	0.18	0.10
$\xi = 5.10^{-6}$	0.50	0.30	0.17	0.30	0.18	0.14	0.21	0.15	0.08
	Preconditioner setup time								
$M_{\mathrm{d}-64}$	208.0	124.6	70.8	89.0	52.7	30.4	30.0	20.4	13.0
$\xi = 5.10^{-7}$	55.6	31.6	21.9	26.1	20.6	15.5	16.0	15.7	10.5
$\xi = 10^{-6}$	42.2	22.1	17.4	19.5	15.4	12.1	13.0	12.6	7.9
$\xi = 5.10^{-6}$	26.3	11.6	8.0	10.8	8.0	6.1	6.1	6.9	3.9
	Iterative system unknowns								
All preconditioners	40200			61251			87294		
	Max of the local Schur size								
All preconditioners	12420			9444			6420		
	Initialization time								
All preconditioners	223.0			50.3			36.0		

Table 7.21: Detailed parallel performance of the *2-level parallel* method for the Fuselage problem with about one million elements and 6.5 Mdof when the number of subdomains is varied, for the various variants of the preconditioner and for various choices of ξ.

# subdomains or SMP-nodes	**4**			**8**			**16**		
# processors per subdomain	*1-level parallel*	*2-levels parallel*		*1-level parallel*	*2-levels parallel*		*1-level parallel*	*2-levels parallel*	
	1	2	4	1	2	4	1	2	4
	Total solution time								
$M_{\mathrm{d}-64}$	411.1	315.3	262.7	199.4	146.0	114.7	107.7	79.0	61.1
$\xi = 5.10^{-7}$	334.2	262.5	243.5	133.0	111.3	100.0	79.4	63.2	55.9
$\xi = 10^{-6}$	311.6	247.5	235.0	125.1	105.5	95.0	72.9	59.0	53.0
$\xi = 5.10^{-6}$	282.8	225.3	214.1	114.3	98.1	88.6	65.0	62.9	52.9
$\xi = 10^{-5}$	282.3	223.5	212.9	114.1	98.4	88.1	67.3	53.6	46.2
	Time in the iterative loop								
$M_{\mathrm{d}-64}$	32.5	18.6	10.1	46.2	25.9	14.3	37.2	22.1	13.4
$\xi = 5.10^{-7}$	23.2	15.2	9.0	31.6	18.9	11.9	27.5	17.9	13.0
$\xi = 10^{-6}$	22.5	14.0	9.4	30.3	18.1	10.7	25.9	16.2	12.2
$\xi = 5.10^{-6}$	25.1	14.4	8.1	29.5	16.7	9.2	24.8	24.7	16.2
$\xi = 10^{-5}$	29.5	16.5	9.3	31.9	18.5	10.0	28.8	16.6	10.4
	# iterations								
$M_{\mathrm{d}-64}$	38			92			124		
$\xi = 5.10^{-7}$	39			92			124		
$\xi = 10^{-6}$	40			92			124		
$\xi = 5.10^{-6}$	51			99			130		
$\xi = 10^{-5}$	64			114			155		
	Time per iteration								
$M_{\mathrm{d}-64}$	0.85	0.49	0.27	0.50	0.28	0.15	0.30	0.18	0.11
$\xi = 5.10^{-7}$	0.60	0.39	0.23	0.34	0.20	0.13	0.22	0.14	0.10
$\xi = 10^{-6}$	0.56	0.35	0.23	0.33	0.20	0.12	0.21	0.13	0.10
$\xi = 5.10^{-6}$	0.49	0.28	0.16	0.30	0.17	0.09	0.19	0.19	0.12
$\xi = 10^{-5}$	0.46	0.26	0.15	0.28	0.16	0.09	0.19	0.11	0.07
	Preconditioner setup time								
$M_{\mathrm{d}-64}$	182.0	100.0	55.8	79.1	46.0	26.4	37.9	24.3	15.0
$\xi = 5.10^{-7}$	114.3	50.6	37.8	27.4	18.4	14.1	19.2	12.7	10.3
$\xi = 10^{-6}$	92.5	36.9	28.9	20.8	13.3	10.3	14.4	10.1	8.2
$\xi = 5.10^{-6}$	61.0	14.2	9.4	10.8	7.4	5.3	7.5	5.6	4.0
$\xi = 10^{-5}$	56.2	10.3	6.9	8.2	5.8	4.0	5.8	4.4	3.1
	Iterative system unknowns								
All preconditioners	17568			28644			43914		
	Max of the local Schur size								
All preconditioners	11766			8886			7032		
	Initialization time								
All preconditioners	196.7			74.0			32.7		

Table 7.22: Detailed parallel performance of the *2-level parallel* method for the Fuselage problem with about 0.5 million elements and 3.3 Mdof when the number of subdomains is varied, for the various variants of the preconditioner and for various choices of ξ.

# subdomains or SMP-nodes	**8**			**16**			**32**		
# processors per subdomain	*1-level parallel*	*2-levels parallel*		*1-level parallel*	*2-levels parallel*		*1-level parallel*	*2-levels parallel*	
	1	2	4	1	2	4	1	2	4
	Total solution time								
$M_{\mathrm{d}-64}$	453.7	322.9	257.6	264.6	180.3	136.9	110.9	72.9	65.1
$\xi = 5.10^{-7}$	499.4	488.8	435.1	262.2	202.3	166.1	143.6	109.1	92.4
$\xi = 10^{-6}$	433.4	397.8	332.0	212.6	184.3	165.3	124.6	103.4	85.2
$\xi = 5.10^{-6}$	277.5	238.8	215.5	151.7	130.4	111.7	86.5	73.5	65.9
$\xi = 10^{-5}$	246.7	215.6	200.1	134.6	122.6	101.6	70.5	-	54.5
$\xi = 5.10^{-5}$	214.0	193.8	185.2	122.1	106.2	91.2	63.4	-	45.9
	Time in the iterative loop								
$M_{\mathrm{d}-64}$	57.2	34.3	21.1	60.8	38.7	26.2	44.5	22.4	25.1
$\xi = 5.10^{-7}$	54.9	56.7	33.7	50.4	45.2	25.1	39.6	27.0	29.4
$\xi = 10^{-6}$	51.9	36.4	30.3	49.8	41.0	28.4	36.7	29.4	29.9
$\xi = 5.10^{-6}$	42.0	26.8	18.8	47.9	28.8	20.3	37.6	27.4	26.1
$\xi = 10^{-5}$	42.2	23.8	17.9	41.8	28.0	18.0	30.2	-	21.3
$\xi = 5.10^{-5}$	38.5	21.2	15.5	44.3	29.7	17.1	34.1	-	18.9
	# iterations								
$M_{\mathrm{d}-64}$	59			79			106		
$\xi = 5.10^{-7}$	59			80			107		
$\xi = 10^{-6}$	59			83			108		
$\xi = 5.10^{-6}$	60			87			114		
$\xi = 10^{-5}$	63			89			116		
$\xi = 5.10^{-5}$	70			103			131		
	Time per iteration								
$M_{\mathrm{d}-64}$	0.97	0.58	0.36	0.77	0.49	0.33	0.42	0.21	0.24
$\xi = 5.10^{-7}$	0.93	0.96	0.57	0.63	0.56	0.31	0.37	0.25	0.28
$\xi = 10^{-6}$	0.88	0.62	0.51	0.60	0.49	0.34	0.34	0.27	0.28
$\xi = 5.10^{-6}$	0.70	0.45	0.31	0.55	0.33	0.23	0.33	0.24	0.23
$\xi = 10^{-5}$	0.67	0.38	0.28	0.47	0.32	0.20	0.26	-	0.18
$\xi = 5.10^{-5}$	0.55	0.30	0.22	0.43	0.29	0.17	0.26	-	0.14
	Preconditioner setup time								
$M_{\mathrm{d}-64}$	235.0	127.1	75.0	137.0	74.8	43.9	43.5	27.7	17.2
$\xi = 5.10^{-7}$	283.0	270.6	239.9	145.0	90.3	74.2	81.2	59.3	40.1
$\xi = 10^{-6}$	220.0	199.9	140.2	96.0	76.5	70.1	65.0	51.2	32.5
$\xi = 5.10^{-6}$	74.0	50.5	35.2	37.0	34.8	24.6	26.0	23.3	16.9
$\xi = 10^{-5}$	43.0	30.3	20.7	26.0	27.8	16.9	17.5	-	10.3
$\xi = 5.10^{-5}$	14.0	11.1	8.3	11.0	9.8	7.3	6.5	-	4.2
	Iterative system unknowns								
All preconditioners	31535			49572			73146		
	Max of the local Schur size								
All preconditioners	13296			10953			7404		
	Initialization time								
All preconditioners	161.5			66.8			22.9		

Table 7.23: Detailed parallel performance of the *2-level parallel* method for the Rouet problem with about 0.33 million elements and 1.3 Mdof when the number of subdomains is varied, for the various variants of the preconditioner and for various choices of ξ . "-" means run not available.

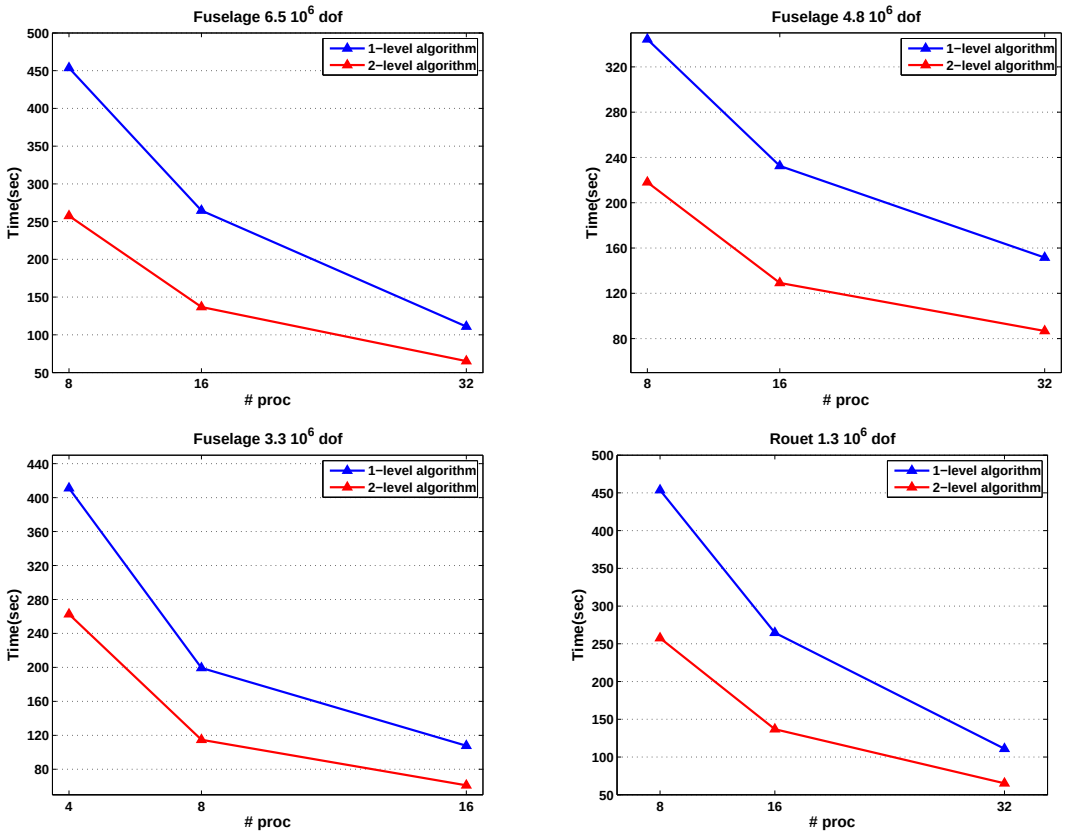


Figure 7.8: Parallel performance of the *2-level parallel* method for the Fuselage and the Rouet test cases, when consuming the same resource as the *1-level parallel* method.

of this solution technique within the complete simulation chain. Other investigations should also be developed to study the effect of the stopping criterion threshold on the overall solution time when the linear solver would be embedded in the nonlinear Newton solver. Although a loose accuracy would certainly delay the nonlinear convergence some saving in computing time could be expected thanks to cheaper linear solves. In that context, we might also reuse the preconditioner from one nonlinear iteration to the next, specially close to the nonlinear convergence. Another possible source of gain for the sparse variant is a more sophisticated dropping strategy. More work on this aspect would also deserve to be invested as well as on the automatic tuning of the threshold parameter.

In the context of parallel SMP platforms, we have illustrated the benefit of a *2-level parallel* implementation when the memory storage is the main bottleneck. In this case, the *2-level parallel* algorithm can be of great interest. Such an implementation can also be attractive in situation where the increase of the number of iterations is significant when the number of domains is increased. Finally, because the iterative part (phase 3 in contrast to phase 1 and 2) performs efficiently in parallel, the possibility of reusing the preconditioner between various consecutive Newton steps could make this variant even more attractive in practice.

Chapter 8

Preliminary investigations in seismic modelling

8.1 Introduction

Important applications of the acoustic wave equation can be found in many fields, for instance, in geophysics, marine, aeronautics and acoustics. The wave equation can be solved either in the time-domain or in the frequency-domain. In this chapter, we investigate the parallel performance of an hybrid solver in the frequency-domain, for problems related to the seismic wave propagation [82, 100]. The study presented in this chapter has been developped in collaboration with the members of the SEISCOPE consortium (http://seiscope.unice.fr/).

Frequency-domain full-waveform tomography has been extensively developed during last decade to build high-resolution velocity models [12, 80, 96]. One advantage of the frequency-domain is that inversion of a few frequencies are enough to build velocity models from wide-aperture acquisitions. Multisource frequency-domain wave modeling requires the solution of a large sparse system of linear equations with multiple right-hand sides (RHS). In $2D$, the traditional method of choice for solving these systems relies on sparse direct solvers because multiple right-hand side solutions can be efficiently computed once the LU factorization of the matrix was computed. However, in $3D$ or for very large $2D$ problems, the matrix size becomes very large and thus the memory requirements of the sparse direct solvers preclude applications involving hundred millions of unknowns. To overcome this limitation, the development of efficient hybrid methods for large $3D$ problems remains a subject of active research. Recently, we investigate the hybrid approach in the context of the domain decomposition method based on the Schur complement for $2D/3D$ frequency-domain acoustic wave modeling [95].

A possible drawback of the hybrid approach is that the time complexity of the iterative part linearly increases with the number of right-hand sides, when traditional Krylov subspace method is simply used on the sequence of right-hand sides. For a sequence of right-hand sides that do not vary much, a straightforward idea is to use the former solution as an initial guess for the next solve. More sophisticated changes in the Krylov solver can be envisaged ranging from the seed approach [92], where the initial guess vector is chosen so that it complies with an optimum norm or an orthogonality criterion over the Krylov space associated with the previous right-hand sides, to the more elaborated approaches as GCRO-DR recently proposed [79] that further exploits deflating ideas present in GMRES-E [73] or GMRES-DR [74]. The underlying idea in these latter techniques is to recycle Krylov vectors to build the space where the minimal residual norm solution will be searched for the subsequent systems. Other possibly complementary alternatives would consist in improving a selected preconditioner [46]. In most of the situations, the linear systems are solved using an application dependent preconditioner whose efficiency and cost are controlled by a few

parameters. Because the preconditioner is used for all the right-hand sides some extra effort can be dedicated to improve it because the extra work involved in its construction can be amortized along the solution of the right-hand sides. Even though such an approach is certainly beneficial, other complementary techniques can be envisaged such as the use of *2-level parallel* algorithm that aims at reducing the number of iterations by decreasing the number of subdomains. This latter technique can be also combined with more sophisticated algorithms that exploit the multiple right-hand side feature.

We omit the description of such approaches in this thesis, we focus only on the traditional Krylov solver method. Our goal is the study of the numerical behaviour and the parallel performance of the hybrid method. However, we analyze the *2-level parallel* algorithm that can be used in both traditional or sophisticated Krylov solvers. The outline of this chapter is as follow. We briefly describe the Helmholtz equation, then in Section 8.2, we present our experimental environment with the description of the test problems. In Section 8.3, we analyze the accuracy of the hybrid approach by comparing to the results obtained from an analytical solution for a homogeneous media and with a direct method solution for a $3D$ heterogeneous media. In Section 8.4, and Section 8.5, a parallel performance study for respectively large $2D$ and $3D$ models arising in geophysical applications are reported. A performance comparison with a direct approach is also presented. Finally, we evaluate the parallel performance of the *2-level parallel* algorithm, and report a set of numerical results.

Helmholtz equation The visco-acoustic wave equation is written in the frequency-domain as

$$\frac{\omega^2}{\kappa\left(\mathbf{x}\right)}p\left(\mathbf{x},\omega\right)+\nabla\left(\frac{1}{\rho\left(\mathbf{x}\right)}\nabla p\left(\mathbf{x},\omega\right)\right)=-s\left(\mathbf{x},\omega\right) \tag{8.1}$$

where $\rho\left(\mathbf{x}\right)$ is the density, $\kappa\left(\mathbf{x}\right)$ is the bulk modulus, ω is angular frequency, $p\left(\mathbf{x},\omega\right)$ and $s\left(\mathbf{x},\omega\right)$ denote the pressure and source respectively. Equation (8.1) can be recast in matrix form as

$$\mathcal{A}\mathbf{p}=\mathbf{s},$$

where the complex-valued impedance matrix $\mathcal{A}$ depends on ω, κ and ρ. The vector $\mathbf{p}$ and $\mathbf{s}$ are of dimension equal to the product of the dimensions of the cartesian computational grid. We discretized Equation (8.1) with the mixed-grid finite-difference stencil [76] which has an accuracy similar to that of 4^{th}-order accurate stencils while minimizing the numerical bandwidth of $\mathcal{A}$. This is a key point to mitigate the fill-in during LU factorization.

8.2 Experimental framework

We start with a brief description of the proposed implementation framework for these applications. The direct method used in the local subdomains is based on a multifrontal approach implemented by the sparse direct solver MUMPS. The iterative method used to solve the interface problem is the right preconditioned GMRES method. We choose the ICGS (Iterative Classical Gram-Schmidt orthogonalization) strategy which is suitable for parallel implementation. The iterations began with a zero initial guess and were stopped when the normwise backward error becomes smaller than 10^{-3} or when 500 steps are taken. We use a variant of our preconditioner based on the introduction of a complex perturbation to the Laplace operator [40], resulting in a shifted additive Schwarz preconditioner. The experiments were carried out in single precision arithmetic using the IBM JS21 supercomputer described in Section 7.2.2.

To investigate the parallel performance of the hybrid approach, we consider a few real life problems from the geophysics applications. These test cases are described below.

8.2.1 The $2D$ Marmousi II model: synthetic data in a structurally complex environment

The $2D$ Marmousi II model is available on the University of Houston website (http://www.agl.uh.edu/) the velocity and the density are shown in Figure 8.1. We test several frequencies starting from $100\,Hz$ up to $200\,Hz$. This model, covering an area of $17 \times 3.5\,km^2$, was modeled by 4 grid points per minimum wavelength, and with 20 layers of PML (Perfectly-Matched Layer [15]) in each direction. We report in Table 8.1, the global mesh size and the total number of unknowns, for each of the tested frequencies.

	$100Hz$	$120Hz$	$140Hz$	$160Hz$	$180Hz$	$200Hz$
grid	1441×6841	1721×8201	2001×9521	2281×10921	2561×12281	2841×13641
unknowns	$9.8 \cdot 10^6$	$14 \cdot 10^6$	$19 \cdot 10^6$	$25 \cdot 10^6$	$31.4 \cdot 10^6$	$38.7 \cdot 10^6$

Table 8.1: Grid size and number of unknowns for each of the tested frequencies.

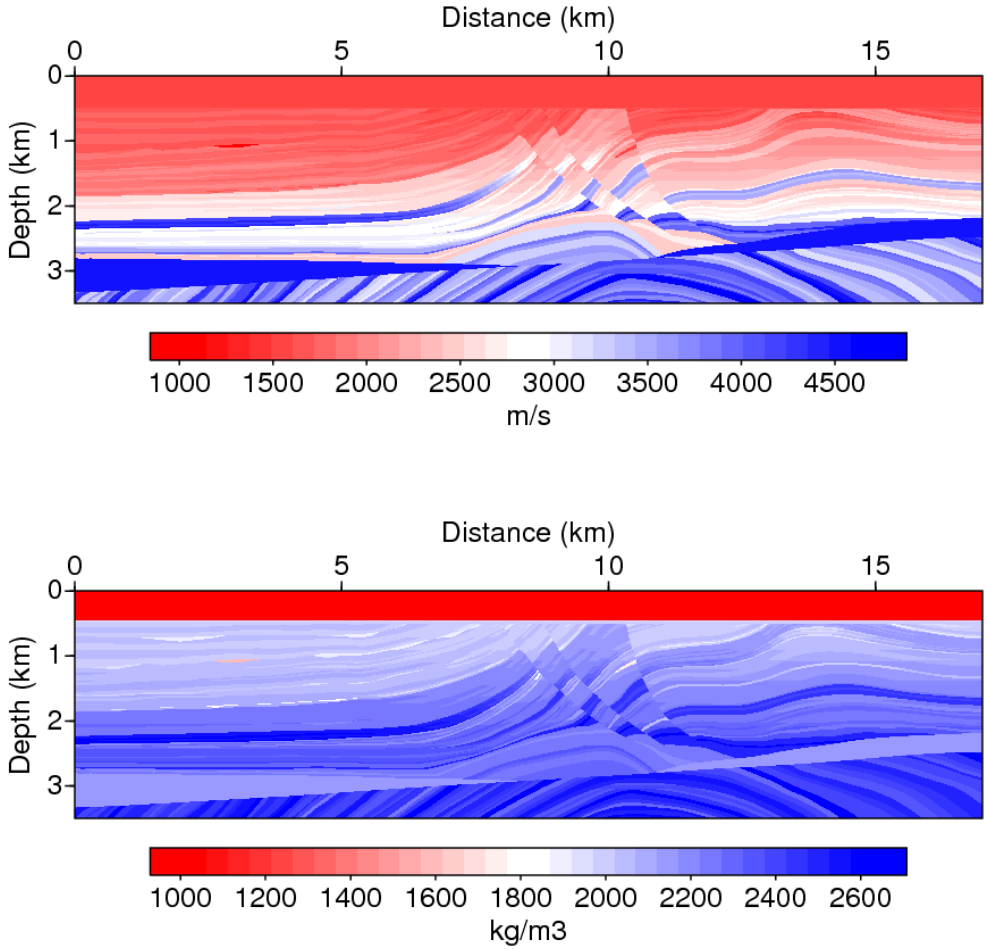


Figure 8.1: Velocity (top) and density (bottom) model of the Marmousi II data set.

8.2.2 The $3D$ Overthrust model: SEG/EAGE

The $3D$ SEG/EAGE Overthrust model is a constant-density acoustic model covering an area of $20 \times 20 \times 4.65\, km^3$ (Figure 8.2). We performed simulations for the $7\, Hz$ frequency. The model was resampled with a grid interval of 75 m corresponding to 4 grid points per minimum wavelength at 7 Hz. This led to a velocity grid of $277 \times 277 \times 73$ nodes including PML (Perfectly-Matched Layer [15]) layers (5.6 millions of unknowns).

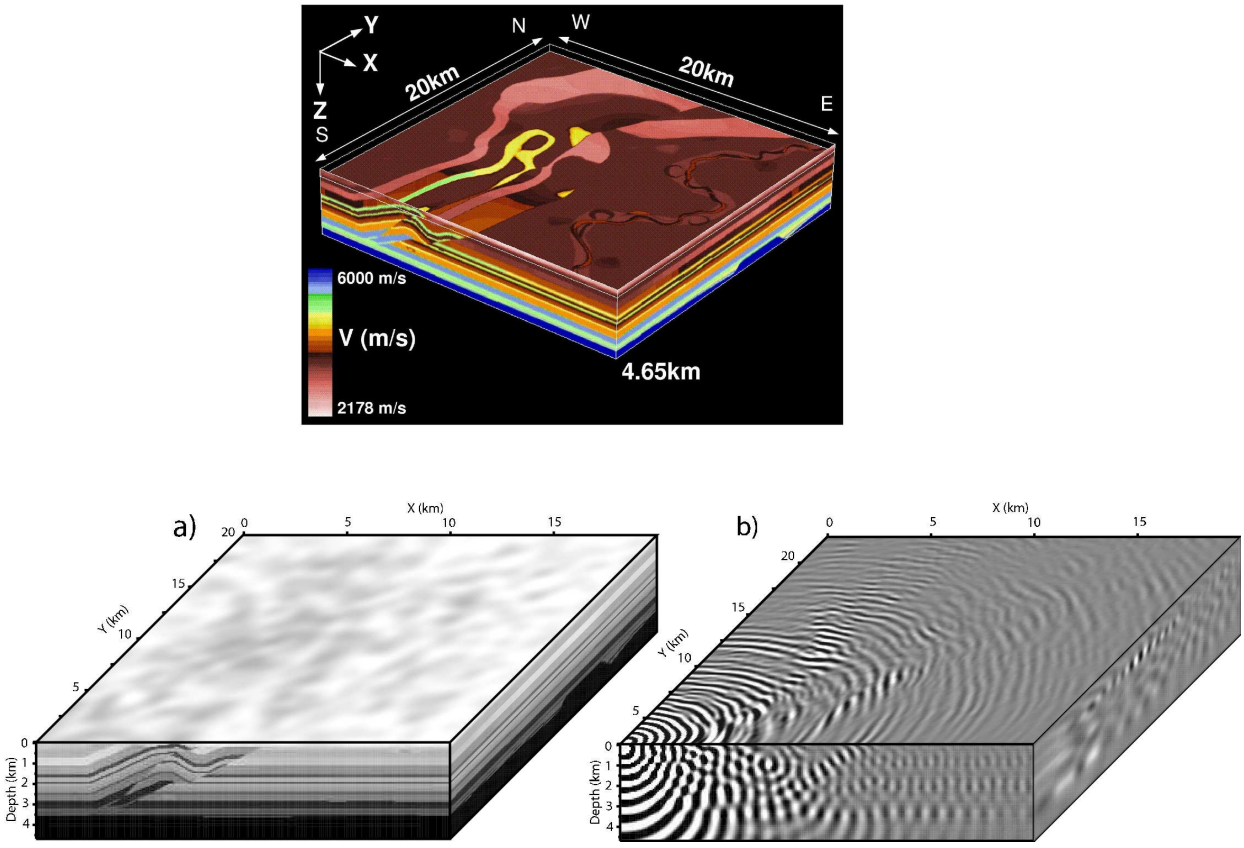


Figure 8.2: (top) $3D$ SEG/EAGE Overthrust model (a) $3D$ SEG/EAGE Overthrust model (b) $7\, Hz$ monochromatic wavefield computed solution.

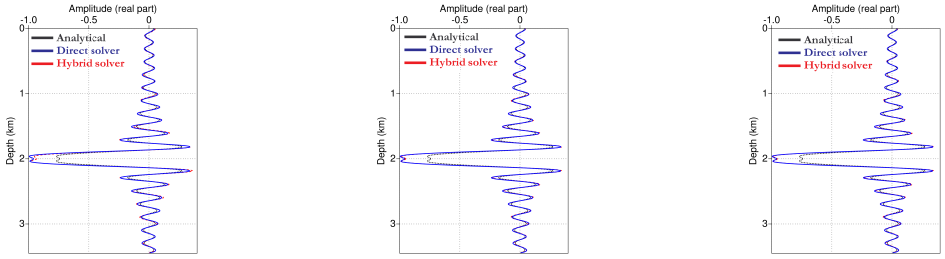
8.3 Numerical accuracy analysis

This section is devoted to analyze the accuracy of the computed solution for the hybrid method. Contrarily to the direct solvers that usually compute solutions that are accurate to machine precision level, the iterative solvers can be monitored to deliver solution with a prescribed accuracy controlled by the stopping criterion threshold. Since an iterative method computes successive approximations of the solution of a linear system, we performed an heuristic analysis of the stopping criterion η_b required by our application. Because of the uncertainty of the data, the simulation does not need very high level of accuracy, we can stop the iterative process much before attaining the machine precision level. On those problems, all the simulations are performed in 32-bit arithmetic for both the direct and the iterative methods.

We first compare the analytical, direct and hybrid solver solution of a $3D$ Green function in a homogeneous media (see Figure 8.3).

$$G(\mathbf{r}) = \frac{\mathbf{1}}{\mathbf{4}\pi}\frac{\rho}{\mathbf{r}}\mathrm{e}^{\frac{\mathbf{i}\omega\mathbf{r}}{\mathbf{c}}}.$$

A value of $\eta_b = 10^{-3}$ seems to provide the best compromise between accuracy and iterations count.



(a) $\eta_b = 10^{-1}$. (b) $\eta_b = 10^{-2}$. (c) $\eta_b = 10^{-3}$.

Figure 8.3: Comparison of the solution of a $3D$ Green function in a homogeneous media for different values of the stopping criterion η_b. Note that a value of $\eta_b = 10^{-3}$ provides a meaningful solution compared to both the analytical and the direct solutions.

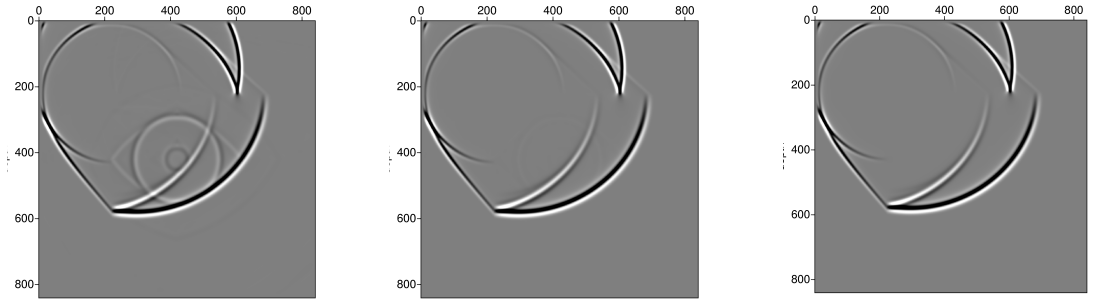
Secondly, we compare numerical solutions in $2D$ heterogeneous media provided by a finite-difference frequency-domain method based on a direct and hybrid solvers respectively. The velocity model is a corner-edge model composed of two homogeneous layers delineated by a horizontal and vertical interfaces forming a corner. The grid is 801×801 with a grid step size of 40 m. Velocities are $4km/s$ and $6km/s$ in the upper-left and bottom-right layers respectively. The source wavelet is a Ricker with a dominant frequency of 5 Hz. The snapshots computed with the hybrid solver for different values of η_b (10^{-1}, 10^{-2}, and 10^{-3}), are shown in Figure 8.4 for a decomposition of 2×2 subdomains (top, a, b and c) and for a decomposition of 4×4 subdomains (bottom, d, e and f). A value of $\eta_b = 10^{-1}$ clearly provides unacceptable solutions as illustrated by the diffraction from the intersection between the subdomains in Figure 8.4 (a and d) while the solution computed with $\eta_b = 10^{-3}$ provides an accurate solution.

Time-domain seismograms computed with both the direct and the hybrid solvers for several values of η_b are shown in Figure 8.5. Two hundred receivers have been used. Comparison with the direct solver at $\eta_b = 10^{-3}$ showed quite similar results. Implementing the hybrid solver into a finite difference Frequency domain Full Waveform Tomography (FFWT) code is the next step to assess precisely which convergence tolerance is needed for imaging applications. Preliminary results using the inverse crime [12], i.e., the same solver is used to generate the data and to perform inversion, seems to confirm that $\eta_b = 10^{-3}$ is an appropriated value.

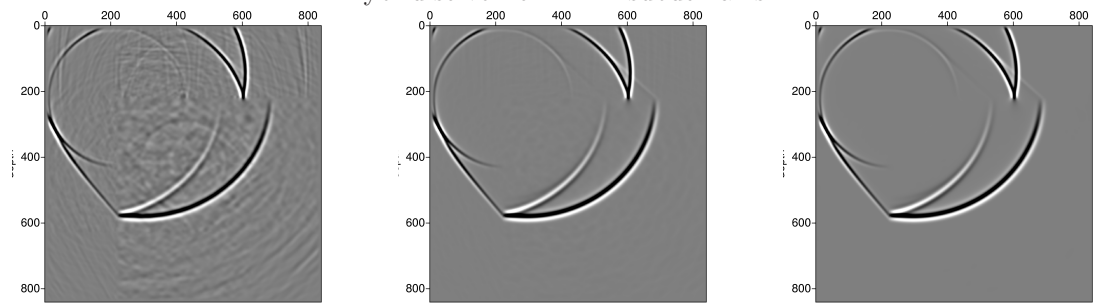
8.4 Parallel performance investigations on $2D$ problems

In this section, we strive to study the numerical behaviour and analyze the parallel efficiency of the proposed hybrid method. We intend to present and evaluate the parallel performance of the hybrid approach and compare its computational cost with a direct approach.

In that respect, we investigate experiments of the Marmousi II model, when we vary the tested frequency from 100 Hz to 200 Hz, that is, when we increase the overall size of the problem from

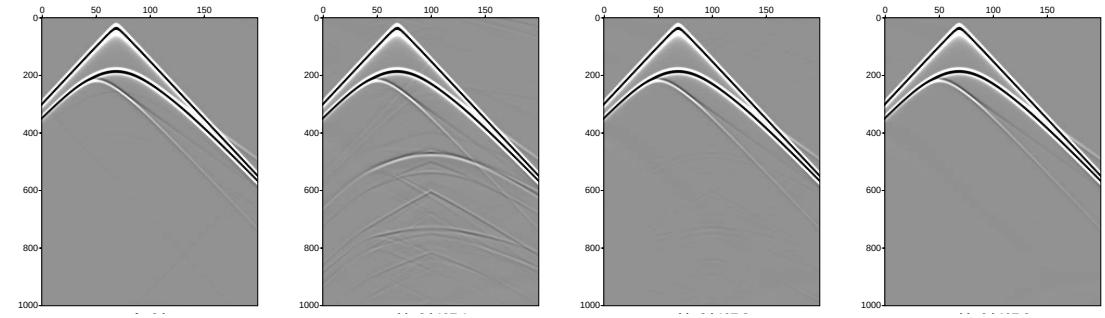


(a) $\eta_b = 10^{-1}$. (b) $\eta_b = 10^{-2}$. (c) $\eta_b = 10^{-3}$.
hybrid solver for 2×2 subdomains.



(d) $\eta_b = 10^{-1}$. (e) $\eta_b = 10^{-2}$. (f) $\eta_b = 10^{-3}$.
hybrid solver for 4×4 subdomains.

Figure 8.4: Snapshots computed in the corner-edge model with different values of the stopping criterion η_b. Note the diffraction at the intersection between the subdomains for $\eta_b = 10^{-1}$.



(a) *direct*. (b) $\eta_b = 10^{-1}$. (c) $\eta_b = 10^{-2}$. (d) $\eta_b = 10^{-3}$.

Figure 8.5: Seismograms computed in the corner-edge model with direct solver (a) and with different value of η_b for the hybrid solver. η_b is 10^{-1} (b), 10^{-2} (c) and 10^{-3} (d).

$9.8 \cdot 10^6$ up to $38.7 \cdot 10^6$ unknowns. The global mesh size and the total number of unknowns, for each of the tested frequencies are reported in Table 8.1. For each of these frequencies (fixed mesh size), we also expose in Table 8.3, a detailed result of the parallel performance, when increasing the number of subdomains. We note that the submesh of the border subdomain, is increased by 40 points in each direction to include the PML layers.

To get an idea of the effect of the size and the number of subdomains on the convergence of the hybrid method, we report in Table 8.3, the number of iterations required to attain the desired accuracy of 10^{-3}. It is easy to see the pronounced increases in the number of iterations, when increasing the number of subdomains. This behaviour is typically expected for such problems. For example, for a fixed mesh size at $9.8\ 10^6$ unknowns (frequency equal to 100 Hz), the number of iterations increases from 82 up to 313 iterations when increasing the number of subdomains from 16 up to 64 subdomains. That is, it increases linearly with the number of subdomains. This increase is rather moderate when increasing the overall size of the problem; that is, when increasing the frequency, for a fixed number of subdomains. For example, we see that for a decomposition into 16 subdomains, the number of iterations grows up from 82 to 110 when increasing the size of the problem from $9.8\ 10^6$ unknowns up to $38.7\ 10^6$ unknowns. A first conclusion regarding the numerical behaviour of the method is, that we should strive to keep as small as possible the number of subdomains.

When looking at the parallel efficiency of the hybrid method, we should analyze the computational cost of the main three phases of our algorithm, as we did in the previous chapters. A comparison of the timing results of each of these three phases is illustrated in Table 8.3. We would like to underline the fact that, even with the growth in the iteration numbers, a significant decrease in the computing time can be observed when increasing the number of processors.

This speedup is highlighted for the first two phases of our algorithm. Let us evaluate the *phase1*, which consists in the factorization of the local problem and the computation of the local Schur complement. When we increase the number of processors, the subdomains size becomes smaller and thus the factorization becomes faster decreasing the *phase1* computing time by a great ratio. Likewise, the setup of the preconditioner (*phase2*) gives rise to an effective speedup. For the same reason, when we increase the number of subdomains, the interface size of a subdomain becomes smaller; this decreases the computational cost required to build the preconditioner. Contrary to the first two phases, the cost of the *phase3* which is the iterative loop phase, increases with the number of subdomains. This is due to the significant growth of the number of iterations, when increasing the number of subdomains. For the solution of one right-hand side, the overall computing time decreases linearly when the number of processors increases. It can be seen that, the global computational cost is about 3.3 times faster when increasing the number of processors from 16 up to 64.

We now compare the performance of the hybrid method with that of a direct method which is popular in this application area. First of all, we depict in Table 8.2, the required memory to perform the simulation with both the hybrid and the direct approaches. For the hybrid approach, this corresponds to the sum of the memory allocated to perform the concurrent factorizations, the storage of both the Schur complement and the dense preconditioner, and finally the sum of the required workspace for the full-GMRES solver. For the direct approach, it consists into the memory required to perform the factorization. In Table 8.2, we report the required memory for all the frequencies (except for 180 and 200 Hz for the direct solver due to the unaffordable memory requirements) for the different decompositions illustrated. As shown in Table 8.2, the memory needed by the direct solver increases with the number of processors. This behaviour comes from the fact that the factorization needs a set of communication buffers to manage the parallel load balancing between computational cost and communication cost. Contrarily, for the hybrid approach, it can be seen that the amount of required memory slightly decreases or remains roughly constant when increasing the number of processors. We should highlight the fact that, in term of memory storage, the hybrid approach is more scalable and about 3 times less expensive than the direct approach. For the $2D$ simulations, the memory complexity of the direct approach

is affordable, while for the $3D$ problems it might no longer be true.

# processors		16	32	36	64	72
100 Hz	*Direct*	24.8	31.6	34.9	48.6	52.4
	Hybrid	15.7	16.0	15.8	15.7	15.3
120 Hz	*Direct*	35.7	46.7	49.1	69.4	76.3
	Hybrid	23.7	23.3	23.3	22.8	22.8
140 Hz	*Direct*	50.1	64.3	68.9	94.2	103.1
	Hybrid	32.5	33.4	31.8	32.2	32.1
160 Hz	*Direct*	67.2	88.7	88.6	122.6	133.3
	Hybrid	43.2	43.0	42.5	42.3	42.3
180 Hz	*Direct*	-	-	-	-	-
	Hybrid	54.9	54.0	55.7	54.3	54.8
200 Hz	*Direct*	-	-	-	-	-
	Hybrid	69.3	69.3	69.8	69.2	69.6

Table 8.2: Comparison of the data storage needed by the hybrid and the direct methods. Total memory of all processors is reported here in GBytes. "-" means too large memory requirement.

From the computational point of view, we report in Table 8.4, the cost of the main step of a direct approach. This consists into the analysis step, the factorization step, and finally the solution step. The direct solver used here is MUMPS, it was called in the context of unsymmetric complex matrices. In term of permutation, we have performed experiments using the Metis nested dissection routine. We have performed runs using the assembled matrix format, that allows the solver to make a full analysis and then distributes the matrix among the processors. We exclude the time of the assembling and the distribution of the matrices entries, and we report the minimum time required after several runs. Experiments are reported for the frequencies 100 Hz up to 160 Hz, where we omit both the last two frequencies (180 and 200 Hz) and the 72 processor runs, due to memory requirements that induce swap effects. If we compare only the factorization time (excluding the analysis, the distribution and the solve time) of the direct method with the overall computing time of the hybrid approach, we can clearly observe that the hybrid approach remains faster than the direct one. It is around twice as fast as the direct method for a small number of subdomains (for example on 16 subdomains), whereas it is around 3 times faster on larger number of subdomains (for example on 64 subdomains).

Multisource frequency-domain simulation, requires the solution for multiple right-hand sides. On this respect, we should mention the required time for the solution of one right-hand side with the two approaches. For that we report in Table 8.3, the elapsed time to solve one right-hand side using the hybrid approach, that is the cost of the iterative loop, and the cost of the forward/backward substitutions. We report also in Table 8.4, the required time for one solve with the direct approach. On the hybrid method, it is clear that when increasing the number of processors, the computational cost of one right-hand becomes more expensive. This behaviour was expected because of the increase on the number of iterations, which leads to an increase of the time spent in the iterative loop. However, for the direct approach, we can observe that the required time for one right-hand side slightly decreases when increasing the number of processors. For example, for a decomposition with 16 subdomains, we can observe that the cost for the solution with one right-hand side of the hybrid approach is similar to the direct one, while it performs 1.5 times slower than the direct solver on 64 subdomains. Consequently for a moderate number of subdomains the hybrid and the direct solvers behave comparably for multiple right-hand sides. For a large number of subdomains, the hybrid approach is around 3 times faster than the direct method for one right-hand side. Due to the lack of numerical scalability of the preconditioner and the efficiency of the forward/backward substitutions of the sparse direct solver, the hybrid

	Total solution time				
# processors	16	32	36	64	72
100 Hz	55.8	33.2	28.6	17.9	18.1
120 Hz	94.2	53.6	46.4	26.6	27.8
140 Hz	145.1	84.3	68.7	45.4	41.9
160 Hz	204.9	121.7	107.3	62.5	66.0
180 Hz	288.2	175.4	144.4	86.6	78.5
200 Hz	392.0	238.7	199.0	116.8	109.4
	Time for one RHS				
# processors	16	32	36	64	72
100 Hz	4.7	6.4	6.6	7.2	9.4
120 Hz	7.6	10.2	8.7	10.5	13.2
140 Hz	10.3	14.4	13.8	18.5	19.8
160 Hz	13.0	16.7	21.0	25.3	32.3
180 Hz	20.3	26.1	29.2	34.9	32.0
200 Hz	21.6	29.9	34.0	47.3	44.4
	Time in the iterative loop				
# processors	16	32	36	64	72
100 Hz	3.0	5.5	5.8	6.7	9.0
120 Hz	5.1	8.9	7.6	9.8	12.6
140 Hz	6.9	12.6	12.2	17.5	18.9
160 Hz	8.5	14.4	18.9	24.0	31.1
180 Hz	14.6	23.0	26.5	33.2	30.5
200 Hz	14.5	25.9	30.6	45.3	42.2
	# iterations				
# processors	16	32	36	64	72
100 Hz	82	121	158	313	318
120 Hz	102	137	152	295	346
140 Hz	102	138	183	292	384
160 Hz	97	127	213	305	485
180 Hz	125	162	241	380	385
200 Hz	110	148	223	472	451
	Preconditioner setup time				
# processors	16	32	36	64	72
100 Hz	4.9	5.8	3.8	1.7	1.4
120 Hz	8.2	9.8	6.3	2.8	2.3
140 Hz	12.9	15.7	10.1	4.3	3.9
160 Hz	19.0	23.1	14.7	6.4	5.7
180 Hz	26.8	32.8	20.8	9.0	8.0
200 Hz	35.7	52.3	28.5	12.3	10.7
	Initialization time				
# processors	16	32	36	64	72
100 Hz	46.2	20.9	18.2	9.0	7.3
120 Hz	78.5	33.6	31.4	13.3	12.2
140 Hz	121.9	54.2	44.8	22.6	18.3
160 Hz	172.9	81.9	71.6	30.8	28.0
180 Hz	241.1	116.5	94.4	42.8	38.5
200 Hz	334.6	156.5	136.4	57.2	54.4

Table 8.3: Detailed performance for the $Marmousi\ II$ test case when varying the frequency, that is varying the global size of the problem from 9.8 $\cdot 10^6$ unknowns up to 38.7 $\cdot 10^6$ unknowns, and also when the number of processors is varied from 16 to 72.

	Time for analysis			
# processors	16	32	36	64
100 Hz	295.5	290.4	342.3	292.6
120 Hz	459.0	444.0	537.7	452.6
140 Hz	705.3	695.5	824.6	704.0
160 Hz	1061.3	1058.0	1184.0	1043.2
	Time for factorization			
# processors	16	32	36	64
100 Hz	97.2	63.9	57.1	48.5
120 Hz	167.1	108.4	107.7	79.0
140 Hz	283.5	181.1	182.6	122.1
160 Hz	517.4	318.4	268.4	247.3
	Time for one RHS			
# processors	16	32	36	64
100 Hz	5.3	4.8	4.8	4.6
120 Hz	8.0	7.1	7.0	6.9
140 Hz	11.0	10.0	9.5	11.3
160 Hz	18.2	20.5	12.8	18.9
	Total solution time			
# processors	16	32	36	64
100 Hz	398.0	359.0	404.2	345.7
120 Hz	634.1	559.5	652.4	538.6
140 Hz	999.8	886.7	1016.7	837.4
160 Hz	1596.9	1396.9	1465.2	1309.5

Table 8.4: Detailed computing time needed by the direct solver for the *Marmousi II* test case when varying the frequency from 100 Hz to 160 Hz, that is varying the global size of the problem from 9.8 $\cdot 10^6$ unknowns up to 25 $\cdot 10^6$ unknowns, and also when the number of processors is varied from 16 to 64.

solver becomes slower if more than 15 right-hand sides have to be considered. We notice that only the factorization time of the direct method is taken into account here. If we take into account the analysis time of the direct method, then the hybrid method is still faster up to a number of right-hand sides around 120.

8.5 Parallel performance investigations on $3D$ problems

In this section, we present a preliminary work on a more realistic $3D$ case that is the Overthrust SEG/EAGE problem. We first illustrate comparison of the computational cost of both approaches (direct and hybrid), and then we evaluate the parallel performance of the hybrid method.

Due to the memory requirement of the direct solver, the simulations based on the direct solver can only be performed on 75% of the model corresponding to $4.25 \cdot 10^6$ unknowns using 192 processors of our target machine. Whereas the hybrid approach allows us to perform a simulation on the complete model only using 48 processors, $48 = 6 \times 4 \times 2$ subdomains.

Algorithm	All memory GBytes	Max memo per proc	Init or factorization	Precond-itioner	# iterations	Time per RHS	Total time
Direct	234.9	1.5	2876	-	-	9.1	2885
Hybrid	101.3	0.5	36	40	231	52.2	128

Table 8.5: Comparison between direct and hybrid methods to solve the Overthrust SEG/EAGE problem mapped onto 192 processors. Timings and storage of both methods are reported. Total memory of all processors in GBytes is also reported.

subdomains			Memory (GB)		Initial-ization	Precond-itioner	# of iterations	Time per RHS	Total time
#	size	interface	All	Max/proc					
48	$67 \times 44 \times 31$	11570	191.8	3.71	638	573.1	105	96.4	1307.5
50	$54 \times 54 \times 31$	11056	191.6	3.57	614	497.1	81	67.8	1178.9
72	$45 \times 45 \times 31$	8833	179.3	2.01	334	273.5	103	73.9	681.4
81	$30 \times 30 \times 63$	8760	182.1	1.87	224	256.3	109	77.4	557.7
96	$45 \times 33 \times 31$	7405	167.8	1.53	184	153.8	119	61.1	398.9
98	$38 \times 38 \times 31$	7216	169.7	1.52	189	141.5	148	66.7	397.2
128	$33 \times 33 \times 31$	6121	161.2	1.08	116	88.5	134	53.7	258.2
162	$30 \times 30 \times 31$	5281	151.6	0.80	79	58.4	153	53.0	190.4
192	$33 \times 33 \times 21$	5578	147.4	0.74	90	78.2	235	85.8	254.0

Table 8.6: Parallel performance of the whole Overthrust SEG/EAGE problem (5.6 Mdof) when varying the number of subdomains. Timings and storage of the hybrid method are reported. Total memory of all processors in GBytes is also reported.

For the sake of comparison between the two approaches, experimental results are reported for calculation performed on 75% of the model mapped onto 192 processors. The results of these simulations are summarized in Table 8.5. The hybrid solver requires 2.3 less memory than the direct one. Moreover, the hybrid approach can perform such a simulation on a smaller number of processors. From a computational viewpoint, it is clear that the hybrid method is 22 times faster than the direct one. This is true for one right-hand side and still true for a number of right-hand sides smaller than 64. Beyond this number, the high cost of the factorization is amortized and the direct method starts to be faster. We note that we evaluate here the iterative algorithm using a sequence of solutions computed simply using full-GMRES; block or deflation strategies could be considered that would improve the efficiency of the hybrid technique.

We now evaluate the parallel performance of the hybrid method. All simulations were performed at 7 Hz on the whole model corresponding to 5.6 millions of unknowns. We display in Table 8.6 results where we vary the number of subdomains from 48 up to 192. We report the computational cost and the required memory to perform the simulation for each of the decompositions. Increasing the number of subdomains reduces the memory requirement from 191 down to 147 GBytes, and the computational cost from 1307.5 seconds to 190.4 seconds. We remark that, the required memory storage decreases with the number of processors, contrary to the direct method where we observe an increase of memory due to some buffer overheads. We can notice that the best computing performance of the three kernels of the algorithm is achieved when the subdomain shape is closed to a cube (perfect aspect ratio). In other term, when the size of the interface is minimized that is because, the three phases of the hybrid method are related to the size of the local interface. For example, we can observe that, the computing time for the decomposition into 162 subdomains, each of size $30 \times 30 \times 31$ (size interface equal to 5281) is better than the decomposition onto 192 subdomains, each of size $33 \times 33 \times 21$ (size interface equal to 5578). The solution time is for this example 53.0 seconds on 162 subdomains versus 96.4 seconds on 192 subdomains, whereas this latter is roughly similar for 128 and 162 subdomains where the local size of subdomains is respectively $33 \times 33 \times 31$ and $30 \times 30 \times 31$.

8.6 Parallel efficiency of the *2-level parallel* implementation

Initially this work was motivated by the fact that, most acoustic simulations require multiple right-hand side solution to build a good model of the continuum. We believe that exploiting *2-levels* of parallelism [49] can be very suitable for these simulations. Because of the lack of numerical scalability, such an implementation might enable us to achieve higher efficiency and performance of the hybrid solver. The goal of this section is to demonstrate and evaluate the performance of the *2-level parallel* algorithm. The first subsection describes the efficiency of the algorithm when used to improve the numerical behaviour of the hybrid method, whereas the second subsection illustrates the ability of the *2-level parallel* algorithm to run at higher performance of the available computing resources.

8.6.1 Numerical benefits

In the Section 8.4 and Section 8.5, the discussion was closed to the fact that, for these simulations an efficient algorithm has to perform the iterative loop as fast as possible. The goal is to keep very small its computational cost especially when dealing with multiple right-hand side simulations. We should mention that, as analyzed above, the convergence of the iterative solver depends highly on the number of subdomains. Thus, keeping small or fixed the number of iterations is critical to make reachable our goal. In other term, we should strive to improve the numerical behaviour of the hybrid method by, instead of increasing the number of subdomains, increasing the number of processors per subdomain.

We report in Table 8.7 the number of iterations, the computing time of the iterative loop, and the required time for the solution of one right-hand side for the $2D$ Marmousi II problem. We consider three frequencies $100Hz$, $120Hz$, and $140Hz$. For each frequency, we fix the number of processors, and compare the computational cost of the direct method, the *1-level parallel* algorithm and the different possible decompositions for the *2-level parallel* algorithm.

We mention that, by decreasing the number of subdomains, the size of the local problem increases giving rise to larger Schur complements. In these tables, it can be seen that even with this latter constraint, the time of the iterative loop is reduced by a great factor. This is due to the fact that each subdomain is handled in parallel and the required number of iterations is reduced. It is clear that for a fixed number of processors, when reducing the number of subdomains by a factor of 4, while running each subdomain in parallel over a grid of 4 processors, the iterative loop

Available processors	Algo	# subdomains	Processors/ subdomain	# iter	Iterative loop	One right-hand side
Frequency equal to 100 Hz						
32 processors	*Direct*	-	-	-	-	4.8
	1-level parallel	32	1	121	5.5	6.4
	2-level parallel	16	2	82	2.2	3.6
	2-level parallel	8	4	17	0.6	2.9
36 processors	*Direct*	-	-	-	-	4.7
	1-level parallel	36	1	158	5.8	6.6
	2-level parallel	18	2	59	1.5	2.7
64 processors	*Direct*	-	-	-	-	4.6
	1-level parallel	64	1	313	6.7	7.2
	2-level parallel	32	2	121	4.1	4.9
	2-level parallel	16	4	82	1.6	2.7
72 processors	*Direct*	-	-	-	-	4.6
	1-level parallel	72	1	318	9.0	9.4
	2-level parallel	36	2	58	4.3	4.9
	2-level parallel	18	4	59	1.2	2.2
Frequency equal to 120 Hz						
32 processors	*Direct*	-	-	-	-	7.1
	1-level parallel	32	1	137	8.9	10.2
	2-level parallel	16	2	102	3.8	5.8
	2-level parallel	8	4	14	0.6	4.0
36 processors	*Direct*	-	-	-	-	7.1
	1-level parallel	36	1	152	7.6	8.7
	2-level parallel	18	2	107	3.8	5.5
64 processors	*Direct*	-	-	-	-	6.9
	1-level parallel	64	1	295	9.8	10.5
	2-level parallel	32	2	137	6.4	7.4
	2-level parallel	16	4	102	2.6	4.0
72 processors	*Direct*	-	-	-	-	6.7
	1-level parallel	72	1	346	12.6	13.2
	2-level parallel	36	2	152	5.6	6.5
	2-level parallel	18	4	107	3.0	4.4
Frequency equal to 140 Hz						
32 processors	*Direct*	-	-	-	-	11.1
	1-level parallel	32	1	138	12.6	14.4
	2-level parallel	16	2	102	5.1	7.8
	2-level parallel	8	4	16	0.9	5.5
36 processors	*Direct*	-	-	-	-	11.1
	1-level parallel	36	1	183	12.2	13.8
	2-level parallel	18	2	117	5.5	7.9
64 processors	*Direct*	-	-	-	-	11.3
	1-level parallel	64	1	292	17.5	18.5
	2-level parallel	32	2	138	8.4	9.8
	2-level parallel	16	4	102	3.4	5.7
72 processors	*Direct*	-	-	-	-	11.2
	1-level parallel	72	1	384	18.9	19.8
	2-level parallel	36	2	183	8.8	10.1
	2-level parallel	18	4	117	4.2	6.2

Table 8.7: Numerical performance and advantage of the *2-level parallel* method compared to the standard *1-level parallel* method for the Marmousi II test cases and for different decompositions.

becomes more than four times faster. This is explained by a better numerical convergence and good efficiency of the parallel kernels involved in the iterative loop when several processors are dedicated to each subdomain. For example in Table 8.7, if we compare the cost of the iterative loop for a fixed number of processors, let say 64, it is easy to see that the standard implementation on 64 subdomains requires 313 iterations to converge in 6.7 seconds whereas the *2-level parallel* implementation on 16 subdomains only requires 82 iterations to converge in 1.6 seconds. It can be seen that the *2-level parallel* algorithm performs faster than both the *1-level parallel* algorithm and the direct method. For example, on Table 8.7, for 64 processors, the direct method requires 4.6 seconds to solve for one right-hand side, the *1-level parallel* method requires 7.2 seconds, whereas the *2-level parallel* method converges in only 2.7 seconds. This is not the only example, the Table 8.7 illustrates a detailed comparison where we quantify the performance of the *2-level parallel* approach. We mention that, in term of memory space, for the same number of subdomains, the *2-level parallel* algorithm requires slightly more memory space than the *1-level parallel* algorithm; this accounts for the communication buffers of the local factorization phase.

Frequency equal to 7 Hz						
Available processors	Algo	# subdomains	Processors/ subdomain	# iter	Iterative loop	One right-hand side
$\approxeq$ 200 processors	*1-level parallel*	192	1	235	79.0	85.8
	2-level parallel	96	2	119	38.2	45.1
	2-level parallel	48	4	105	42.9	51.1
	2-level parallel	50	4	81	28.1	35.5
$\approxeq$ 100 processors	*1-level parallel*	96	1	119	57.0	61.1
	1-level parallel	98	1	148	66.7	66.7
	2-level parallel	48	2	105	62.1	67.8
	2-level parallel	50	2	81	39.1	45.1

Table 8.8: Numerical performance of the *2-level parallel* method compared to the standard *1-level parallel* method for the $3D$ Overthrust SEG/EAGE test case and for different decompositions.

We report in Table 8.8, the number of iterations, the computing time of the iterative loop, and the required time for the solution of one right-hand side for the $3D$ Overthrust SEG/EAGE problem introduced in Section 8.5. For two different numbers of processors, we compare the computational cost of the *1-level parallel* algorithm and the different possible configurations for the *2-level parallel* algorithm. It can be seen that handling in parallel each subdomain enables us to significantly reduce the solution time for a fixed number of processors. For instance on around 200 processors, running the *1-level parallel* method on 192 subdomains (192 processors) needs 85,9 seconds to solve for one right-hand side. The *2-level parallel* algorithm on 50 subdomains with 4 processors per subdomain (200 processors) needs only 35.5 second for the same calculation. For a set of 100 processors, running the *1-level parallel* algorithm on 96 subdomains (96 processors) requires 61.1 seconds. The *2-level parallel* algorithm on 48 subdomains using 2 processors per subdomain (96 processors) needs 67.8 seconds. In this latter case the penalty comes from the poor numerical behaviour on 48 subdomains. This might be due to the bad aspect ratio of the subdomains for that 48 subdomain decomposition. However, if we consider a 50 subdomain decomposition using 2 processors per subdomains (100 processors), we can see the improvement of the *2-level parallel* method. In this latter situation the solution time is around 45.1 seconds for the *2-level parallel* algorithm and 61.1 seconds for the *1-level parallel* algorithm.

Therefore, we conclude, that the *2-level parallel* approach is a promising candidate for the $2D/3D$ multisource acoustic simulations. The idea is that when the number of iterations significantly increases, the *2-level parallel* algorithm should be preferred. It can be the method of choice for a wide range of $2D/3D$ simulations; it enables a better usage of both the memory

and the computing capabilities. We should mention, that using the *2-level parallel* method leads also to decrease the computational cost of both the initialization (*phase1*) and the setup of the preconditioner (*phase2*), this will be studied in the next subsection.

8.6.2 Parallel performance benefits

Motivated by the idea of exploiting as much as possible the performance of the computing machine used to run large simulations, we consider the implementation with *2-levels* of parallelism. When running large simulations that need all the memory available on a node of an parallel machine, standard parallel codes are forced to use only one processor per node, thus leaving the remaining processors of the node in an "idle" state. The *2-level parallel* method allows us to take advantage of the idle processors and to compute the solution faster close to the per node performance. For the parallel machine we use, each node comprises two dual-processors and is equipped with 8 GBytes of main memory.

We report in Table 8.9, the performance results of the *2-level parallel* approach and compare it with the standard *1-level parallel* approach. We report experiments for the $2D$ Marmousi II test case, for the different frequencies reported in Subsection 8.4, and for the various decompositions where the *2-level parallel* algorithm is applied. We structure the discussion by a detailed analysis of the performance on the three phases of our method.

Let us start with *phase1* (the initialization phase) that consists in building the local factorization and computing the local Schur complement. The benefit of the *2-level parallel* algorithm is considerable for all the considered frequencies, the computing time is divided by about 1.3 when using two processors per subdomain and by more than 2 when using 4 processors per subdomain.

When looking at *phase2*, the preconditioner setup phase, we highlight the success of the *2-level parallel* algorithm on achieving high performance compared to the standard *1-level parallel* algorithm. The *2-level parallel* algorithm is about twice faster than the *1-level parallel* algorithm when only two processors are used per subdomain and it is more than 3 times faster when 4 processors are used per subdomain.

A crucial part of the method when dealing with multiple right-hand side solution is the third phase. Thus, the goal is to be as efficient as possible to improve the parallel performance of this step. It can be seen that the *2-level parallel* implementation performs the iterative loop 1.3 time faster than the *1-level parallel* implementation when using two processors per subdomain and more than twice faster than the *1-level parallel* method when using 4 processors per subdomain.

The *2-level parallel* algorithm demonstrates again better performance than the *1-level parallel* method and the direct method for the overall computing time. For example, for a frequency of 100 Hz, the global solution time, on a decomposition of 16 subdomains needs 55.7 seconds using the *1-level parallel* method and 295.5 seconds using a direct solver whereas it needs only 27.0 seconds using the *2-level parallel* algorithm. Similar trend can be observed for other decompositions and other frequencies.

We report in Table 8.10 the performance of the *2-level parallel* algorithm for the $3D$ Overthrust SEG/EAGE problem presented in Section 8.5. The results shown in this table highlight the efficiency of the *2-level parallel* method. It is clear that the *2-level parallel* method decreases the time required for the setup of the preconditioner by a factor of 1.6 when using 2 processors per subdomain and by a factor around 3.6 when using 4 processors per subdomain. Moreover, the time for one right-hand side solution is also decreased by a factor of 1.5 when 2 processors are used to handle a subdomain, and by around 2 when 4 processors are used per subdomain. For example, on a decomposition of 50 subdomains, the *1-level parallel* method requires 1178.9 seconds to perform the simulation whereas the *2-level parallel* method needs only 419.8 to perform the same calculation.

We can conclude, that the *2-level parallel* method is very suitable for the parallel solution of a wide range of real applications. The results in Table 8.9 and Table 8.10, illustrate the parallel performance achieved by the *2-level parallel* method. It allows us to exploit large number of

# subdomains or SMP-nodes	**16**			**32**		**36**	
# processors per subdomain	*1-level parallel*	*2-levels parallel*		*1-level parallel*	*2-levels parallel*	*1-level parallel*	*2-levels parallel*
	1	2	4	1	2	1	2
	Total solution time						
100 Hz	55.8	39.7	27.0	33.2	23.9	28.6	22.0
120 Hz	94.2	70.5	44.6	53.6	39.8	46.4	35.6
140 Hz	145.1	107.3	68.0	84.3	62.2	68.7	52.2
160 Hz	204.9	149.8	91.9	121.7	88.2	107.3	74.8
180 Hz	288.2	212.2	129.6	175.4	123.3	144.4	101.8
200 Hz	392.0	265.1	172.7	238.7	165.0	199.0	146.1
	Time for one RHS						
100 Hz	4.7	3.6	2.7	6.4	4.9	6.6	4.9
120 Hz	7.6	5.8	4.0	10.2	7.4	8.7	6.5
140 Hz	10.3	7.8	5.7	14.4	9.8	13.8	10.1
160 Hz	13.0	9.6	7.1	16.7	11.7	21.0	14.3
180 Hz	20.3	14.4	9.9	26.1	17.9	29.2	19.3
200 Hz	21.6	15.8	11.5	29.9	20.4	34.0	24.0
	Time in the iterative loop						
100 Hz	3.0	2.2	1.6	5.5	4.1	5.8	4.3
120 Hz	5.1	3.8	2.6	8.9	6.4	7.6	5.6
140 Hz	6.9	5.1	3.4	12.6	8.4	12.2	8.8
160 Hz	8.5	6.0	4.1	14.4	9.8	18.9	12.4
180 Hz	14.6	9.9	6.6	23.0	15.6	26.5	16.9
200 Hz	14.5	10.3	6.8	25.9	17.3	30.6	21.4
	# iteration						
100 Hz	82			121		158	
120 Hz	102			137		152	
140 Hz	102			138		183	
160 Hz	97			127		213	
180 Hz	125			162		241	
200 Hz	125			162		241	
	Preconditioner setup time						
100 Hz	4.9	2.8	1.6	5.8	3.2	3.8	2.1
120 Hz	8.2	4.5	2.5	9.8	5.3	6.3	3.5
140 Hz	12.9	7.1	3.9	15.7	8.4	10.1	5.5
160 Hz	19.0	10.4	5.8	23.1	12.1	14.7	7.9
180 Hz	26.8	14.5	7.9	32.8	16.9	20.8	11.1
200 Hz	35.7	19.2	10.5	52.3	23.0	28.5	14.9
	Initialization time						
100 Hz	46.2	33.4	22.7	20.9	15.9	18.2	14.9
120 Hz	78.5	60.2	38.1	33.6	27.1	31.4	25.6
140 Hz	121.9	92.4	58.4	54.2	44.1	44.8	36.7
160 Hz	172.9	129.8	79.1	81.9	64.5	71.6	52.5
180 Hz	241.1	183.3	111.7	116.5	88.4	94.4	71.4
200 Hz	334.6	230.0	150.8	156.5	121.7	136.4	107.1

Table 8.9: Detailed parallel performance of the *2-level parallel* method for the $2D$ Marmousi II problem when varying the frequency from $100\,Hz$ to $200\,Hz$, that is varying the global size of the problem from 9.8 $\cdot 10^6$ unknowns up to 38.7 $\cdot 10^6$ unknowns, and also when the number of subdomains is varied from 16 to 36.

processors with higher efficiency, and to drive down the execution time by a significant factor. For the test cases reported here, the *2-level parallel* method performs much better as both the *1-level parallel* method and the direct method.

8.7 Concluding remarks

This chapter describes the results of preliminary investigations of a shifted variant of our preconditioner for the solution of the Helmholtz equation in two and three dimensional domains.

A very poor numerical scalability of the preconditioner has been observed on the $2D$ examples where the use of a *2-level parallel* implementation is an efficient remedy to reduce the elapsed time when the number of processors is increased. For the $3D$ problem, the scalability is not as worse as in $2D$ and the *2-level parallel* implementation enables some improvement. More experiments on larger problems and larger computing platforms are needed to determine whether the performance of the *2-level parallel* method would live up to expectations. Analysis and experiments show that when exploiting the *2-levels* of parallelism the algorithm runs closed to the aggregate performance of the available computing resources.

For those applications the parallel sparse direct solver and the *2-level parallel* method remain the methods of choice for $2D$ multisource acoustic simulations considering that the memory requirement is tractable with currently available computers. For large $3D$ problems, the hybrid approach (*1-level parallel* or *2-level parallel*) are a possible alternative to the direct solvers that exhibit an unaffordable increase of computing time and memory requirement. In the context of multisource simulations (i.e., multiple right-hand sides) some extra effort should be devoted to take advantage of this feature such as block-Krylov solvers or numerical technique to recycle some spectral information between the solution of the various right-hand sides.

# subdomains	**50**			**81**		**96**	
# processors per subdomain	*1-level parallel*	*2-levels parallel*		*1-level parallel*	*2-levels parallel*	*1-level parallel*	*2-levels parallel*
	1	2	4	1	2	1	2
	Total solution time						
$7\ Hz$	1178.9	854.5	419.8	557.7	431.4	398.9	299.3
	Time for one RHS						
$7\ Hz$	67.8	45.1	35.5	77.4	57.2	61.1	45.1
	Time in the iterative loop						
$7\ Hz$	64.4	39.1	28.1	73.6	53.7	57.0	38.2
	# iteration						
$7\ Hz$	81			109		119	
	Preconditioner setup time						
$7\ Hz$	497.1	262.4	135.3	256.3	169.2	153.8	81.2
	Initialization time						
$7\ Hz$	614	547	249	224	205	184	173

Table 8.10: Detailed parallel performance of the *2-level parallel* method for the $3D$ Overthrust SEG/EAGE test case when the number of subdomains is varied from 50 to 96.

IV

Part IV

Further performance study and applications

Chapter 9

Conclusion and future work

The main topic of this research work was the study of a numerical technique that had attractive features for an efficient solution of large scale linear systems on large massively parallel platforms. In this respect we have investigated several algebraic preconditioning techniques, discussed their numerical behaviours, their parallel implementations and scalabilities. Finally, we have compared their performances on a set of $3D$ grand challenge problems. The algebraic additive Schwarz preconditioner defined by [25] for $2D$ problems was the starting point for our study in the context of non-overlapping domain decomposition techniques.

We have defined different variants based either on mixed arithmetics or on sparse approximations. We have investigated and analyzed their numerical behaviours. We have evaluated their efficiency and accuracy compared to the dense 64-bit variant. The results show that the sparse variant enables us to get reasonable numerical behaviour and permits the saving of a significant amount of memory. On all our experiments this variant is the most efficient and reduces the solution time and the memory space. The mixed precision approach appears very promising in the context of multi-core heterogeneous massively parallel computers, where some devices (such a the graphic cards) only operate in 32-bit arithmetic. Several questions are still open, and some works deserve to be undertaken to validate the approach in this computing context as well as theoretical developments to assess their numerical validity.

In Chapter 5 and Chapter 6, our analysis was focused on the numerical and the parallel scalability of the preconditioners on a set of academic $3D$ symmetric and unsymmetric model problems. We have studied the numerical behaviour of the proposed preconditioners for the solution of heterogeneous anisotropic diffusion problems with and without a convection term. We have observed reasonably good parallel performance and numerical scalability on massively parallel platforms. All the preconditioner variants were able to exploit a large number of processors with an acceptable efficiency. For the $3D$ problems the convergence of all the local preconditioners depends slightly on the number of subdomains. For up-to a thousand of processors/subdomains the use of a coarse space component for 3D elliptic problems appeared less crucial, although still beneficial, than for 2D problems.

We have also applied these hybrid iterative/direct methods for solving unsymmetric and indefinite problems such as those arising in some structural mechanics and seismic modelling applications. For that purpose, we have investigated in Chapter 7 and Chapter 8, the use of our parallel approaches in the context of these real life applications. Our purpose was to evaluate the robustness and the performance of our preconditioners for the solution of the challenging linear systems that are often solved using direct solvers in these simulations.

Chapter 7 was devoted to the engineering area of structural mechanics simulations where very large problems have to be solved. For those simulations, the meshes involved in the discretization are composed by a large number of finite elements. The efficient parallel solution requires a good balanced partitioning of the unstructured meshes to ensure the efficiency of the solution technique

as well as its parallel performance. We have illustrated the sensitivity of our preconditioner to the partitioning of the mesh. Other partitioning strategies would deserve to be studied and investigated. We have presented several results, that show that the preconditioners can achieve high performance. The attractive feature is that both sparse and mixed preconditioners are a source of gain even for these difficult problems. More work would deserve to be invested on a sophisticated dropping strategies, as well as on the automatic tuning of the dropping parameter. Further developments are required, such as the integration of the hybrid method in the nonlinear Newton solver in structural analysis. We should study the effect of the linear solver accuracy when embedded within the nonlinear scheme. In order to improve the performance for large-scale simulations and to run close to the aggregate resources peak floating-point operation rate, we have considered the introduction of a *2-levels* parallelism strategy. Numerical tests confirm the good properties of the hybrid *2-levels* parallel method; a comparison with the direct method shows that, from the point of view of both numerical and computational costs, the *2-levels* parallel method can be of interest for a wide range of applications.

In Chapter 8, we have further considered the parallel performance of the hybrid method in the context of a $2D/3D$ seismic application. In this framework, the traditional method for solving these systems relies on sparse direct solvers because multiple right-hand sides have to be considered. However for large $3D$ problems the memory requirement and the computational complexity become unaffordable. We have discussed how the algebraic additive Schwarz preconditioner can be applied in this context. We have used a variant of the preconditioner based on the introduction of a complex perturbation to the Laplace operator [40], resulting in a shifted additive Schwarz preconditioner. For $2D$ problems, we have observed that the numerical scalability of the preconditioner can significantly deteriorate when the number of subdomains increases. In this case, the *2-levels* parallel algorithm can be of great interest. We have thus investigated the idea of using a small number of subdomains while increasing the number of processors per subdomain as a possible remedy to this weakness. The significant benefit in 2D does not translate as clearly for 3D problems where only moderate improvements have been observed. In the context of multi-source seismic simulations (i.e., multiple right-hand sides) some extra effort should be devoted to take advantage of this feature such as block-Krylov solvers or numerical techniques to recycle some spectral information between the solution of the various right-hand sides.

The development of efficient and reliable parallel algebraic hybrid solvers is a key for successful applications of scientific simulations for the solution of many challenging large-scale problems. We have attempted to highlight some of the studies and developments that have taken place in the course of the three years of the thesis. There are numerous further important problems and ideas that we have not been addressed. This research area is very active and productive; there is still room for the development of new general-purpose parallel black-box hybrid methods based on algebraic approaches. This class of algebraic preconditioners is very attractive for parallel computing. We intend to extend the work presented here for the solution of general linear systems, where the techniques has some natural counterparts. It mainly consists in extending the ideas and apply them to the graphs of general sparse matrices in order to identify the blocks and the interface between the blocks. The basic idea is to split the matrix entries into different blocks as shown in Figure 9.1.

A preliminary Matlab prototype has been developed to validate the basic concepts. For the sake of preliminary comparisons a block-Jacobi preconditioner has also been implemented. The numerical experiments have been conduced on sets of general sparse matrices from Matrix Market repository (http://math.nist.gov/MatrixMarket/) and Tim Davis collection (http://www.cise.ufl.edu/research/sparse/matrices/). We present in Table 9.1 the number of iterations required by right preconditioned GMRES. For each matrix, we consider decompositions into 8, 16, 32, 64 and 96 blocks. Those preliminary experiments are encouraging and confirm that the additive Schwarz preconditioner for the Schur complement initially introduced for the solution of linear systems arisen from PDE discretization can be extended to solve general sparse linear systems. For those matrices, it can be seen that the number of iterations increases moderately

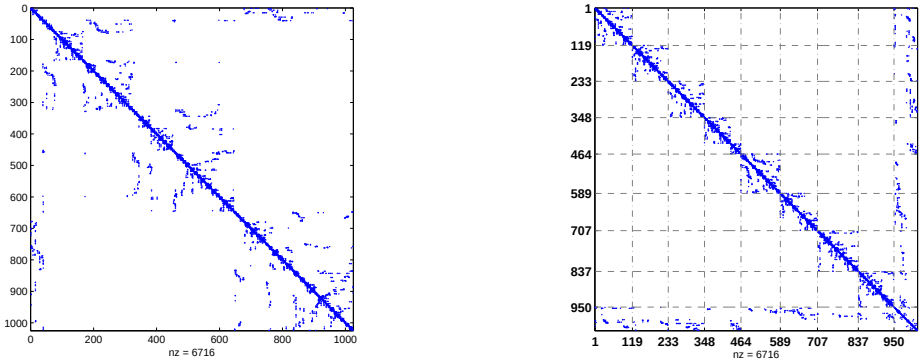


Figure 9.1: Partitioning a general matrix on 8 blocks.

when the number of blocks is varied. We intend to continue these investigations with the goal of designing and implementing an efficient black-box parallel solver package suited for large massively computing platforms.

Matrix			Preconditioner	# blocks				
name	size	nnz		8	16	32	64	96
bcsstk18	11,948	149,090	Block Jacobi	88	135	171	192	208
			Additive Schwarz	26	42	60	83	86
bcsstk37	25,503	1,140,977	Block Jacobi	140	200	325	547	560
			Additive Schwarz	24	53	79	137	155
nasa4704	4,704	104,756	Block Jacobi	125	183	344	417	547
			Additive Schwarz	25	43	47	88	114
nasasrb	54,870	1,366,097	Block Jacobi	72	189	649	885	-
			Additive Schwarz	42	97	148	165	251
ex11	16,614	1,096,948	Block Jacobi	266	656	931	-	-
			Additive Schwarz	17	17	35	43	57

Table 9.1: Number of preconditioned GMRES iterations when the number of blocks is varied. "-" means that no convergence was observed after 1000 iterations.

Finally, hybrid techniques include other ways to combine and mix iterative and direct approaches. We start another study of hybrid solver in the context of $3D$ electromagnetic simulations in collaboration with the INRIA NACHOS project. Of particular interest is the study of the interaction of electromagnetic waves with humans or, more precisely, living tissues [33]. Both the heterogeneity and the complex geometrical features of the underlying media motivate the use of the hybrid methods working on non-uniform meshes. The objective of our contribution is to improve the numerical behaviour of the hybrid method by developping new algebraic preconditioners.

Collaboration acknowledgments

I gratefully acknowledge the valuable assistance of many people of the MUMPS team. Especially, I wish to thank Patrick Amestoy and Jean-Yves L'Excellent for their support and help, their reactivity and availability as well as their usefull advices.

I wish to express my sincere gratitude to the SAMCEF project, especially to Stéphane Pralet, who provided me with support on the structural mechanics problems and software. I am gratefull to Stéphane for his kind development of special functionalities that enable me to easily use the SAMCEF code and for his advices and numerous suggestions.

It is my great pleasure for me to thank all the members of the SEISCOPE Consortium with whom I had fruitful discussions. It enables me to get in touch with the seismic applications; many thanks to Romain Brossier, Stéphane Operto, Florent Sourbier and Jean Virieux.

My sincere thanks go to the INRIA-NACHOS project. I must thank Stéphane Lanteri who opened me the door of an enriching collaboration, who introduced me to the computational electromagnetics field and who gave me constructive advices.

Bibliography

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

[2] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. `MUMPS`: a general purpose distributed memory sparse solver. In A. H. Gebremedhin, F. Manne, R. Moe, and T. Sørevik, editors, *Proceedings of PARA2000, the Fifth International Workshop on Applied Parallel Computing, Bergen, June 18-21*, pages 122–131. Springer-Verlag, 2000. Lecture Notes in Computer Science 1947.

[3] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[4] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK's user's guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[5] M. Arioli. Private communication, 2007. During Matrix Analysis and Applications CIRM Luminy - October 15-19, 2007 in honor of Gérard Meurant for his 60th birthday.

[6] M. Arioli, I.S. Duff, S. Gratton, and S. Pralet. Note on GMRES preconditionned by a perturbed LDLt decomposition with static pivoting. *SIAM Journal on Scientific Computing*, 25(5):2024–2044, 2007.

[7] W. E. Arnoldi. The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Quart. Appl. Math.*, 9(1):17–29, 1951.

[8] C. Ashcraft. The fan-both family of column-based distributed Cholesky factorisation algorithm. In A. George, J.R. Gilbert, and J.W.H Liu, editors, *Graph Theory and Sparse Matrix Computations*, pages 159–190. Springer-Verlag NY, 1993.

[9] O. Axelsson. *Iterative Solution Methods*. Cambridge University Press, Cambridge, 1994.

[10] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. Curfman McInnes, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

[11] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, PA, second edition, 1994.

[12] H. Ben-Hadj-Ali, S. Operto, J. Virieux, and F. Sourbier. Three-dimensional frequency-domain full waveform tomography. *Geophysical Research Abstracts*, 10, 2008.

[13] M. Benzi, C. D. Meyer, and M. Tuma. A sparse approximate inverse preconditioner for the conjugate gradient method. *SIAM Journal on Scientific Computing*, 17:1135–1149, 1996.

[14] M. Benzi and M. Tuma. A sparse approximate inverse preconditioner for nonsymmetric linear systems. *SIAM Journal on Scientific Computing*, 19:968–994, 1998.

[15] Jean-Pierre Berenger. A perfectly matched layer for the absorption of electromagnetic waves. *J. Comput. Phys.*, 114(2):185–200, 1994.

[16] P. Bjørstad and O. Widlund. Iterative methods for the solution of elliptic problems on regions partitioned into substructures. *SIAM Journal on Numerical Analysis*, 23(6):1097–1120, 1986.

[17] P. E. Bjørstad and O. B. Widlund. Iterative methods for the solution of elliptic problems on regions partitioned into substructures. *SIAM J. Numer. Anal.*, 23(6):1093–1120, 1986.

[18] 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*. SIAM Press, 1997.

[19] J.-F. Bourgat, R. Glowinski, P. Le Tallec, and M. Vidrascu. Variational formulation and algorithm for trace operator in domain decomposition calculations. In Tony Chan, Roland Glowinski, Jacques Périaux, and Olof Widlund, editors, *Domain Decomposition Methods*, pages 3–16, Philadelphia, PA, 1989. SIAM.

[20] J.H. Bramble, J.E. Pasciak, and A.H. Schatz. The construction of preconditioners for elliptic problems by substructuring I. *Math. Comp.*, 47(175):103–134, 1986.

[21] A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek, and S. Tomov. Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy. *ACM Transactions on Mathematical Software*, 34(4), 2008.

[22] X.-C. Cai and Y. Saad. Overlapping domain decomposition algorithms for general sparse matrices. *Numerical Linear Algebra with Applications*, 3:221–237, 1996.

[23] X.-C. Cai and M. Sarkis. A restricted additive Schwarz preconditioner for general sparse linear systems. *SIAM Journal on Scientific Computing*, 21:792–797, 1999.

[24] B. Carpentieri, I. S. Duff, L. Giraud, and G. Sylvand. Combining fast multipole techniques and an approximate inverse preconditioner for large electromagnetism calculations. *SIAM Journal on Scientific Computing*, 27(3):774–792, 2005.

[25] L. M. Carvalho, L. Giraud, and G. Meurant. Local preconditioners for two-level non-overlapping domain decomposition methods. *Numerical Linear Algebra with Applications*, 8(4):207–227, 2001.

[26] L. M. Carvalho, L. Giraud, and P. Le Tallec. Algebraic two-level preconditioners for the schur complement method. *SIAM Journal on Scientific Computing*, 22(6):1987–2005, 2001.

[27] F. Chaitin-Chatelin and V. Frayssé. *Lectures on Finite Precision Computations*. SIAM, Philadelphia, 1996.

[28] E. Chow. A priori sparsity patterns for parallel sparse approximate inverse preconditioners. *SIAM Journal on Scientific Computing*, 21(5):1804–1822, 2000.

[29] Y.-H. De Roeck and P. Le Tallec. Analysis and test of a local domain decomposition preconditioner. In Roland Glowinski, Yuri Kuznetsov, Gérard Meurant, Jacques Périaux, and Olof Widlund, editors, *Fourth International Symposium on Domain Decomposition Methods for Partial Differential Equations*, pages 112–128. SIAM, Philadelphia, PA, 1991.

[30] J. Demmel, Y. Hida, W. Kahan, S. X. Li, S. Mukherjee, and E. J. Riedy. Error bounds from extra precise iterative refinement. Technical Report UCB/CSD-04-1344, LBNL-56965, University of California in Berkeley, 2006. Short version appeared in ACM Trans. Math. Software, vol. 32, no. 2, pp 325-351, June 2006.

[31] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, S. X. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755, 1999.

[32] J. W. Demmel, J. R. Gilbert, and X. S. Li. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. *SIAM Journal on Matrix Analysis and Applications*, 20(4):915–952, October 1999.

[33] V.a Dolean, S. Lanteri, and R. Perrussel. A domain decomposition method for solving the three-dimensional time-harmonic maxwell equations discretized by discontinuous galerkin methods. *J. Comput. Phys.*, 227(3):2044–2072, 2008.

[34] J. Drkošová, M. Rozložník, Z. Strakoš, and A. Greenbaum. Numerical stability of the GMRES method. *BIT*, 35:309–330, 1995.

[35] M. Dryja, B. F. Smith, and O. B. Widlund. Schwarz analysis of iterative substructuring algorithms for elliptic problems in three dimensions. *SIAM J. Numer. Anal.*, 31(6):1662–1694, 1993.

[36] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, London, 1986.

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

[38] I. S. Duff and J. K. Reid. The multifrontal solution of unsymmetric sets of linear systems. *SIAM Journal on Scientific and Statistical Computing*, 5:633–641, 1984.

[39] J. Erhel, K. Burrage, and B. Pohl. Restarted GMRES preconditioned by deflation. *J. Comput. Appl. Math.*, 69:303–318, 1996.

[40] Y. A. Erlangga, C. Vuik, and C. W. Oosterlee. On a class of preconditioners for solving the helmholtz equation. *Appl. Numer. Math.*, 50(3-4):409–425, 2004.

[41] C. Farhat and F.-X. Roux. A method of finite element tearing and interconnecting and its parallel solution algorithm. *Int. J. Numer. Meth. Engng.*, 32:1205–1227, 1991.

[42] V. Frayssé and L. Giraud. A set of conjugate gradient routines for real and complex arithmetics. Technical Report TR/PA/00/47, CERFACS, Toulouse, France, 2000. Public domain software availabe on www.cerfacs.fr/algor/Softs.

[43] V. Frayssé, L. Giraud, and S. Gratton. A set of flexible GMRES routines for real and complex arithmetics on high performance computers. Technical Report TR/PA/06/09, CERFACS, Toulouse, France, 2006. This report supersedes TR/PA/98/07.

[44] V. Frayssé, L. Giraud, S. Gratton, and J. Langou. Algorithm 842: A set of GMRES routines for real and complex arithmetics on high performance computers. *ACM Transactions on Mathematical Software*, 31(2):228–238, 2005. Preliminary version available as CERFACS TR/PA/03/3, Public domain software availabe on www.cerfacs.fr/algor/Softs.

[45] A. C. Gale, R. C. Almeida, S. M. C. Malta, and A. F. D. Loula. Finite element analysis of convection dominated reaction-diffusion problems. *Appl. Numer. Math.*, 48(2):205–222, 2004.

[46] L. Giraud, S. Gratton, , and E. Martin. Incremental spectral preconditioners for sequences of linear systems. *Applied Numerical Mathematics*, 57:1164–1180, 2007.

[47] L. Giraud, S. Gratton, and J. Langou. Convergence in backward error of relaxed GMRES. *SIAM Journal on Scientific Computing*, 29(2):710–728, 2007.

[48] L. Giraud and A. Haidar. Parallel algebraic hybrid solvers for large 3d convection-diffusion problems. *to appear Numerical Algorithms*.

[49] L. Giraud, A. Haidar, and S. Pralet. Using multiple levels of parallelism to enhance the performance of hybrid linear solvers. *Submitted to Parallel Computing*.

[50] L. Giraud, A. Haidar, and L. T. Watson. Mixed-precision preconditioners in parallel domain decomposition solvers. *Springer: Lectures notes computational science and engineering*, 60:357–364, 2008.

[51] L. Giraud, A. Haidar, and L. T. Watson. Parallel scalability study of hybrid preconditioners in three dimensions. *Parallel Computing*, 34:363–379, 2008.

[52] L. Giraud, A. Marrocco, and J.-C. Rioual. Iterative versus direct parallel substructuring methods in semiconductor device modelling. *Numerical Linear Algebra with Applications*, 12(1):33–53, 2005.

[53] L. Giraud and R. Tuminaro. Algebraic domain decomposition preconditioners. In F. Magoules, editor, *Mesh partitioning techniques and domain decomposition methods*, pages 187–216. Saxe-Coburg Publications, 2007.

[54] G. H. Golub and C. Van Loan. *Matrix computations*. Johns Hopkins University Press, 1996. Third edition.

[55] A. Greenbaum. *Iterative methods for solving linear systems*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1997.

[56] W. D. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of MPI message passing interface standard. *Parallel Computing*, 22(6), 1996.

[57] M. Grote and T. Huckle. Parallel preconditionings with sparse approximate inverses. *SIAM Journal on Scientific Computing*, 18:838–853, 1997.

[58] M. Heroux. AztecOO user guide. Technical Report SAND2004-3796, Sandia National Laboratories, Albuquerque, NM, 87185, 2004.

[59] M. Heroux, R. Bartlett, V. Howle, R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, A. Williams, and K. Stanley. An overview of the Trilinos project. *ACM Transactions on Mathematical Software*, 31(3):397–423, September 2005.

[60] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear system. *J. Res. Nat. Bur. Stds.*, B49:409–436, 1952.

[61] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[62] G. Karypis and V. Kumar. METIS, unstructured graph partitioning and sparse matrix ordering system. version 2.0. Technical report, University of Minnesota, Department of Computer Science, Minneapolis, MN 55455, August 1995.

[63] L. Yu Kolotilina, A. Yu. Yeremin, and A. A. Nikishin. Factorized sparse approximate inverse preconditionings. III: Iterative construction of preconditioners. *Journal of Mathematical Sciences*, 101:3237–3254, 2000. Originally published in Russian in *Zap. Nauchn. Semin. POMI*, 248:17-48, 1998.

[64] J. Kurzak and J. Dongarra. Implementation of the mixed-precision high performance LINPACK benchmark on the CELL processor. Technical Report LAPACK Working Note #177 UT-CS-06-580, University of Tennessee Computer Science, September 2006.

[65] C. Lanczos. Solution od systems of linear equations by minimized iterations. *J. Res. Nat. Bur. Stand.*, 49:33–53, 1952.

[66] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra. Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy. Technical Report LAPACK Working Note #175 UT-CS-06-574, University of Tennessee Computer Science, April 2006.

[67] P. Le Tallec. *Domain decomposition methods in computational mechanics*, volume 1 of *Computational Mechanics Advances*, pages 121–220. North-Holland, 1994.

[68] Z. Li, Y. Saad, and M. Sosonkina. pARMS: A parallel version of the algebraic recursive multilevel solver. Technical Report Technical Report UMSI-2001-100, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, 2001.

[69] J. Mandel. Balancing domain decomposition. *Communications in Numerical Methods in Engineering*, 9:233–241, 1993.

[70] J. Mandel and M. Brezina. Balancing domain decomposition for problems with large jumps in coefficients. *Mathematics of Computation*, 65:1387–1401, 1996.

[71] J. Mandel and R. Tezaur. Convergence of substructuring method with Lagrange multipliers. *Numer. Math.*, 73:473–487, 1996.

[72] G. Meurant. *The Lanczos and conjugate gradient algorithms: from theory to finite precision computations*. Software, Environments, and Tools 19. SIAM, Philadelphia, PA, USA, 2006.

[73] R. B. Morgan. Implicitely restarted GMRES and Arnoldi methods for nonsymme tric systems of equations. *SIAM Journal on Matrix Analysis and Applications*, 21(4):1112–1135.

[74] R. B. Morgan. A restarted GMRES method augmented with eigenvectors. *SIAM Journal on Matrix Analysis and Applications*, 16:1154–1171, 1995.

[75] K. W. Morton. *Numerical Solution of Convection-Diffusion Problems*. Chapman & Hall, 1996.

[76] S. Operto, J. Virieux, P. Amestoy, J.Y. L'Excellent, Luc Giraud, and H. Ben-Hadj-Ali. 3D finite-difference frequency-domain modeling of visco-acoustic wave propagation using a massively parallel direct solver: A feasibility study. *Geophysics*, 72:195–211, 2007.

[77] C. Paige, M. Rozložník, and Z. Strakoš. Modified Gram-Schmidt (MGS), least-squares, and backward stability of MGS-GMRES. *SIAM Journal on Matrix Analysis and Applications*, 28(1):264–284, 2006.

[78] C. C. Paige and M. A. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM Journal on Numerical Analysis*, 12:617–629, 1975.

[79] M. L. Parks, E. de Sturler, G. Mackey, D. D. Jhonson, and S. Maiti. Recycling Krylov subspaces for sequences of linear systems. Technical Report UIUCDCS-R-2004-2421 (CS), University of Illinois at Urbana-Champaign, 2004.

[80] R. G. Pratt. Velocity models from frequency-domain waveform tomography: past, present and future. In *Expanded Abstracts*. Eur. Ass. Expl. Geophys., 2004.

[81] G. Radicati and Y. Robert. Parallel conjugate gradient-like algorithms for solving nonsymmetric linear systems on a vector multiprocessor. *Parallel Computing*, 11:223–239, 1989.

[82] C. Ravaut, S. Operto, L. Improta, J. Virieux, A. Herrero, and P. Dell'Aversana. Multi-scale imaging of complex structures from multifold wide-aperture seismic data by frequency-domain full-waveform tomography: application to a thrust belt. *Geophysical Journal International*, 159:1032–1056, 2004.

[83] H.-G. Roos, M. Stynes, and L. Tobiska. *Numerical methods for singularly perturbed differential equations*, volume 24 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, 1996. Convection-diffusion and flow problems.

[84] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Comput.*, 14:461–469, 1993.

[85] Y. Saad. ILUT: A dual threshold incomplete LU factorization. *Numerical Linear Algebra with Applications*, 1:387–402, 1994.

[86] Y. Saad. *Iterative Methods for Sparse Linear Systems.* SIAM, Philadelphia, 2003. Second edition.

[87] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comp.*, 7:856–869, 1986.

[88] Y. Saad and H. A. van der Vorst. Iterative solution of linear systems in the 20-th century. Tech. Rep. UMSI-99-152, University of Minnesota, 1999.

[89] J. Schulze. Towards a tighter coupling of bottom-up and top-down sparse matrix ordering methods. *BIT Numerical Mathematics*, 41(4):800–841, 2001.

[90] H. A. Schwarz. Über eine grenzübergang durch alternirendes verfahren. *Gesammelete Mathematische Abhandlungen, Springer-Verlag*, 2:133–143, 1890. First published in Vierteljahrsschrift der Naturforschenden Gesellschaft in Zürich, vol.15, pp. 272-286, 1870.

[91] V. Shanthi and N. Ramanujam. Asymptotic numerical methods for singularly perturbed fourth order ordinary differential equations of convection-diffusion type. *Appl. Math. Comput.*, 133(2-3):559–579, 2002.

[92] V. Simoncini and E. Gallopoulos. An iterative method for nonsymmetric systems with multiple right-hand sides. *SIAM Journal on Scientific Computing*, 16:917–933, 1995.

[93] B. F. Smith. *Domain Decomposition Algorithms for the Partial Differential Equations of Linear Elasticity.* PhD thesis, Courant Institute of Mathematical Sciences, September 1990. Tech. Rep. 517, Department of Computer Science, Courant Institute.

[94] B. F. Smith, P. Bjørstad, and W. Gropp. *Domain Decomposition, Parallel Multilevel Methods for Elliptic Partial Differential Equations.* Cambridge University Press, New York, 1st edition, 1996.

[95] F. Sourbier, A. Haidar, L. Giraud, R. Brossier, S. Operto, and J. Virieux. Combining direct and iterative solvers for improving efficiency of solving wave equations when considering multi-sources problems. In *EAGE European Association of Geophysientists and Engineer*, 2008.

[96] F. Sourbier, S. Operto, and J. Virieux. Acoustic full-waveform inversion in the frequency domain. In W. Marzochi and A. Zollo, editors, *In Conception, verification and application of innovative techniques to study active volcanoes*, pages 295–307, 2008.

[97] M. Stynes. Steady-state convection-diffusion problems. *Acta Numer.*, 14:445–508, 2005.

[98] P. Le Tallec, Y.-H. De Roeck, and M. Vidrascu. Domain-decomposition methods for large linearly elliptic three dimensional problems. *J. of Computational and Applied Mathematics*, 34:93–117, 1991.

[99] R. S. Tuminaro, M. Heroux, S. Hutchinson, and J. Shadid. Official Aztec user's guide: Version 2.1. Technical Report Sand99-8801J, Sandia National Laboratories, Albuquerque, NM, 87185, Nov 1999.

[100] T. Vanorio, J. Virieux, P. Capuano, and G. Russo. Three-dimensional seismic tomography from p wave and s wave microearthquake travel times and rock physics characterization of the campi flegrei caldera. *J. Geophys. Res*, 110, 2005.

[101] J. H. Wilkinson. *Rounding Errors in Algebraic Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, 1963.