Algebraic domain decomposition methods for hybrid (iterative/direct) solvers

Louis Poirel

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Louis POIREL**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : MATHÉMATIQUES APPLIQUÉES

Méthodes de décomposition de domaine algébriques pour solveurs hybrides (direct/itératif)

Date de soutenance : 28 Novembre 2018

Devant la commission d'examen composée de :

Emmanuel Agullo ..	Chargé de recherche, Inria	Co-encadrant
Bénédicte Cuenot ...	Senior researcher, CERFACS	Examinateur
Martin J. Gander ...	Professeur, Université de Genève	Rapporteur
Luc Giraud	Directeur de recherche, Inria	Directeur de thèse
Arnaud Legrand ...	Directeur de recherche CNRS, IMAG Grenoble	Président
Michael A. Heroux .	Senior Scientist, Sandia National Laboratories	Rapporteur
Patrick Le Tallec ..	Professeur, École polytechnique	Rapporteur
François-Xavier Roux	Professeur, Université Pierre et Marie Curie ...	Examinateur

– 2018 –

Résumé

La résolution de grands systèmes linéaires est une des étapes les plus consommatrices en temps des simulation numérique. Des solveurs linéaires haute performance ont été développés dans un contexte algébrique (à partir du système $\mathcal{K}u = f$) ; d'autres méthodes, dites de décomposition de domaine, offrent d'excellentes performances en exploitant l'information au niveau de l'équation aux dérivées partielles sous-jacente au système linéaire. Dans cette thèse, on tente de concilier ces deux approches: une analyse de convergence des méthodes de Schwarz abstraites à deux niveaux conduit à la définition de nouveaux préconditionneurs robustes pour les problèmes symétriques définis positifs basés sur une généralisation algébrique de la méthode GenEO. Ces préconditionneurs robustes ne nécessitent que la donnée de la matrice $\mathcal{K}$ comme une somme de matrices locales $\mathcal{K}_i$ symmétriques semi-definies positives. Un préconditionneur robuste suivant cette méthode a été implémenté dans un solveur hybride parallèle distribué et testé sur des cas applicatifs. Une nouvelle boîte à outils de décomposition de domaine a aussi été développée en python pour faciliter le développement de nouveaux solveurs par décomposition de domaines basés sur des solveurs haute performance. Le code de ce module nommé `ddmpy` est inclus dans le présent document par programmation lettrée dans une approche de science reproductible.

Title

Algebraic Domain Decomposition Methods for Hybrid (direct/iterative) Solvers

Abstract

The solution of large linear problems is one of the most time consuming kernels in many numerical simulations. On the one hand, the computational linear algebra community has developed several high performance linear solvers that only require algebraic information (the matrix $\mathcal{K}$ and its associated right-hand side f) to compute the solution x such that $\mathcal{K}x = f$. On the other hand, the Domain Decomposition (DD) community has developed many efficient and robust methods in the last decades, that take into account the underlying partial differential equation and the geometry to accelerate the solution of such problems. In this thesis, both approaches are combined: an analysis of coarse correction for abstract Schwarz (aS) DD solvers is proposed, leading to a new methodology for building robust preconditioners for Symmetric Positive Definite (SPD) matrices based on an algebraic generalization of the Generalized Eigenvalue in the Overlap (GenEO) approach. The only requirement is that the SPD matrix $\mathcal{K}$ is provided as a sum of local symmetric positive semi-definite (SPSD) matrices $\mathcal{K}_i$. A robust preconditioner following this methodology was developed for a sparse hybrid parallel distributed solver and applied on several test cases. A new algebraic parallel DD toolbox in python was developed to facilitate the development of new DD solvers relying on state-of-the-art high performance solvers. This `ddmpy` module is exposed in this document using a literate programming approach for reproducible science.

Keywords

domain decomposition methods, parallel hybrid (direct/iterative) solver, coarse space, literate programming, reproducible science

Mots-clés

méthodes de décomposition de domaine, solveur parallèle hybride (direct/iteratif), espace grossier, programmation lettrée, science reproductible

Laboratoire d'accueil

Équipe-projet HiePACS
Inria Bordeaux – Sud-Ouest
200 avenue de la vieille tour
33405 Talence Cedex France

Contents

Remerciements

Un grand merci à toutes les personnes qui m'ont accompagné durant cette thèse ! Merci Luc et Manu pour votre encadrement et votre amitié : je ne connaissais rien à l'algèbre linéaire ni au calcul haute performance avant d'arriver chez vous (merci POM), et vous avez su me guider et me faire aimer ces sujets. Au-delà de tout ce que vous m'avez transmis scientifiquement, je tiens à vous remercier pour vos conseils et votre aide ces quatre dernières années, y compris dans les vraies difficultés.

Merci à mes autres coauteurs, en particulier Gilles et Matthieu ; merci aussi pour les cours de swing !

Un merci tout particulier, Chrystel, pour ton aide discrète et efficace (même quand je me suis retrouvé bloqué de l'autre côté du cercle polaire) !

Merci à toute l'équipe HiePACS pour les soirées vin et fromage, les galettes des rois, les baby-foot... un petit mot pour dire mon admiration toute particulière pour des génies (incompris) croisés pendant cette thèse : Grégoire, JM et Maria.

Merci aussi aux colocs de Bègles : feux de cheminée virtuels, «j'habite chez Mamère», planquages botaniques... je ne me suis pas ennuyé avec vous !

Merci aux rapporteurs anonymes des articles que nous avons soumis, et aux trois rapporteurs de ce manuscrit, qui ont contribué à l'améliorer. Merci Nicole pour nos échanges qui m'ont aidé à mieux comprendre GenEO.

Je remercie les professeurs de mathématiques qui m'ont fait découvrir et aimer cette discipline : M. Huynh, Mme Fisset, M. Maurice, Mme Benhamou, M. Tosel.

Un très grand merci à mes parents qui m'ont transmis (entre autres) le goût d'apprendre et de découvrir et que j'embrasse avec mes frères et sœurs, dont le Dr. M. Poirel.

Pendant ces quatre années, d'autres priorités sont venues s'ajouter à la thèse puisque je me suis marié et suis devenu papa ; la rédaction de ce manuscrit n'aurait pas été possible sans les patients encouragements de mon épouse Verena qui a su me supporter pendant la période de rédaction tout en gérant les petits soucis du quotidien, les tracasseries administratives et l'aventure de la grossesse : merci pour tout ! Notre fille Helena Marie est restée sage pendant la soutenance, merci à elle ! Cette thèse lui est dédiée.

Résumé étendu

La simulation numérique de phénomènes physiques complexes requiert la résolution de systèmes linéaires de grande taille: le comportement ou l'évolution d'un phénomène sont souvent modélisés sous forme d'un système d'Equations aux Dérivées Partielles (EDP) faisant intervenir des conditions limites. Des méthodes de discrétisation, comme la méthode des eléments finis par exemple, permettent de traduire l'EDP en un système linéaire de la forme $\mathcal{K}u = f$, dont la solution discrète approche d'autant mieux la solution de l'EDP que la taille du système augmente. Au cours des dernières décennies, plusieurs algorithmes et programmes (appelés solveurs) ont été développés pour la résolution informatique des grands systèmes linéaires sur des machines parallèles multicœurs.

Solveurs hybrides (direct/itératif)

Entre des méthodes directes, qui éliminent les inconnues les unes après les autres (Gauss, 1811; Amestoy et al., 2001; Duff et al., 2017; Hénon et al., 2002) et les méthodes itératives, qui améliorent une solution approchée par corrections successives (Hestenes and Stiefel, 1952; Saad and Schultz, 1986; Saad, 2003), des méthodes dites hybrides conjuguent les avantages des deux classes précédentes (Giraud et al., 2008; Rajamanickam et al., 2012a; Yamazaki and Li, 2010). Aujourd'hui, les simulations de grande échelle se font dans un contexte parallèle, où plusieurs unités de calcul interagissent pour résoudre le système ; dans ce cadre, les solveurs hybrides permettent de combiner des solveurs directs et itératifs en les utilisant respectivement dans les régimes où ils sont le plus performant : le solveur direct localement pour éliminer les inconnues qui peuvent l'être sans perturber les autres unités de calcul, et le solveur itératif globalement pour assurer la cohérence des solutions locales. Cette approche est qualifiée d'algébrique, en ce qu'elle n'a besoin que de la donnée de $\mathcal{K}$ et f pour calculer la solution u.

Méthodes de décomposition de domaine

Cette approche hybride peut être rapprochée des Méthodes dites de Décomposition de Domaine (MDD) (Quarteroni and Valli, 1999; Dryja and Widlund, 1994; Mathew, 2008; Toselli and Widlund, 2006): une approche utilisée par Schwarz (1870) pour prouver l'existence d'une solution à certaines EDP a été étendue et adaptée pour le calcul parallèle (Lions, 1988, 1989, 1990). Souvent, les MDD sont introduites au niveau continu, avant discrétisation : le domaine sur lequel l'EDP est définie est découpé en N sous-

domaines, aux bords desquels des conditions aux limites d'interface sont introduites. C'est seulement ensuite que l'EDP est discrétisée, indépendamment dans chaque sous-domaine. Souvent, la résolution de problèmes locaux (dans les sous-domaines) intervient dans une formulation globale d'un problème assurant la continuité de la solution à l'interface entre les sous-domaines. Dans le Chapitre 2, des MDD classiques sont présentées, tant dans leur expression continue que dans une formulation la plus algébrique possible ; on montre que la plupart de ces méthodes peuvent s'utiliser dans un contexte algébrique à condition de fournir la matrice $\mathcal{K}$ sous la forme d'une somme de matrices locales $\mathcal{K} = \sum_{i=1}^{N} \mathcal{R}_{\Omega_i}^T \ \mathcal{K}_i \ \mathcal{R}_{\Omega_i}$, où $\mathcal{R}_{\Omega_i}$ représente la restriction canonique d'un vecteur global à un sous-domaine Ω_i. Il est possible d'éliminer les inconnues d'intérieur (celles qui n'ont pas d'interaction avec d'autres domaines) et de réécrire le système sous une forme réduite $\mathcal{S} u_\Gamma = \widetilde{f}_\Gamma$, où $\mathcal{S} = \sum_{i=1}^{N} \mathcal{R}_{\Gamma_i}^T \ \mathcal{S}_i \ \mathcal{R}_{\Gamma_i}$ est le complément de Schur de l'intérieur dans l'interface. Ce système ne fait intervenir que des quantités définies sur l'interface (ensemble des inconnues partagées par plus d'un sous-domaine).

Convergence des méthodes de Schwarz abstraites

La méthode originale de Schwarz (1870), parallélisée par Lions (1988) sous sa forme continue, a été adaptée au cas algébrique sous la forme d'un préconditionneur de Schwarz additif (Dryja and Widlund, 1987; Matsokin and Nepomnyaschikh, 1985) dans le cas où la matrice $\mathcal{K}$ est Symétrique Définie Positive (SDP). Avec d'autres préconditionneurs comme Neumann-Neumann, il peut être inclus dans une famille de préconditionneurs dits de Schwarz abstraits (Toselli and Widlund, 2006), dont la forme générale est $\mathcal{M}_{aS} = \sum_{i=1}^{N} \mathcal{R}_i^T \widehat{\mathcal{A}}_i^\dagger \mathcal{R}_i$, où $\mathcal{A}$ représente la matrice $\mathcal{K}$ ou $\mathcal{S}$, selon que l'on a éliminé ou pas les inconnues d'intérieur. Généralement, avec ces préconditionneurs, le conditionnement de la matrice, et donc le nombre d'itérations pour qu'une méthode itérative comme le gradient conjugué (Hestenes and Stiefel, 1952) converge, augmente avec le nombre de sous-domaines N. Dans le Chapitre 3, on borne le conditionnement des méthodes de Schwarz abstraites en comparant dans chaque sous-domaine le quotient de Rayleigh de la matrice locale $\widehat{\mathcal{A}}_i$ du préconditionneur $\mathcal{M}_{aS}$ avec celles des préconditionneurs de Schwarz additif et Neumann-Neumann. Suivant une méthodologie introduite par Spillane et al. (2014a); Spillane and Rixen (2013), nous proposons de construire un espace grossier adaptatif en résolvant les problèmes aux valeurs propres généralisés correspondant aux quotients de Rayleigh mentionnés ci-dessus. Avec cette correction, on prouve que le conditionnement et donc le nombre d'itérations restent borné. Dans le cas général, la correction est appliquée en utilisant une technique de déflation : la composante grossière de la solution est obtenue en résolvant le problème grossier par une méthode directe, et une projection est appliquée à chaque itération de la méthode itérative afin de ne pas venir polluer cette composante grossière. Dans le cas particulier du préconditionneur de Schwarz additif, l'espace grossier peut être traité, au niveau algébrique, comme les N sous-domaines. Ces résultats théoriques sont illustrés par des résultats numériques. D'autres travaux dans ce sens incluent (Efendiev et al., 2012; Galvis and Efendiev, 2010; Haferssas et al., 2017; Klawonn et al., 2016b, 2018; Nataf et al., 2011).

Solveur algébrique par décomposition de domaine en python: la boîte à outils `ddmpy`

Les différentes méthodes présentées ci-dessus peuvent être implémentées en combinant différents solveurs denses et creux, directs et itératifs, séquentiels ou distribués. Pour pouvoir développer efficacement de nouveaux solveurs hybrides et méthodes de décomposition de domaine algébriques, nous proposons dans le Chapitre 4 une boîte à outils de décomposition de domaine en python. Le language python est un language interprété de haut niveau. En comparaison à des langages compilés comme C, C++ ou Fortran, l'écriture d'un programme en python est souvent beaucoup plus rapide, au prix d'une performance réputée moins bonne à l'exécution. En analysant différentes implémentations de certaines opérations (addition de vecteurs, multiplication matricielle, calcul d'un complément de Schur et algorithme du gradient conjugué), on montre qu'il est possible d'écrire un code performant en python à condition d'utiliser des bibliothèques optimisées (souvent écrites dans des languages compilés) pour les opérations coûteuses et/ou répétées.

Sur ce constat, on a développé en python une boîte à outils pour construire des solveurs par décomposition de domaine au niveau algébrique. Ce module nommé `ddmpy` est présenté dans une approche de programmation lettrée (ou programmation littéraire) définie par Knuth (1984):

> *Au lieu de considérer que notre tâche principale est de dire à un ordinateur ce qu'il doit faire, appliquons-nous plutôt à expliquer à des êtres humains ce que nous voulons que l'ordinateur fasse.*

Ainsi, le code entier du module `ddmpy` est inclus dans ce chapitre ; les exemples inclus peuvent être exécutés de manière interactive en ouvrant la source `.org` (Dominik, 2010; Schulte and Davison, 2011) de ce document avec un éditeur compatible avec les fonctionnalités de ce format orgmode, comme emacs.

Une expérience de passage à l'échelle jusqu'à 3.072 cœurs de calcul est ensuite proposée : 16 solveurs par décomposition de domaines construits à partir d'éléments de `ddmpy` sont comparés. Tous les scripts nécessaires pour reproduire l'expérience et générer les figures sont inclus dans la source de ce document.

Conception parallèle d'une correction d'espace grossier

Dans le Chapitre 5, on présente et compare plusieurs stratégies parallèles pour la correction d'espace grossier présentée dans le Chapitre 3 : sans cette correction, les méthodes de Schwarz abstraites ne nécessitent que très peu de communications globales, ce qui les rend plus facile à implémenter efficacement ; c'est aussi ce qui explique leur mauvais comportement numérique pour résoudre des problèmes fortement distribués : l'information ne transite d'un sous-domaine à l'autre que par des communication de voisins à voisins ; pour une décomposition unidimensionnelle du domaine, il faut donc N itérations pour que l'information puisse transiter entre les deux sous-domaines extrêmes. L'idée générale des techniques de correction d'espace grossier consiste à transmettre de façon globale une

information grossière pour accélérer la résolution, par la résolution d'un problème global de taille réduite (avec, typiquement, seulement quelques inconnues par sous-domaine). La mise en œuvre d'une correction d'espace grossier réduit le nombre d'itérations mais rend chaque itération plus coûteuse puisqu'un problème grossier global doit y être résolu. Plusieurs stratégies, dont l'utilisation d'un solveur direct distribué sur l'ensemble des sous-domaines, ou sur un nombre réduit de sous-domaines, avec ou sans réplication du problème grossier, sont implémentées en Fortran 90 dans le solveur haute performance hybride MaPHyS[1] (Agullo et al., 2011, 2016b; Giraud et al., 2008). Ces différentes stratégies sont appliquées à la résolution de problèmes réguliers fortement hétérogènes avec une décomposition unidimensionnelle ou tridimensionnelle. Ce solveur est également utilisé dans deux codes applicatifs. Le premier est une code de simulation des voies respiratoires de l'homme réalisée avec le logiciel Alya[2] (Vázquez et al., 2016). Le second est un code de simulation de propulsion plasmique mis en œuvre dans le code AVIP (Joncquieres et al., 2018). Dans les deux cas, les systèmes linéaires résultent de la discrétisation d'une équation de type Poisson sur des maillages 3D non structurés. Ces expériences montrent que pour des problèmes de grande taille distribués sur un grand nombre de cœurs de calcul, il est avantageux de répartir les sous-domaines en plusieurs groupes de taille adéquate pour appeler un solveur creux distribué à la bonne granularité.

Perspectives ouvertes par ces travaux

Méthodes robustes pour problèmes non symétriques

Les travaux présentés dans les chapitres 3 et 5 sur les méthodes de Schwarz abstraites robustes ne concernent que les problèmes SDP : la preuve et l'implémentation reposent sur le fait que toutes les matrices ont des valeurs propres réelles et positives. La présentation générale des méthodes de décomposition de domaine dans le Chapitre 2, ainsi que l'implémentation proposée au Chapitre 4 ne sont pas concernés par cette restriction, et il serait intéressant d'intégrer dans `ddmpy` des méthodes robustes pour les problèmes non symmétriques. L'interface est présente dans `ddmpy` pour tester des méthodes de Robin optimisé algébriques, soit sous forme d'un préconditionneur Robin-Robin (Achdou and Nataf, 1997), soit en utilisant une formulation N-Lagrange (Nataf et al., 1995). Des travaux sont en cours dans l'équipe pour tester ces méthodes sur des problèmes d'électromagnétisme issus d'applications externes. Une autre piste serait d'utiliser un solveur itératif multi-préconditionné (Bridson and Greif, 2006; Greif et al., 2011; Spillane, 2016).

[1] See `https://gitlab.inria.fr/solverstack/maphys/`

[2] `https://www.bsc.es/research-development/research-areas/engineering-simulations/alya-high-performance-computational`

Implémentation en C++ d'une bibliothèque de décomposition de domaine

Une seconde perspective ouverte par cette thèse est l'écriture d'algorithme de décomposition de domaine dans un formalisme algébrique. Le language python utilisé pour la boîte à outils `ddmpy` présentée ci-dessus présente des limites en terme de parallélisation et quant à l'interfaçage avec des applications écrites en C, C++ ou Fortran. Nous pensons qu'il est possible, en utilisant un standard récent du langage C++, de remédier à ces difficultés sans perdre l'expressivité haut-niveau de python. Un projet en ce sens a été initié [3].

[3] `https://gitlab.inria.fr/solverstack/maphys/maphyspp`

Extended Summary

The numerical simulations of complex physical problem require the solution of large linear systems: the behavior and dynamics of such phenomena are often modeled by Partial Differential Equations (PDE) with some boundary conditions. Discretization techniques, such a the finite element methods, allow the translation of the PDE into a system of linear equations of the form $\mathcal{K}u = f$, whose solution becomes closer to the solution of the PDE as the size of the linear system increases. In the last decades, several algorithms and computer implementations, referred to as solvers, have been designed to allow the solution of large linear systems on parallel multicore computing platforms.

Hybrid (direct/iterative) solvers

Hybrid methods attempt to exploit the advantageous of both the direct methods, that eliminate the unknowns one after each other (Gauss, 1811; Amestoy et al., 2001; Duff et al., 2017; Hénon et al., 2002), and the iterative methods that improve an initial guess through successive updates (Hestenes and Stiefel, 1952; Saad and Schultz, 1986; Saad, 2003). Nowadays, large scale simulations are performed on parallel computing platforms where many computing units collaborate to solve a common global problem. In that framework, the hybrid methods allow the combination of direct and iterative solvers in their best functioning regimes: the direct solver concurrently and independently on small enough sub-problems and the iterative solver to ensure the consistency of the global computed solution. This methodology is termed algebraic, because the solver can fully be defined based on its main input data that simply are $\mathcal{K}$ and f to compute the solution u.

Domain decomposition methods

The hybrid approaches discussed above are closely related to the Domain Decomposition Methods (DDM) (Quarteroni and Valli, 1999; Dryja and Widlund, 1994; Mathew, 2008; Toselli and Widlund, 2006), the pioneer idea was originally introduced by Schwarz (1870) as a proof of existence of solutions to a PDE, and subsequently extended and adapted to parallel computing by Lions (1988, 1989, 1990). Often, DDM are considered at the continuous level before discretization; the domain, where the PDE is defined, is split into N subdomains with new boundary conditions that are defined on the interfaces introduced by the splitting. Next, the PDE is discretized independently in each subdomains. The

solution of the local sub-problems, at the subdomain level, is involved in a global procedure that ensures the continuity of the solution across the interface between the subdomains. In Chapter 2, some DDM are introduced both in a continuous setting as well as in their more algebraic counterpart as long as the global matrix $\mathcal{K}$ can be expressed as the sum of local matrices $\mathcal{K} = \sum_{i=1}^{N} \mathcal{R}_{\Omega_i}^T \mathcal{K}_i \mathcal{R}_{\Omega_i}$ where $\mathcal{R}_{\Omega_i}$ denotes the canonical restriction of a global vector defined on Ω to the subdomain Ω_i. It is further possible to eliminate the interior unknowns (those that do not have any interaction with other subdomains) so that the problem can be written in a condensed form $\mathcal{S} u_\Gamma = \widetilde{f}_\Gamma$, where $\mathcal{S} = \sum_{i=1}^{N} \mathcal{R}_{\Gamma_i}^T \mathcal{S}_i \mathcal{R}_{\Gamma_i}$ is the Schur complement system. This new system only involves quantities that are defined on the interfaces (unknowns that are shared by more than one subdomain).

Convergence of abstract Schwarz methods

The original method by Schwarz (1870), parallelized by Lions (1988) in its continuous form, has been extended in an algebraic framework into an additive Schwarz preconditioner (Dryja and Widlund, 1987; Matsokin and Nepomnyaschikh, 1985) for Symmetric Positive Definite (SPD) matrices $\mathcal{K}$. Including other well-known preconditioners such as Neumann-Neumann, they belong to a wider family of preconditioners named abstract Schwarz (Toselli and Widlund, 2006), whose generic form is $\mathcal{M}_{aS} = \sum_{i=1}^{N} \mathcal{R}_i^T \widehat{\mathcal{A}}_i^\dagger \mathcal{R}_i$ where $\mathcal{A}$ is either $\mathcal{K}$ or $\mathcal{S}$ depending on whether or not the internal unknowns have been eliminated. In general, when these preconditioners are used to accelerate the convergence of the conjugate gradient method (Hestenes and Stiefel, 1952), the number of iterations increases when N increases. In Chapter 3, we show how the condition number of the abstract Schwarz preconditioned systems can be bounded by bounding the generalized Rayleigh quotients of the local component $\widehat{\mathcal{A}}_i$ of the preconditioner $\mathcal{M}_{aS}$ with respect to their Neumann-Neumann and additive Schwarz counterparts. Following a methodology introduced by Spillane et al. (2014a); Spillane and Rixen (2013), we propose to build an adaptive coarse space by solving, on each subdomain, the generalized eigenproblems associated with the above mentioned generalized Rayleigh quotients. With this correction, we show that the condition number is bounded independently of N. This correction is generally applied through a deflation technique where the coarse space component is computed via a direct solver on the coarse space and a projection is applied at each iteration of the iterative scheme to ensure that the iteratively computed components do not come back into the coarse space. In the particular case of the additive Schwarz preconditioner, the coarse space correction can algebraically be handled as the N components associated with the subdomains. We refer to (Efendiev et al., 2012; Galvis and Efendiev, 2010; Haferssas et al., 2017; Klawonn et al., 2016b, 2018; Nataf et al., 2011) for other results on closely related techniques.

Algebraic domain decomposition method in python: the ddmpy toolbox

The various numerical methods introduced above can be implemented in combining various dense or sparse, iterative or direct, sequential or parallel solvers. In order to efficiently design new hybrid solvers and algebraic domain decomposition solvers, we propose in Chapter 4 a python toolbox. Python is a high level interpreted programming language. At the price of slower execution time, developing codes in python is often much faster compared with compiled languages such as C, C++ or Fortran, By analyzing the performance of various implementations of a same numerical kernel (vector sum, matrix-matrix multiplication, Schur complement calculation, conjugate gradient), we show that it is possible to write efficient python codes, if computing intensive kernels are implemented using third party optimized libraries (often written in compiled languages).

Based on this observation, we have designed a python toolbox to develop efficient parallel algebraic domain decomposition solvers. This module, called `ddmpy`, is presented in a literate programming style defined by Knuth (1984):

> *Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.*

Consequently, the complete `ddmpy` code is included in this chapter; the provided examples can be run interactively by opening the `.org` (Dominik, 2010; Schulte and Davison, 2011) source file in a text editor that supports the functionalities of this format (*e.g.* emacs).

A scalability study is presented up-to 3,072 cores: the performance of 16 algebraic domain decomposition solvers built using `ddmpy` are compared. All the python scripts implemented to perform this comparison are included in the source of this document and can consequently be reproduced.

Parallel design of a coarse space correction

In Chapter 5, we describe and compare the performance of several parallel strategies to compute the coarse space correction introduced in Chapter 3. Without this coarse space correction calculation, the abstract Schwarz preconditioners do not require any global communications that constitute known parallel performance bottlenecks. This lack of global communication is actually their numerical weakness: the information only moves from one subdomain to its neighbors at each iteration; for a one-dimensional decomposition N iterations are necessary to transfer the information between the first and last subdomains. The underlying idea of the coarse space correction is to allow for a global information exchange of a coarse information through the solution of a small problem (with typically a few unknowns per subdomain). This numerical mechanism allows the reduction of the number of iterations that conversely become more expensive on a large parallel platform. Several parallel strategies for the implementation of the coarse space correction kernels are considered, including the solution of the coarse problem on all or

a subset of cores involved in the global communicator, with or without replication of the solution of the coarse problem. We implemented these strategies in the fully features Fortran 90 hybrid solver MaPHyS[4] (Agullo et al., 2011, 2016b; Giraud et al., 2008). These implementations are compared for the solution of structured highly heterogeneous diffusion problems with 1D or 3D decompositions. The new solver has been integrated in two application codes. Firstly in the Alya code[5] (Vázquez et al., 2016) to compute the flow in a human respiratory system. Secondy, in the AVIP code (Joncquieres et al., 2018) that simulated plasma propulsion devices. In both case, the linear systems arise from the discretization of a Poisson equation on 3D unstructured meshes.

Perspective and future work

Robust methods for non symmetric problems

The work presented in chapters 3 and 5 addresses robust abstract Schwarz methods for SPD problems: the proof and implementation strongly rely on the fact that all matrices have real positive eigenvalues. The general description of the domain decomposition methods in Chapter 2, as well as the parallel implementation proposed in Chapter 4 do not have this limitation, it would be interesting to design robust methods for non symmetric problems using the `ddmpy` toolbox. In particular, `ddmpy` can support algebraic optimized Robin techniques either as a Robin-Robin preconditioner (Achdou and Nataf, 1997), or using a N-Lagrange formulation (Nataf et al., 1995). There is an ongoing action in the Inria team to experiment those techniques for the solution of large wave propagation problems. Another avenue of interest would be to design multi-preconditioned schemes (Bridson and Greif, 2006; Greif et al., 2011; Spillane, 2016).

C++ implementation of an algebraic domain decomposition library

Another option, opened by the results of this PhD work, is to write domain decomposition algorithms in an algebraic formalism. The python language selected to design the `ddmpy` toolbox has some weaknesses in term of parallelism and the possibility to be called by application code developed in C, C++ or Fortran. We believe that these limitations could be overcome by using recent C++ standard without loosing the algorithmic expressivity enabled by python. A new project has been recently initiated in this direction[6].

[4] See `https://gitlab.inria.fr/solverstack/maphys/`

[5] `https://www.bsc.es/research-development/research-areas/engineering-simulations/alya-high-performance-computational`

[6] `https://gitlab.inria.fr/solverstack/maphys/maphyspp`

Chapter 1

Introduction

1.1 Historical context

1.1.1 Newton, Leibniz and the introduction of Calculus

The invention of modern calculus by Leibniz (1684) and Newton (1687) provided a new method for formulating and solving physical problems, using differentiation and integration of mathematical functions.

One problem of particular historical and practical importance (Gander and Wanner, 2014) is known as *Laplace's equation*: given a domain Ω and a function $v_D : \partial\Omega \to \mathbb{R}$ defined on its boundary

$$\begin{aligned} &\text{find} \quad u : \Omega \to \mathbb{R} \quad \text{such that} \\ &\begin{cases} \Delta u = \partial_{xx} u + \partial_{yy} u + \ldots = 0 \text{ in } \Omega, & (1.1) \\ \quad u = v_D \text{ on } \partial\Omega. & (1.2) \end{cases} \end{aligned}$$

Equation (1.1) states that the sum of second-order derivatives along all dimensions must be null. Equation (1.2) is called a *Boundary Condition* (BC). As it specifies the value of the function on the boundary, it is called a *Dirichlet*, or *first-type* BC. It would be possible, instead, to specify the value of the normal derivative $\partial_{\vec{n}} u = v_N$ on the boundary: this would be called a *Neumann* or *second-type* BC. A third type of BC is the *Robin* BC in which the value of a linear combination of u and its normal derivative would be specified: $\partial_{\vec{n}} u + pu = v_R$. Different types of BC may be used on different parts of the boundary $\partial\Omega$, yielding a *mixed* BC.

A more generic PDE in a domain Ω, including its boundary condition on $\partial\Omega$ can be written as

$$\begin{aligned} &\text{find } u : \Omega \to \mathbb{R} \text{ such that} \\ &\begin{cases} \mathcal{L}(u) = f \text{ in } \Omega, \\ \mathcal{B}(u) = g \text{ on } \partial\Omega, \end{cases} \end{aligned} \tag{1.3}$$

where f and g are given functions and $\mathcal{L}$ and $\mathcal{B}$ are *linear* operators. This means that given any functions u_1 and u_2 and scalar λ, we have $\mathcal{L}(u_1 + \lambda u_2) = \mathcal{L}(u_1) + \lambda\mathcal{L}(u_2)$ and

$\mathcal{B}(u_1 + \lambda u_2) = \mathcal{B}(u_1) + \lambda\mathcal{B}(u_2)$. Some non-linear problems can be solved too by solving a sequence of linear problems, in a process called *linearization*.

When needed in this introduction and in Chapter 2, we will choose clarity and concision over precision: the goal is not to introduce rigorously all the theory but to guide the reader through some analytic and algebraic methods and illustrate the connection between them. References are provided in each section for more accurate and in-depth definitions and analyses.

Our discussion will be illustrated by a very simple example: Poisson's equation in a rectangle with a mixed BC. *Poisson's equation* is a variation of Laplace's equation obtained by adding a source term f, replacing Equation (1.1) with $\Delta u = f$. The domain is the rectangle $\Omega = [0,3] \times [0,1]$. The source term is chosen to be $f = -1$. A Dirichlet BC $u = 0$ is chosen on the left of the domain, and a Neumann BC is chosen on the right ($\partial_x u = 0$), top and bottom ($\partial_y u = 0$), as presented in Figure 1.1. Our example problem is

$$\begin{aligned} &\text{find} \quad u : \Omega \to \mathbb{R} \quad \text{such that} \\ &\begin{cases} \Delta u = \partial_{xx} u + \partial_{yy} u = -1, \\ u(0,\cdot) = 0, \\ \partial_x u(3,\cdot) = \partial_y u(\cdot,0) = \partial_y u(\cdot,1) = 0. \end{cases} \end{aligned} \tag{1.4}$$

$$\begin{array}{ccc} & \partial_y u(\cdot,1) = 0 & \\ u(0,\cdot) = 0 & \boxed{\Delta u = -1} & \partial_x u(3,\cdot) = 0 \\ & \partial_y u(\cdot,0) = 0 & \end{array}$$

Figure 1.1: Problem (1.4): steady-state heat equation in a rectangle, with a Dirichlet BC on the left and Neumann BC on the top, right and bottom.

Since there is in fact no dependency on y in Problem (1.4), the system reduces to an ordinary differential equation $u''(x) = -1$, with a BC $u(0) = u'(3) = 0$. Integrating twice gives $u(x) = -\frac{1}{2}x^2 + ax + b$. The BC gives $a = 3$ and $b = 0$. The analytical solution of Problem (1.4) is therefore $u(x,y) = -\frac{1}{2}x^2 + 3x$ and is represented in red in Figure 1.5.

1.1.2 Schwarz and the birth of domain decomposition methods

Fourier series (1807, 1822) gave a solution to Laplace's equation for simple geometries (such as a circle or a rectangle). However, in the general case, the mere existence of a solution remained an open question.

When the domain Ω is the union of two simpler subdomains Ω_1 and Ω_2 on which a solution can be computed, Schwarz (1870) gave a constructive proof of existence of a global solution on Ω (see Figure 1.2). He devised an iterative method, the *Schwarz Alternating Method*, for solving Laplace's equation in Ω by solving it alternatively in each

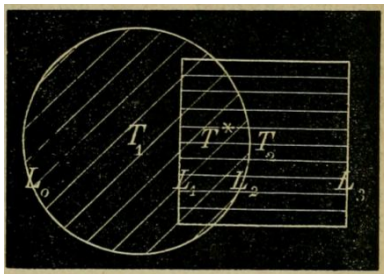


Figure 1.2: Two overlapping subdomains as drawn by Schwarz (1870). The two subdomains Ω_1 and Ω_2 are labeled T_1 and T_2, and their intersection $\Omega_1 \cap \Omega_2$ is labeled T^*.

simpler subdomain, namely the circle Ω_1 and the square Ω_2, using the value of the previous solution in the other subdomain as a Dirichlet BC

find ${u_1}^{(n)} : \Omega_1 \to \mathbb{R}$ such that
$$\begin{cases} \Delta u_1^{(n)} = 0 \text{ in } \Omega_1, \\ u_1^{(n)} = v_D \text{ on } \partial\Omega \cap \partial\Omega_1, \\ u_1^{(n)} = u_2^{(n-1)} \text{ on } \partial\Omega \cap \Omega_2; \end{cases}$$

and ${u_2}^{(n)} : \Omega_2 \to \mathbb{R}$ such that
$$\begin{cases} \Delta u_2^{(n)} = 0 \text{ in } \Omega_2, \\ u_2^{(n)} = v_D \text{ on } \partial\Omega \cap \partial\Omega_2, \\ u_2^{(n)} = u_1^{(n)} \text{ on } \partial\Omega \cap \Omega_1. \end{cases}$$

Schwarz proved that for any initial choice for $u_1^{(1)}$, $u_1^{(n)}$ and $u_2^{(n)}$ converge towards the same function in the overlap $\Omega_1 \cap \Omega_2$, giving a unique limit $u : \Omega \to \mathbb{R}$ solution of Laplace's equation in $\Omega = \Omega_1 \cup \Omega_2$. Variations on this method are presented in Section 2.7.

1.1.3 The finite element method

Ritz (1908) developed a more general method for solving problems like (1.3) by expressing the solution as a sum of basis functions. First, a *weak formulation* for the problem has to be introduced: Problem (1.3) is rewritten

$$\text{find } u \in V \quad \text{such that} \quad \forall v \in \tilde{V} \quad a(u, v) = f(v)$$

where V and $\tilde{V}$ are suitable function spaces, a is a bilinear form and f is a linear form. This is typically obtained by multiplying Equation (1.3) by a test function v, and integrating over Ω.

The second step is to replace V and $\tilde{V}$ with subspaces V_n and $\tilde{V}_n$ of finite dimension n

$$\text{find } u \in V_n \quad \text{such that} \quad \forall v \in \tilde{V}_n \quad a(u, v) = f(v).$$

Projecting u and v onto bases of V_n and $\tilde{V}_n$ gives a discrete linear system that can be written as a matrix equation

$$\mathcal{K}u = f, \tag{1.5}$$

where $\mathcal{K}$ is a (n, n) matrix and u and f are vectors of $\mathbb{R}^n$. u is called the *unknown*, and f is called the *Right-Hand Side*.

Ritz's method was immediately put to use and perfected in Russia, where it was popularized by Galerkin (1915). More historical perspective is provided in (Gander and Wanner, 2012).

A case of particular interest is when using a particular set of basis functions defined on a mesh. This method, known as the *Finite Element Method* (FEM), yields a *sparse* system, meaning that most coefficients in the matrix $\mathcal{K}$ are equal to zero. This sparsity can be exploited, in practice, to reduce both the memory needed to represent the matrix and the computational cost of some operations such as the matrix-vector product or factorization.

In the current section as well as in Chapter 2, the FEM will often be used to relate operations performed on the continuous PDE to their algebraic counterpart performed on the matrix to illustrate the link between them. However, the algorithms presented here never rely explicitly on a particular discretization method: they only may have some algebraic prerequisites on the matrices, such as symmetry or positive definiteness that are best illustrated using the FEM. More advanced discretization techniques such as, for instance, the *Hybrid Discontinuous Galerkin* method (HDG) (Cockburn et al., 2009) can generate matrices with the same properties, as shown in (Agullo et al., 2016c).

A weak formulation of Poisson's equation in Problem (1.4) is obtained by multiplying the PDE by v and integrating over Ω

$$\text{find } u \text{ such that } \forall v \quad \int_\Omega \Delta u \; v \; d\Omega = \int_\Omega -v \; d\Omega,$$

which after integration by parts gives the symmetric weak formulation

$$\text{find } u \text{ such that } \forall v \quad \int_\Omega \nabla u \cdot \nabla v \; d\Omega = \int_\Omega v \; d\Omega. \tag{1.6}$$

We introduce a mesh of 3 square elements and 8 vertices over Ω (known as a Q_1

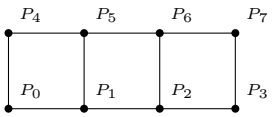


Figure 1.3: Mesh used to discretize the weak formulation (1.6). The labels of the nodes represent the ordering of the indices in the matrix in Equation (1.7).

discretization), as presented in Figure 1.3. The 8 points in the mesh are labelled P_i $(0 \leq i \leq 7)$. Then, we introduce the subspace generated by the basis functions e_i, for $0 \leq i \leq 7$ defined as

$$\begin{cases} e_i(P_i) = 1, & \\ e_i(P_j) = 0 & \text{if} \quad j \neq i, \\ e_i(x, y) = a + bx + cy + dxy & \text{inside each element of the mesh,} \end{cases}$$

and in each element, there is only one choice of a, b, c and d that gives the prescribed value (1 or 0) on the 4 corners. Each basis function e_i is therefore continuous and piecewise-polynomial on Ω.

The solution can then be searched as $u = \sum_{i=0}^{7} u_i e_i$. The resulting global system is

$$\begin{array}{c} \\ e_0 \\ e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \\ e_6 \\ e_7 \end{array} \begin{array}{c} \begin{array}{cccccccc} e_0 & e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1.33 & -0.167 & 0 & 0 & -0.333 & -0.333 & 0 \\ 0 & -0.167 & 1.33 & -0.167 & 0 & -0.333 & -0.333 & -0.333 \\ 0 & 0 & -0.167 & 0.667 & 0 & 0 & -0.333 & -0.167 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & -0.333 & -0.333 & 0 & 0 & 1.33 & -0.167 & 0 \\ 0 & -0.333 & -0.333 & -0.333 & 0 & -0.167 & 1.33 & -0.167 \\ 0 & 0 & -0.333 & -0.167 & 0 & 0 & -0.167 & 0.667 \end{pmatrix} \end{array} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \end{pmatrix} = \begin{pmatrix} 0 \\ 0.5 \\ 0.5 \\ 0.25 \\ 0 \\ 0.5 \\ 0.5 \\ 0.25 \end{pmatrix}; \tag{1.7}$$

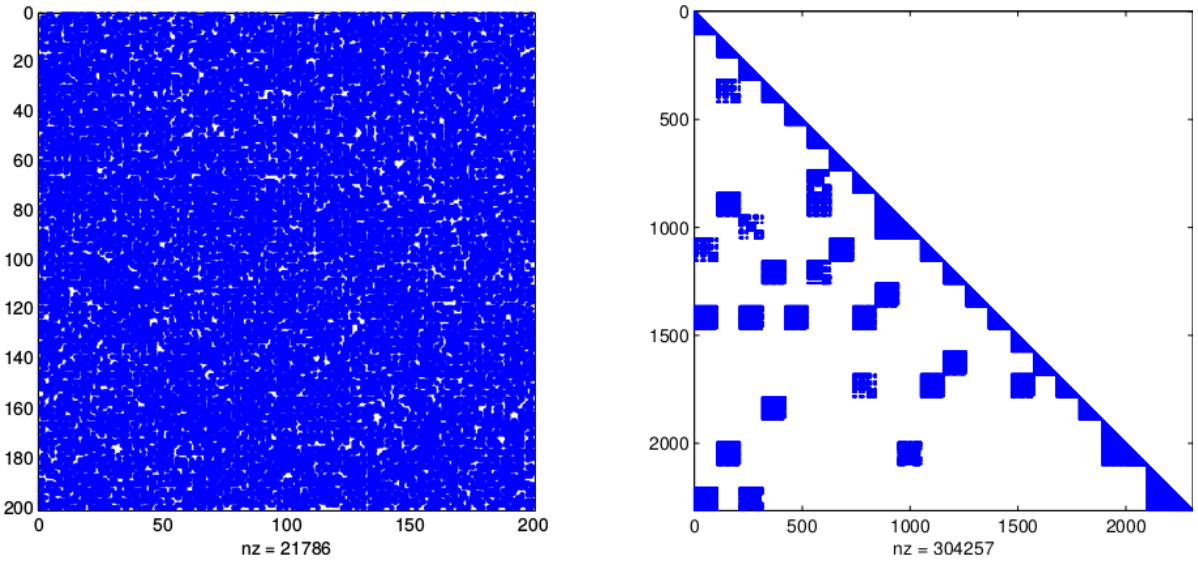
this linear system can be solved using one of the methods presented in Section 1.1.4.

1.1.4 Sparse linear solvers

Sparse matrices. A matrix is said to be sparse if it contains only very few nonzero elements, as depicted in Figure 1.4b, where nonzero elements are represented in blue color. There is no accurate definition of the proportion of nonzero elements in sparse matrices. However, a matrix can be considered as sparse when one can take advantage computationally of taking into account only its nonzero elements. Even if the matrix presented in Figure 1.4a contains 54 % of nonzero elements, it cannot be termed sparse, althought the one presented in Figure 1.4b can be clearly considered as sparse. As reported for example by Saad (2003), partial differential equations are among the most important sources of sparse matrices. These matrices are not only sparse, but they may also be very large, which leads to a storage problem. For example, a matrix $\mathcal{A} \in \mathbb{C}^{n \times n}$, of order $n = 10^6$, contains $n \times n = 10^{12}$ elements (zero and nonzero elements). In double precision arithmetic, 16 terabytes[1] are necessary to store all its entries. There are special data structures to store sparse matrices and their main goal is to store only non-zero elements while at the same time facilitating algebraic operations. We refer to (Saad, 2003) for a detailed description of possible data structures for sparse matrices.

Direct methods. To solve a linear system of equations of form $\mathcal{A}x = b$ where $\mathcal{A}$ is a square non-singular matrix of order n, b is the right-hand side vector and x is the unknown vector, as illustrated by Equation (1.7), a broad class of methods is based on Gaussian elimination (Gauss, 1811; Grcar, 2011). One variant decomposes the coefficient matrix of the linear system (here $\mathcal{A}$) into a product of a lower triangular matrix L (diagonal elements of L are unity) and of an upper triangular matrix U such that $\mathcal{A} = LU$. This decomposition is called the LU factorization of the matrix $\mathcal{A}$; for the matrix in (1.7) those

[1] $10^{12} \times 2 \times 8$ bytes $= 16 \times 10^{12}$ bytes. Each complex element requires 2×8 bytes, 8 bytes for imaginary part and 8 for real part, in double precision.



(a) matrix with 54% of nonzero elements.

(b) matrix with 3% of nonzero elements.

Figure 1.4: Sparse matrices contains only a very few percentage of nonzero elements. With 54% of nonzero elements the matrix in (a) cannot be referred as sparse whereas, with only 3% of non zero elements, the matrix in (b) satisfies a sparsity criterion.

factors are:

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -0.125 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -0.127 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & -0.25 & -0.286 & -0.0738 & 0 & 1 & 0 & 0 \\ 0 & -0.25 & -0.286 & -0.59 & 0 & -0.338 & 1 & 0 \\ 0 & 0 & -0.254 & -0.324 & 0 & -0.0971 & -0.537 & 1 \end{pmatrix},$$

$$U = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1.33 & -0.167 & 0 & 0 & -0.333 & -0.333 & 0 \\ 0 & 0 & 1.31 & -0.167 & 0 & -0.375 & -0.375 & -0.333 \\ 0 & 0 & 0 & 0.646 & 0 & -0.0476 & -0.381 & -0.209 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1.14 & -0.385 & -0.111 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.788 & -0.423 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.277 \end{pmatrix}.$$

Once the LU factorization is performed, the linear system solution consists of two steps:

1: *the forward substitution* that solves the triangular systems $Ly = b$;
2: *the backward substitution* that solves $Ux = y$.

In our example, it computes $y = (0,\ 0.5,\ 0.56,\ 0.32,\ 0,\ 0.81,\ 1.2,\ 1.2)^T$, which leads to the solution $x = (0,\ 2.5,\ 4,\ 4.5,\ 0,\ 2.5,\ 4,\ 4.5)^T$, represented in blue in Figure 1.5.

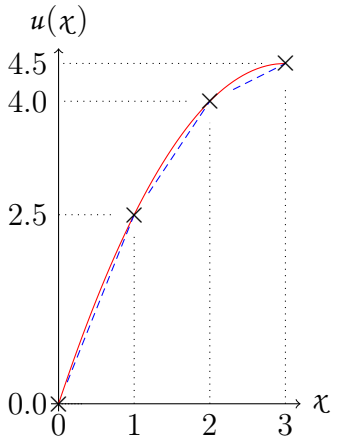


Figure 1.5: The solution to problems (1.4), in red, and (1.7), in blue. The solutions are constant along the y axis (not represented).

The advantage of this approach is that most of the work is performed in the decomposition step ($\mathcal{O}(n^3)$ floating point operations for dense matrices) and very little in the forward and backward substitutions ($\mathcal{O}(n^2)$ operations). The solution of successive linear systems using the same matrix but with different right-hand sides, often arising in practice, is then relatively cheap. Furthermore, if the matrix is symmetric (or symmetric positive definite), an LDL^T (or Cholesky) factorization may be performed. In finite arithmetics, direct methods enable one to solve linear systems in practice with a high accuracy in terms of backward error (Higham, 2002). However, this numerical robustness has a cost. First, the number of arithmetic operations is very large. Second, in the case of a sparse matrix $\mathcal{A}$, the number of non-zeros of L and U is often much larger than the number of non-zeros in the original matrix. This phenomenon, so-called fill-in, may be prohibitive in terms of memory usage and computational time. Solving large sparse linear algebra problems using direct methods is very challenging because of memory limitation.

In order to minimize computational cost and guarantee a stable decomposition and limited fill-in intensive studies have been carried on that lead to efficient code implementations such as CHOLMOD (Davis, 2008), Mumps (Amestoy et al., 2001), Pardiso (Schenk et al., 2000), Pastix (Hénon et al., 2002), SuperLU (Li and Demmel, 2003), to name a few. Sparse methods work well for matrices arising from 2D PDE discretizations, but they may be very penalizing in terms of memory usage and computational time for those coming from large 3D test cases.

Iterative methods To solve very large sparse problems, iterative solvers may be more scalable and considerably decrease the memory consumption. Iterative methods produce a sequence of approximates to the solution. On the one hand, successive iterations implemented by iterative methods require a small amount of storage and floating point operations. On the other hand iterative methods are generally less robust than direct solvers for general sparse linear systems, as they might converge slowly or not converge

at all. Iterative methods for linear systems are broadly classified into two main types: stationary and Krylov subspace methods.

Consider the solution of the linear system $\mathcal{A}x = b$; stationary methods can be expressed in the general form

$$Mx_{k+1} = Nx_k + b \tag{1.8}$$

where x_k is the approximate solution at the k^{th} iteration. The matrices N and M do not depend on k, and satisfy $\mathcal{A} = M - N$ with M non singular. These methods are called stationary because the solution to a linear system is expressed as finding the stationary fixed point of Equation (1.8) when k will go to infinity. Given any initial guess x_0, the stationary method described in Equation (1.8) converges if and only if $\rho(M^{-1}N) < 1$, where the spectral radius $\rho(A)$ of a given matrix A with eigenvalues λ_i is defined by $\rho(A) = max(|\lambda_i|)$ (Saad, 2003). Typical iterative methods for linear systems are Gauss-

M	N	Method
D	$(L+U)$	Jacobi
$(D-L)$	U	Gauss-Seidel
$((\frac{1}{\omega})D-L)$	$(((\frac{1}{\omega})-1)D+U)$	Successive Over Relaxation

Table 1.1: Stationary iterative methods for linear systems. D, $-L$ and $-U$ are the diagonal, strictly lower-triangular and strictly upper-triangular parts of $\mathcal{A}$, respectively.

Seidel, Jacobi, successive over relaxation etc., as described in Table 1.1 according to the choice of M and N.

Another approach to solve linear systems of equations consists in extracting the approximate solution from a subspace of dimension much smaller than the size of the coefficient matrix $\mathcal{A}$. This approach is called projection method. These methods are based on projection processes: orthogonal and oblique projection onto Krylov subspaces, which are subspaces spanned by vectors of the form $p(\mathcal{A})v$, where p is a polynomial (Saad, 2003). Let $\mathcal{A} \in \mathbb{R}^{n\times n}$ and $v \in \mathbb{R}^n$, let $m \leq n$, the space denoted by $\mathcal{K}_m(\mathcal{A}, v) = \text{span}\{v, \mathcal{A}v, ..., \mathcal{A}^{m-1}v\}$ is referred to as the Krylov space of dimension m associated with $\mathcal{A}$ and v. These techniques approximate $\mathcal{A}^{-1}v$ by $p(\mathcal{A})v$, where p is a specific polynomial. Based on a minimal polynomial argument, it can be shown that these methods converge in less than n steps compared to *"infinity"* for stationary schemes.

The convergence of Krylov subspace methods depends mostly on the numerical properties of the coefficient matrix $\mathcal{A}$. To accelerate the convergence, one may use a non-singular matrix $\mathcal{M}$ such that $\mathcal{MA}$ has *better* convergence properties for the selected solver. The linear systems $\mathcal{MA}x = \mathcal{M}b$ has the same solution as the original linear system. This method is called preconditioning and depending on the authors, either the matrix $\mathcal{M}$ or its inverse $\mathcal{M}^{-1}$ is called a left preconditioner; in this thesis, we chose the convention of calling $\mathcal{M}$ the preconditioner: the preconditioner should somehow approximate $\mathcal{A}^{-1}$. On the other hand, linear systems of equations can also be preconditioned from the right: $\mathcal{AM}y = b$, and $x = \mathcal{M}y$. One can also consider split preconditioning that is expressed as follows: $\mathcal{M}_1\mathcal{AM}_2y = \mathcal{M}_1b$, and $x = \mathcal{M}_2y$, where the preconditioner is $\mathcal{M} = \mathcal{M}_1\mathcal{M}_2$.

Krylov methods do not require the matrices $\mathcal{A}$ or $\mathcal{M}$ to be explicitly formed. Instead, procedures for applying $\mathcal{A}$ and $\mathcal{M}$ to a vector must be provided. Preconditioners are

commonly applied by performing sparse matrix-vector products or solving simple linear systems. The numerical requirement for a good preconditioner is that the spectrum of the preconditioned matrix is clustered. Such a feature generally ensures fast convergence of the Conjugate Gradient method (CG) for Symmetric Positive Definite (SPD) problems as illustrated by the CG convergence rate bound given by (Golub and Van Loan, 1996):

$$||e_k||_{\mathcal{A}} \leq 2 \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right)^k ||e_0||_{\mathcal{A}},$$

where $e_k = x^* - x_k$ denotes the error associated with the iterate at step k and κ is the condition number of the preconditioned linear system $\mathcal{M}^{\frac{1}{2}} \mathcal{A} \mathcal{M}^{\frac{1}{2}}$ (which is simply the ratio of the largest to smallest eigenvalue). From this bound, it can be seen that when the condition number is small (*i.e.*, $\kappa \approx 1$), CG converges rapidly. In addition to improving the spectral distribution, a preconditioner should be inexpensive to compute, to store and apply. In a parallel distributed framework, the construction and the application of the preconditioner should also be easily parallelizable.

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 $\mathcal{A}$-norm over the Krylov space $\mathcal{K}_k = \text{span}\left\{r_0, \mathcal{A}r_0, \ldots, \mathcal{A}^{k-1} r_0\right\}$, where $r_0 = b - \mathcal{A}x_0$. It exists a rich literature dedicated to this method; for more details we, non-exhaustively, refer to (Golub and Van Loan, 1996; Saad, 2003) and the references therein.

A key ingredient in iterative method is the stopping criterion that should control the quality of the computed solution. The backward error analysis, introduced by Givens and Wilkinson (Wilkinson, 1963), 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 (1.9) 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 Chaitin-Chatelin and Frayssé (1996); Higham (2002). 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 (Barrett et al., 1994).

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

$$\begin{aligned} \eta_{\mathcal{A},b}(x_k) &= \min_{\Delta\mathcal{A},\Delta b} \{\tau > 0 : \|\Delta A\| \leq \tau\|\mathcal{A}\|,\ \|\Delta b\| \leq \tau\|b\| \\ &\qquad \text{and } (\mathcal{A} + \Delta A)x_k = b + \Delta b\}, \\ &= \frac{\|\mathcal{A}x_k - b\|}{\|\mathcal{A}\|\|x_k\| + \|b\|} \end{aligned} \tag{1.9}$$

is called the *normwise backward error* associated with x_k. It measures the norm of the smallest perturbations $\Delta\mathcal{A}$ on $\mathcal{A}$ and Δb on b such that x_k is the exact solution of $(\mathcal{A} + \Delta\mathcal{A})x_k = b + \Delta b$. 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) $\|\mathcal{A}\|$. Consequently, another backward error criterion can be considered that is simpler to evaluate and implement in practice. It is defined by

$$\begin{aligned} \eta_b(x_k) &= \min_{\Delta b} \{\tau > 0 : \|\Delta b\| \leq \tau\|b\| \text{ and } \mathcal{A}x_k = b + \Delta b\}, \\ &= \frac{\|\mathcal{A}x_k - b\|}{\|b\|}. \end{aligned} \tag{1.10}$$

This latter criterion measures the norm of the smallest perturbations Δb on b (assuming that they are no perturbations on $\mathcal{A}$) such that x_k is the exact solution of $\mathcal{A}x_k = b + \Delta b$.

The stopping criteria of the Krylov solvers we used for our numerical experiments are based on η_b.

Hybrid methods Direct methods are based on the Gaussian elimination that is probably among the oldest methods (Gauss, 1811; Grcar, 2011) for solving linear systems. Tremendous efforts have been devoted to the design of sparse Gaussian elimination that efficiently exploits the sparsity of the matrices. These methods indeed aim at exhibiting dense submatrices that can then be processed with computational intensive standard dense linear algebra kernels. Sparse direct solvers have been for years the methods of choice for solving linear systems of equations because of their reliable numerical behavior (Higham, 2002). Although there are ongoing efforts in further improving existing parallel packages, such approaches may not be scalable in terms of computational complexity and memory for large problems such as those arising from the discretization of large 3D PDEs. Furthermore, the linear systems involved in the numerical simulation of complex phenomena result from some modeling and discretization, which contain some uncertainties and approximation errors. Consequently, the highly accurate but costly solution provided by stable Gaussian elimination might not be mandatory.

Iterative methods, on the other hand, generate sequences of approximations to the solution either through fixed point schemes or via search in Krylov subspaces. These methods have the advantage that the memory requirements are low. Also, they tend to be easier to parallelize than direct methods. However, the main problem with this class

of methods is the rate of convergence, which depends on the properties of the matrix. In many computational science areas, highly accurate solutions are not required as long as the quality of the computed solution can be assessed against measurements or data uncertainties. In such a framework, the iterative schemes play a central role as they might be stopped as soon as an accurate enough solution is found.

An alternative approach for the high-performance, scalable solution of large sparse linear systems in parallel scientific computing is to combine direct and iterative methods. Such an hybrid approach exploits the advantages of both direct and iterative methods. The iterative component allows us to use a small amount of memory and provides a natural way for parallelization. The direct part provides its favorable numerical properties. Furthermore, this combination enables us to exploit naturally several levels of parallelism that logically match the hardware feature of multicore platforms. In particular, one can use parallel multithreaded sparse direct solvers within the multicore nodes of the machine and message passing among the nodes to implement the gluing parallel iterative scheme. The general underlying numerical ideas are not new. They are often closely related to ideas considered in the design of domain decomposition techniques for the numerical solution of PDEs (see (Mathew, 2008) and references therein). With the need of solving ever larger sparse linear systems while maintaining numerical robustness, multiple variants for computing preconditioners for the Schur complement of such hybrid solvers have been proposed. PDSLin (Li et al., 2009), ShyLU (Rajamanickam et al., 2012b) and HIPS (Gaidamour and Hénon, 2008) first perform an exact factorization of the interior of each subdomain concurrently. PDSLin and ShyLU then compute the preconditioner with a two-fold approach. First, an approximation of the (global) Schur complement is computed. Second, this approximate Schur complement is factorized to form the preconditioner for the Schur complement system, which does not need to be formed explicitly. While PDSLin has multiple options for discarding values lower than some user-defined thresholds at different steps of the computation of the approximation of the Schur complement, ShyLU implements a structure-based approach for discarding values named *probing* and that was first proposed to approximate interfaces in domain decomposition (Chan and Mathew, 1992). Instead of following such a two-fold approach, HIPS forms the preconditioner by computing a global ILU factorization based on the multilevel scheme formulation from Hénon and Saad (2006). Finally, MaPHyS (Carvalho et al., 2001b), computes an Additive Schwarz preconditioner for the Schur complement; it will be further detailed in Chapter 5.

1.2 Our positioning

The most straightforward way to solve a PDE, as introduced in Section 1.1, is to discretize it and use an algebraic solver to solve the resulting linear system. This methodology can be detailed as follows:

1. the PDE and its BC are expressed on the global domain (Section 1.1.1),
2. a discretization method such as the FEM (Section 1.1.3) is used to transform the PDE and its BC into an algebraic linear system $\mathcal{K}u = f$,

3. an algebraic linear solver (Section 1.1.4) is used to compute the discrete solution u,

4. the discrete solution u may be interpolated to compute an approximate solution of the PDE.

Each of these steps introduces some error, and the final solution is only an approximate solution of the initial problem. One way to reduce the *discretization error* (step 2) is to use a finer mesh, allowing for a better approximation of continuous functions by vectors defined on the mesh. Once the size of the domain is fixed, using a finer mesh leads to more discrete unknowns and a larger linear system to solve (step 3).

Large-scale problems can be solved using, for instance, an hybrid (direct/iterative) solver (Gaidamour and Hénon, 2008; Giraud et al., 2008; Rajamanickam et al., 2012a; Saad and Sosonkina, 1999) in step 3:

3. a) a graph partitioner is used to partition the set of unknowns,

3. b) $\mathcal{K}u = f$ is replaced by an equivalent formulation $\sum_{i=1}^{N} \mathcal{R}_i^T \ \mathcal{A}_i \ \mathcal{R}_i = b$ using the partition from 3. a),

3. c) local linear systems are solved using direct solvers,

3. d) the global problem is solved using an iterative solver.

In this algebraic methodology, the global matrix is replaced by a set of smaller local matrices which can be used in a preconditioner to accelerate the iterative solution of the global linear system. This methodology takes advantage of both the robustness of direct solvers and the low complexity of iterative solvers when used on large matrices in a distributed memory setting.

One can instead partition the problem at step 2, before discretization, by using a Domain Decomposition (DD) approach as introduced in Section 1.1.2 and further detailed in Chapter 2. In this approach, steps 2 and 3 are replaced by

2. a) the global domain is decomposed in smaller subdomains with new BC on the interfaces between subdomains,

2. b) the local problems are discretized, yielding a set of local algebraic problems $\mathcal{A}_i x_i = b_i$,

3. a) an algebraic linear solver is used to compute the solution of the local subproblems,

3. b) an iterative solver is used enforce the compatibility of the solution between the subdomains.

Although these two latter approaches bear some similarity, they differ in the fact that a DD approach can rely on more information than the global matrix and the right-hand side that are provided to an algebraic hybrid solver. Although this makes it possible in the DD approach to optimize the choice of the decomposition (*i.e.*, the definition of the subdomains), or the choice of a formulation and preconditioner as well as their parameters (see, for instance, the choice of a transmission operator in (Gander, 2006)), this

implies a tight integration of the DD algorithm inside the application code. As a result, algebraic hybrid solvers are more suited to the development of black-box libraries that can be reused in various applications, reducing the cost of High Performance Computing (HPC) optimizations. For instance, focusing on the MaPHyS solver (Giraud et al., 2008), recent developments include the use of modern parallelization paradigms such as two-level parallelism (Agullo et al., 2016b) and task-based parallelism (Agullo et al., 2016a), improving the resilience (Agullo et al., 2013) and fault-tolerance (Agullo et al., 2017) of the solver and developing new graph partitioning strategies for hybrid solvers (Casadei et al., 2013).

An innovative framework for developing HPC DD solvers was developed by Jolivet et al. (2013). The `hpddm` library interacts with the application code through the element matrices provided by a third-party discretization tool. It is not however fully algebraic, as stated on the project main page (`https://github.com/hpddm/hpddm`): "*For building robust two-level methods, an interface with a discretization kernel like FreeFem++ or Feel++ is also needed. It can then be used to provide, for example, elementary matrices, that the GenEO approach requires. As such preconditioners assembled by HPDDM are not algebraic, unless only looking at one-level methods. Note that for substructuring methods, this is more of a limitation of the mathematical approach than of HPDDM itself.*"

In this thesis, we aim at combining the advantages of both the algebraic and DD approaches. For that, while remaining as algebraic as possible, we identify some key information to be provided to the solver alongside the matrix in order to use robust Domain Decomposition Methods (DDM). Other work in this direction include, among others, recent developments by Agullo et al. (2016c); Gander et al. (2007); Spillane et al. (2014a).

An overview of classic DD methods is provided in Chapter 2, using both functional and algebraic notations. In Chapter 3, we prove that for Symmetric Positive Definite (SPD) problems, providing the matrix in a distributed fashion as a sum of Symmetric Positive Semi-Definite (SPSD) matrices is enough to build a robust and scalable hybrid solver. The only input needed are the subdomain matrices obtained by discretizing the PDE with Neumann BC on the interface between subdomains. For SPD problems, the SPSD condition for local matrices is for instance satisfied when applying a finite element method over a partitioned mesh, but more complex discretizations such as the hybridizable discountinuous Galerkin method (Cockburn et al., 2009) can be used instead.

In Chapter 4, the DD methods introduced in chapters 2 and 3 are implemented in a parallel distributed DD toolbox in python called `ddmpy` exposed using a literate programming approach. All the code needed to reproduce the experiments in this chapter is included. Chapter 5 further studies different strategies for the parallel design of the coarse space management, proposed as an extension to the fully-featured HPC MaPHyS sparse hybrid solver.

Chapter 2

Domain decomposition methods

2.1 Introduction

The growing use of parallel computers for computational mathematics gave rise to a new interest for *Domain Decomposition Methods* (DDM), as explained for instance in the Preface of the proceedings of the first International Conference on Domain Decomposition Methods (Glowinski et al., 1988): "One of the motivations for organizing this conference was the growing popularity of decomposition methods over the last few years, mainly due to the *strong emergence of multiprocessor, array and parallel computers*. A second motivation was the *need to split very large scale problems into smaller ones* in order to solve them."

This chapter provides an overview of classical DDM. It will serve as a basis on top of which robust two-level algebraic preconditioners will be proposed in Chapter 3 and Chapter 4 will propose a parallel implementation of most of the methods introduced in the current chapter. DDM were first introduced using a functional formalism (as in Section 1.1.2), referred to as a *continuous* approach, that still prevails in the DD community. They can also be introduced in matrix notations, following an *algebraic* approach, which is often preferred by the linear solver and HPC community. Despite the risk of being redundant, we decided to expose both approaches in this chapter in order to try and bridge the gap between those two communities, following Maday and Magoules (2006) and St-Cyr et al. (2007) among others.

DDM are first introduced on two subdomains: the introduction of an interface Γ is presented in Section 2.2. Then, the choice of a BC on Γ is discussed in Section 2.3 and 2.4. As for the Schwarz alternating method presented above in Section 1.1.2, DDM proceed by partitioning a global problem into a set of local subproblems. The definition of a particular DDM depends on the exact choice of these local subproblems and the (often iterative) procedure to enforce continuity between the subdomains. In Section 2.5, some reformulations of the global problem are proposed to decompose the global problem into an equivalent set of local problems, coupled through interface equations. A generalization to an arbitrary number N of subdomains is presented next in Section 2.6. Then, the complementary approach of building a global preconditioner through local problems is discussed in Section 2.7.

A more complete overview of DDM can be found in (Quarteroni and Valli, 1999; Toselli

and Widlund, 2006; Mathew, 2008; Dolean et al., 2015b) and in the proceedings of the international Domain Decomposition conference series[1].

2.2 Introduction of an interface in the global domain

As stated above, in order to solve larger problems that could not be solved globally, the domain can be partitioned into subdomains. This can be done either on the continuous problem before the discretization, by partitioning the domain into subdomains and introducing artificial interfaces in the domain, or on the algebraic problem after the discretization by partitioning the set of unknown indices. The introduction of a single interface to split the global domain into only two subdomains is introduced here, both for the continuous problem and for the linear system. A generalization to an arbitrary number N of subdomains is proposed in Section 2.6.

2.2.1 Introduction of an interface for the continuous problem

In our previous example (Problem (1.4), page 20), it is possible to decompose the domain Ω into two subdomains, for instance $\Omega_1 = [0,2]\times[0,1]$ and $\Omega_2 = [2,3]\times[0,1]$ by introducing an interface $\Gamma = \{(2,y), y \in [0,1]\}$, as presented in Figure 2.1. Problem (1.4) can then be

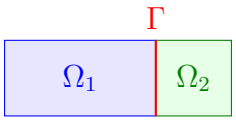


Figure 2.1: An interface $\Gamma = \{(2,y), y \in [0,1]\}$ is introduced to split Ω into two subdomains Ω_1 and Ω_2.

divided into two coupled subproblems in Ω_1 and Ω_2, by adding a compatibility condition on the solutions u_1 and u_2 in the subdomains: the solution and its first-order derivative across the interface ought to be continuous, leading to

$$\text{find } u_1 : \Omega_1 \to \mathbb{R} \text{ such that } \begin{cases} \Delta u_1 = -1 \text{ in } \Omega_1, \\ u_1(0,\cdot) = 0, \\ \partial_y u_1(\cdot,0) = \partial_y u_1(\cdot,1) = 0; \end{cases} \quad \text{and } u_2 : \Omega_2 \to \mathbb{R} \text{ such that } \begin{cases} \Delta u_2 = -1 \text{ in } \Omega_2, \\ \partial_x u_2(3,\cdot) = 0, \\ \partial_y u_2(\cdot,0) = \partial_y u_2(\cdot,1) = 0; \end{cases} \tag{2.1}$$

with the compatibility condition

$$\begin{cases} u_1(2,\cdot) = u_2(2,\cdot) & \text{(continuity of the solution)}, \quad (2.2) \\ \partial_x u_1(2,\cdot) = \partial_x u_2(2,\cdot) & \text{(continuity of the normal derivative)}. \quad (2.3) \end{cases}$$

The equation inside Ω_i and the BC on $\partial\Omega \cap \partial\Omega_i$ are the same in both subproblems associated with (2.1) as in Problem (1.4). The compatibility equations (2.2 – 2.3) introduce

[1] http://www.ddm.org/

a coupling between the two local solutions u_1 and u_2 through boundary conditions on Γ, ensuring that u_1 and u_2 are part of the same global solution u, *i.e.*, $u_i = u|_{\Omega_i}$.

In a more general case, when the global problem is as stated in (1.3), one can partition Ω into two subdomains Ω_1 and Ω_2 separated by an interface Γ, such that $\Omega_1 \cap \Omega_2 = \emptyset$ and $\Gamma = \left(\bar{\Omega}_1 \cap \bar{\Omega}_2\right) \setminus \partial\Omega$. We suppose that Problem (1.3) can be expressed as a coupled problem, through the introduction of a linear operator $\mathcal{N}_i$ to represent a continuity condition. In our example with Poisson's equation, $\mathcal{N}_i(u)$ is the outwards normal derivative $(-1)^{i+1}\, \partial_x u$. The new problem is

$$\begin{array}{ll} \text{find } u_1 : \Omega_1 \to \mathbb{R} \text{ such that} & \text{and } u_2 : \Omega_2 \to \mathbb{R} \text{ such that} \\ \begin{cases} \mathcal{L}(u_1) = f \text{ in } \Omega_1, \\ \mathcal{B}(u_1) = g \text{ on } \partial\Omega_1 \cap \partial\Omega \ ; \end{cases} & \begin{cases} \mathcal{L}(u_2) = f \text{ in } \Omega_2, \\ \mathcal{B}(u_2) = g \text{ on } \partial\Omega_2 \cap \partial\Omega \ ; \end{cases} \end{array} \tag{2.4}$$

with the compatibility condition

$$\begin{cases} u_1 = u_2 \text{ on } \Gamma, \\ \mathcal{N}_1(u_1) + \mathcal{N}_2(u_2) = 0 \text{ on } \Gamma. \end{cases} \tag{2.5}$$

2.2.2 Introduction of an interface in the algebraic linear system

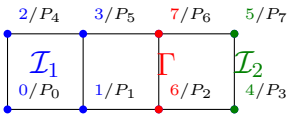


Figure 2.2: The set of indices can be partitioned into two interiors $\mathcal{I}_1$ and $\mathcal{I}_2$ and an interface Γ. In the graph, there are no edges between a node in $\mathcal{I}_1$ and a node in $\mathcal{I}_2$. The partition induces a new ordering of the unknowns.

Alternatively, the domain decomposition can be performed on the algebraic problem (1.5) after discretization, by partitioning the set of indices $\{0, 1, \ldots, 7\}$ in three subsets $\mathcal{I}_1 = \{0, 1, 4, 5\}$, $\mathcal{I}_2 = \{3, 7\}$ and $\Gamma = \{2, 6\}$. Then, it is possible to reorder these indices using the permutation $(\mathcal{I}_1\mathcal{I}_2\Gamma) = (0, 1, 4, 5, 3, 7, 2, 6)$ (see figures 1.3 and 2.2). This permutation can be applied to both row and column indices (so called *symmetric permutation*) in the matrix (and vectors), and the reordered system is

$$\begin{array}{c} \\ 0/e_0 \\ 1/e_1 \\ 2/e_4 \\ 3/e_5 \\ 4/e_3 \\ 5/e_7 \\ 6/e_2 \\ 7/e_6 \end{array} \begin{array}{c} \begin{array}{cccccccc} 0/e_0 & 1/e_1 & 2/e_4 & 3/e_5 & 4/e_3 & 5/e_7 & 6/e_2 & 7/e_6 \end{array} \\ \left(\begin{array}{cccc|cc|cc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1.333 & 0 & -0.333 & 0 & 0 & -0.167 & -0.333 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -0.333 & 0 & 1.333 & 0 & 0 & -0.333 & -0.167 \\ \hline 0 & 0 & 0 & 0 & 0.667 & -0.167 & -0.167 & -0.333 \\ 0 & 0 & 0 & 0 & -0.167 & 0.667 & -0.333 & -0.167 \\ \hline 0 & -0.167 & 0 & -0.333 & -0.167 & -0.333 & 1.333 & -0.333 \\ 0 & -0.333 & 0 & -0.167 & -0.333 & -0.167 & -0.333 & 1.333 \end{array}\right) \end{array} \begin{pmatrix} u_0 \\ u_1 \\ u_4 \\ u_5 \\ u_3 \\ u_7 \\ u_2 \\ u_6 \end{pmatrix} = \begin{pmatrix} 0 \\ 0.5 \\ 0 \\ 0.5 \\ 0.25 \\ 0.25 \\ 0.5 \\ 0.5 \end{pmatrix}. \tag{2.6}$$

It is important to note that, in the permuted matrix, there is a zero-block in the rows of $\mathcal{I}_1 = (0/e_0, \ldots, 3/e_5)$ and columns of $\mathcal{I}_2 = (4/e_3, 5/e_7)$ and vice versa: there is no direct interaction between the indices in $\mathcal{I}_1$ and $\mathcal{I}_2$ through the graph of the matrix.

Similarly, Problem (1.3) can be translated into an algebraic problem (1.5). One can partition the indices in the matrix $\{1, 2, \ldots, n\} = \mathcal{I}_1 \cup \mathcal{I}_2 \cup \Gamma$ such that $\mathcal{K}_{ij} = \mathcal{K}_{ji} = 0$ for any $i \in \mathcal{I}_1$ and $j \in \mathcal{I}_2$, and reorder the matrix with the permutation $(\mathcal{I}_1\ \mathcal{I}_2\ \Gamma)$. The resulting system, which exhibits a (3×3) block structure for the matrix, is

$$\begin{pmatrix} \mathcal{K}_{\mathcal{I}_1,\mathcal{I}_1} & 0 & \mathcal{K}_{\mathcal{I}_1,\Gamma} \\ 0 & \mathcal{K}_{\mathcal{I}_2,\mathcal{I}_2} & \mathcal{K}_{\mathcal{I}_2,\Gamma} \\ \mathcal{K}_{\Gamma,\mathcal{I}_1} & \mathcal{K}_{\Gamma,\mathcal{I}_2} & \mathcal{K}_{\Gamma,\Gamma} \end{pmatrix} \begin{pmatrix} u_{\mathcal{I}_1} \\ u_{\mathcal{I}_2} \\ u_\Gamma \end{pmatrix} = \begin{pmatrix} f_{\mathcal{I}_1} \\ f_{\mathcal{I}_2} \\ f_\Gamma \end{pmatrix}. \tag{2.7}$$

2.3 Local boundary conditions on the interface Γ

Each of the continuous local subproblems in (2.1) or (2.4) taken independently (without the compatibility condition) is not complete without a BC on $\partial\Omega_i \setminus \partial\Omega = \Gamma$. The continuous subproblems with a Dirichlet, Neumann or Robin BC on Γ are presented below, along with their discretizations. The compatibility condition is reintroduced next in Section 2.4 to deduce an optimal BC for the local subproblems.

2.3.1 Dirichlet boundary condition on Γ

We can add a Dirichlet BC $u_1(2, \cdot) = v_D$ on Γ for the left subproblem in our example (2.1), leading to the local problem

$$\begin{array}{l} \text{find} \quad u_1 : \Omega_1 \to \mathbb{R} \quad \text{such that} \\ \left\{ \begin{array}{r} \Delta u_1 = -1 \text{ in } \Omega_1, \\ u_1(0, \cdot) = 0, \\ \partial_y u_1(\cdot, 0) = \partial_y u_1(\cdot, 1) = 0, \\ u_1(2, \cdot) = v_D. \end{array} \right. \end{array} \tag{2.8}$$

Since $v_D'(0) = \partial_y u_1(2, 0) = 0$ and $v_D'(1) = \partial_y u_1(2, 1) = 0$, the Fourier series of v_D is

$$v_D = \sum_k a_k^D \cos(k\pi y).$$

Then, the solution of (2.8) is

$$u_1 = -1/2x^2 + x + a_0^D x/2 + \sum_{k \neq 0} a_k^D \frac{\sinh(k\pi x)}{\sinh(2k\pi)} \cos(k\pi y).$$

In the generic equation (2.4), the addition of a similar Dirichlet BC $u = v_D$ on Γ gives

the local problem

$$\begin{array}{l}\text{find} \quad u_1 : \Omega_1 \to \mathbb{R} \quad \text{such that} \\ \begin{cases} \mathcal{L}(u_1) = f \text{ in } \Omega_1, \\ \mathcal{B}(u_1) = g \text{ on } \partial\Omega_1 \cap \partial\Omega, \\ \quad u_1 = v_D \text{ on } \Gamma, \end{cases}\end{array} \tag{2.9}$$

whose discretization is

$$\begin{pmatrix} \mathcal{K}_{\mathcal{I}_1,\mathcal{I}_1} & \mathcal{K}_{\mathcal{I}_1,\Gamma} \\ 0 & I \end{pmatrix} \begin{pmatrix} u_{\mathcal{I}_1} \\ u_\Gamma \end{pmatrix} = \begin{pmatrix} f_{\mathcal{I}_1} \\ v_D \end{pmatrix}.$$

where I is an identity matrix and v_D is a discretization of v_D. Since the value on Γ is prescribed by the BC and needs not appear in the unknown vector, a simpler equivalent formulation is

$$\mathcal{K}_{\mathcal{I}_1,\mathcal{I}_1}\ u_{\mathcal{I}_1} = f_{\mathcal{I}_1} - \mathcal{K}_{\mathcal{I}_1,\Gamma} v_D, \tag{2.10}$$

Note that the matrix $\mathcal{K}_{\mathcal{I}_1,\mathcal{I}_1}$ in (2.10) is the top left matrix block in (2.7).

Therefore, in our example, the discretization of Problem (2.8) is

$$\begin{array}{c} \\ 0/e_0 \\ 1/e_1 \\ 2/e_4 \\ 3/e_5 \end{array} \begin{array}{c} \begin{array}{cccc} 0/e_0 & 1/e_1 & 2/e_4 & 3/e_5 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1.333 & 0 & -0.333 \\ 0 & 0 & 1 & 0 \\ 0 & -0.333 & 0 & 1.333 \end{pmatrix} \end{array} \begin{pmatrix} u_0 \\ u_1 \\ u_4 \\ u_5 \end{pmatrix} = f_D(v_D), \tag{2.11}$$

where $f_D(v_D)$ is a right-hand side that depends on the value of v_D. The matrix in Equation (2.11) is exactly the $(4,4)$ top left block of the matrix in Equation (2.6).

2.3.2 Neumann boundary condition on Γ

Instead of a Dirichlet BC, it is possible to impose a Neumann BC $\partial_{\vec{n}} u_1 = \partial_\chi u_1 = v_N = \sum_k a_k^N \cos(k\pi y)$ on Γ for the Poisson problem:

$$\begin{array}{l}\text{find} \quad u_1 : \Omega_1 \to \mathbb{R} \quad \text{such that} \\ \begin{cases} \Delta u_1 = -1 \text{ in } \Omega_1, \\ u_1(0,\cdot) = 0, \\ \partial_y u_1(\cdot,0) = \partial_y u_1(\cdot,1) = 0, \\ \partial_\chi u_1(2,\cdot) = \displaystyle\sum_k a_k^N \cos(k\pi y). \end{cases}\end{array} \tag{2.12}$$

The solution of this problem is

$$u_1 = -1/2\chi^2 + 2\chi + a_0^N \chi + \sum_{k\neq 0} a_k^N \frac{\sinh(k\pi\chi)}{k\pi \cosh(2k\pi)} \cos(k\pi y). \tag{2.13}$$

The discretization of this local Neumann problem is

$$\begin{array}{c} \\ 0/e_0 \\ 1/e_1 \\ 2/e_4 \\ 3/e_5 \\ 6/e_2 \\ 7/e_6 \end{array} \begin{array}{c} \begin{array}{cccccc} 0/e_0 & 1/e_1 & 2/e_4 & 3/e_5 & 6/e_2 & 7/e_6 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1.333 & 0 & -0.333 & -0.167 & -0.333 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & -0.333 & 0 & 1.333 & -0.333 & -0.167 \\ 0 & -0.167 & 0 & -0.333 & 0.667 & -0.167 \\ 0 & -0.333 & 0 & -0.167 & -0.167 & 0.667 \end{pmatrix} \end{array} \begin{pmatrix} u_0 \\ u_1 \\ u_4 \\ u_5 \\ u_2 \\ u_6 \end{pmatrix} = f_N(v_N), \tag{2.14}$$

where $f_N(v_N)$ is a right-hand side that depends on the value of v_N.

The addition of a similar Neumann BC $\mathcal{N}_1(u_1) = v_N$ on Γ in the generic equation (2.4) gives the local problem

$$\begin{array}{l} \text{find} \quad u_1 : \Omega_1 \to \mathbb{R} \quad \text{such that} \\ \begin{cases} \mathcal{L}(u_1) = f \text{ in } \Omega_1, \\ \mathcal{B}(u_1) = g \text{ on } \partial\Omega_1 \cap \partial\Omega, \\ \mathcal{N}_1(u_1) = v_N \text{ on } \Gamma, \end{cases} \end{array} \tag{2.15}$$

whose discretization defines a matrix $\mathcal{K}_1$

$$\mathcal{K}_1 \begin{pmatrix} u_{\mathcal{I}_1} \\ u_\Gamma \end{pmatrix} = \begin{pmatrix} \mathcal{K}_{\mathcal{I}_1,\mathcal{I}_1} & \mathcal{K}_{\mathcal{I}_1,\Gamma} \\ \mathcal{K}_{\Gamma,\mathcal{I}_1} & \mathcal{K}^{(1)}_{\Gamma,\Gamma} \end{pmatrix} \begin{pmatrix} u_{\mathcal{I}_1} \\ u_\Gamma \end{pmatrix} = f_N(v_N). \tag{2.16}$$

The matrix blocks $\mathcal{K}_{\mathcal{I}_1,\mathcal{I}_1}$, $\mathcal{K}_{\mathcal{I}_1,\Gamma}$ and $\mathcal{K}_{\Gamma,\mathcal{I}_1}$ are the same in the the three equations (2.7), (2.10) and (2.16), whereas the bottom-right block $\mathcal{K}^{(1)}_{\Gamma\Gamma}$ is different from $\mathcal{K}_{\Gamma\Gamma}$. However, with $\mathcal{K}_2$ and $\mathcal{K}^{(2)}_{\Gamma,\Gamma}$ defined similarly to $\mathcal{K}_1$ and $\mathcal{K}^{(1)}_{\Gamma,\Gamma}$, respectively, for the second subdomain Ω_2, Equation (2.5) implies that $\mathcal{K}^{(1)}_{\Gamma,\Gamma} + \mathcal{K}^{(2)}_{\Gamma,\Gamma} = \mathcal{K}_{\Gamma,\Gamma}$. Introducing a restriction matrix $\mathcal{R}_{\Omega_i}$ from global vectors defined on $(\mathcal{I}_1\ \mathcal{I}_2\ \Gamma)$ to local vectors defined on $(\mathcal{I}_i\ \Gamma)$

$$\mathcal{R}_{\Omega_1} = \begin{pmatrix} I & 0 & 0 \\ 0 & 0 & I \end{pmatrix}, \qquad \mathcal{R}_{\Omega_2} = \begin{pmatrix} 0 & I & 0 \\ 0 & 0 & I \end{pmatrix}, \tag{2.17}$$

it is then possible to retrieve the global matrix from the local Neumann matrices $\mathcal{K}_1$ and $\mathcal{K}_2$

$$\mathcal{K} = \mathcal{R}^T_{\Omega_1}\mathcal{K}_1\mathcal{R}_{\Omega_1} + \mathcal{R}^T_{\Omega_2}\mathcal{K}_2\mathcal{R}_{\Omega_2}. \tag{2.18}$$

The local subproblems with a Neumann BC on Γ can be seen as local contributions to the global problem. In practice, it is therefore possible to compute $\mathcal{K}$ in parallel by discretizing the local problems with a Neumann BC, and express $\mathcal{K}$ as a sum of these local matrices $\mathcal{K}_i$ as above.

2.3.3 Robin boundary condition on Γ

A third type of BC to consider is the Robin BC $\mathcal{N}_1(u_1) + pu_1 = v_R$ on Γ, where p is a (scalar) parameter:

$$\begin{aligned} &\text{find} \quad u_1 : \Omega_1 \to \mathbb{R} \quad \text{such that} \\ &\begin{cases} \mathcal{L}(u_1) = f \text{ in } \Omega_1, \\ \mathcal{B}(u_1) = g \text{ on } \partial\Omega_1 \cap \partial\Omega, \\ \mathcal{N}_1(u_1) + pu_1 = v_R \text{ on } \Gamma. \end{cases} \end{aligned} \tag{2.19}$$

The discretization gives

$$\begin{pmatrix} \mathcal{K}_{\mathcal{I}_1,\mathcal{I}_1} & \mathcal{K}_{\mathcal{I}_1,\Gamma} \\ \mathcal{K}_{\Gamma,\mathcal{I}_1} & \mathcal{K}^{(1)}_{\Gamma,\Gamma} + pM_\Gamma \end{pmatrix} \begin{pmatrix} u_{\mathcal{I}_1} \\ u_\Gamma \end{pmatrix} = f_R(v_R),$$

where M_Γ is a mass matrix (*i.e.*, a discretization of the identity) on Γ.

The Robin BC can be viewed as a linear combination of a Neumann and a Dirichlet BC. It can also be interpreted, instead, as a modification of a Neumann BC $\mathcal{N}_1(u_1) = v_N$ where v_N, instead of being fixed as a constant, has a linear dependence on the unknown $v_N = v_R - pu_1$. The pu_1 term can then be interpreted as a flow towards the other domain Ω_2, proportional to the value of u_1 on Γ, and p can be viewed as a mean to describe the conductivity of Ω_2.

This definition of a Robin BC can be extended by replacing the scalar multiplication $u \to pu$ by any linear operator $\mathcal{T}_i$, whose discretization on Γ is given by a matrix $\mathcal{T}_i$ called the *transmission matrix*

$$\begin{pmatrix} \mathcal{K}_{\mathcal{I}_1,\mathcal{I}_1} & \mathcal{K}_{\mathcal{I}_1,\Gamma} \\ \mathcal{K}_{\Gamma,\mathcal{I}_1} & \mathcal{K}^{(1)}_{\Gamma,\Gamma} + \mathcal{T}_1 \end{pmatrix} \begin{pmatrix} u_{\mathcal{I}_1} \\ u_\Gamma \end{pmatrix} = f_R(v_R). \tag{2.20}$$

The transmission operator $\mathcal{T}_i$ or matrix $\mathcal{T}_i$ represents the linear dependency between the flow from Ω_i to its neighbor and the solution on the interface Γ.

2.4 Optimal boundary conditions

In Section 2.3, we presented the local subproblem in a subdomain associated with a Dirichlet, Neumann or Robin BC on the interface Γ. An optimal choice for the BC on Γ would give a local subproblem in each subdomain Ω_i whose solution u_i is the restriction in Ω_i of the solution u of the global problem in Ω. Then, the global solution u could be obtained by solving the local subproblems instead of the global problem. For instance, in Ω_2, a comparison between equations (2.4 – 2.7) and (2.19 – 2.20) shows that the optimal value for $\mathcal{T}_2(u_2)$ should be equal to $\mathcal{N}_1(u_1)$, where u_1 is a solution of the local equations in Ω_1 with $u_1 = u_2$ on Γ. More precisely, if a Dirichlet BC $u_1 = u_2|_\Gamma$ on Γ is introduced for Ω_1, it is possible to solve this subdomain's local subproblem; the solution u_1 can then be expressed as a (linear) function of $u_2|_\Gamma$, and, it is possible to eliminate it from the global problem. This process, in turn, gives an optimal BC for Ω_2 that accounts for the effect of Ω_1 on Γ viewed by Ω_2. This process is detailed in a continuous setting in Section 2.4.1 and with algebraic notations in Section 2.4.2.

2.4.1 Dirichlet-to-Neumann operator for the PDE

In the example, if the value of u_2 on Γ is $u_2(2,\cdot) = \sum_k a_k^D \cos(k\pi y)$, the solution in Ω_1 of the Dirichlet problem (2.8) with $v_D = u_2(2,\cdot)$ is

$$u_1 = -1/2x^2 + x + a_0^D x/2 + \sum_{k\neq 0} a_k^D \frac{\sinh(k\pi x)}{\sinh(2k\pi)} \cos(k\pi y),$$

and its derivative across the interface is

$$\partial_x u_1(2,\cdot) = -1 + a_0^D/2 + \sum_{k\neq 0} a_k^D \, k\pi \, \coth(2k\pi) \cos(k\pi y).$$

We introduce the linear operator $\mathcal{S}_1$ such that $\partial_x u_1(2,\cdot) = -1 + \mathcal{S}_1(v_D)$

$$\mathcal{S}_1\left(\sum_k a_k^D \cos(k\pi y)\right) = a_0^D/2 + \sum_{k\neq 0} a_k^D \, k\pi \, \coth(2k\pi) \cos(k\pi y). \tag{2.21}$$

Then, $\partial_x u_1(2,\cdot) = -1 + \mathcal{S}_1(u_2(2,\cdot))$, and it is now possible to eliminate u_1 from the partitioned problem (2.1 – 2.3), leading to the following problem in Ω_2

$$\begin{array}{l} \text{find} \quad u_2 : \Omega_2 \to \mathbb{R} \quad \text{such that} \\ \left\{ \begin{array}{r} \Delta u_2 = -1 \text{ in } \Omega_2, \\ \partial_x u_2(3,\cdot) = 0, \\ \partial_y u_2(\cdot,0) = \partial_y u_2(\cdot,1) = 0, \\ \partial_x u_2(2,\cdot) - \mathcal{S}_1(u_2(2,\cdot)) = -1. \end{array} \right. \end{array} \tag{2.22}$$

The BC on Γ for Ω_2, deduced from the global problem, is a Robin BC whose transmission operator is applied by solving a Dirichlet problem in Ω_1. Note that the problem considered here for illustration is simplistic and in particular has no dependency on y. Therefore, in this particular case we have $a_k^D = 0$ for $k \neq 0$, which leads to $v_D = a_0^D$, and $\mathcal{S}_1(v_D) = v_D/2$. The analytic solution to Problem (2.22) is $u_2(x,y) = -\frac{1}{2}x^2 + 3x$ in Ω_2, which is of course equal to the solution of the global problem.

Following the same approach in the generic problem (2.4), one can introduce a Dirichlet BC $v_D : \Gamma \to \mathbb{R}$ on Γ for the subdomain Ω_1 and solve the local Dirichlet problem (2.9). Since $\mathcal{L}$ and $\mathcal{B}$ are linear, the solution u_1 depends linearly on (v_D, f, g). As a result, since $\mathcal{N}_1$ is linear too, there exist two linear operators $\mathcal{S}_1$ and $\mathcal{RHS}_1$ such that $\mathcal{N}_1(u_1) = \mathcal{S}_1(v_D) - \mathcal{RHS}_1(f,g)$. $\mathcal{S}_1$ is called the *Steklov-Poincaré Dirichlet-to-Neumann* operator of Ω_1. The name stems from the fact that the local harmonic problem in Ω_1 with a Dirichlet BC $u_1|_\Gamma = v_D$ and zero source term $f = 0$ and $g = 0$ gives the same solution as the problem with a Neumann BC $\mathcal{N}_1(u_1) = v_N$ on Γ with $f = 0$ and $g = 0$ if $v_N = \mathcal{S}_1(v_D)$. The reduced right-hand side on the interface is noted $rhs_1 = \mathcal{RHS}_1(f,g)$. Using this definition of $\mathcal{S}_1$ and rhs_1, one can eliminate u_1 from the global problem (2.4) and (2.5), and deduce a local

problem in Ω_2:

$$\text{find} \quad u_2 : \Omega_2 \to \mathbb{R} \quad \text{such that}$$
$$\begin{cases} \mathcal{L}(u_2) = f \text{ in } \Omega_2, \\ \mathcal{B}(u_2) = g \text{ on } \partial\Omega_2 \cap \partial\Omega, \\ \mathcal{N}_2(u_2) + \mathcal{S}_1(u_2) = rhs_1 \text{ on } \Gamma. \end{cases} \tag{2.23}$$

This problem, which gives the same solution in Ω_2 as the global problem (1.3) is a Robin problem in Ω_2, whose transmission condition $\mathcal{T}_2$ is given by the Dirichlet-to-Neumann $\mathcal{S}_1$ of Ω_1.

2.4.2 Partial Gaussian elimination and the Schur complement matrix

Performed on the linear system (2.6) or (2.7), this operation of eliminating Ω_1 from the global problem just corresponds to a few steps of *Gaussian Elimination*, as introduced in Section 1.1.4. By combining rows with one another, some coefficients in the matrix are *eliminated*, *i.e.*, become zeros. For instance, adding 0.25 times row 1 to row 3 from the matrix in (2.6) eliminates the coefficient in the second column from the matrix in (2.24). Performing a few steps of this algorithm, one can eliminate $\mathcal{I}_1$, and get the following system

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3{+}.25{\times}1 \\ 4 \\ 5 \\ 6{+}.2{\times}1{+}.3{\times}3 \\ 7{+}.3{\times}1{+}.2{\times}3 \end{array} \begin{array}{c} \begin{array}{cccc|cc|cc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \left(\begin{array}{cccc|cc|cc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1.33 & 0 & -0.333 & 0 & 0 & -0.167 & -0.333 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{0} & 0 & 1.25 & 0 & 0 & -0.375 & -0.25 \\ \hline 0 & 0 & 0 & 0 & 0.667 & -0.167 & -0.167 & -0.333 \\ 0 & 0 & 0 & 0 & -0.167 & 0.667 & -0.333 & -0.167 \\ \hline 0 & \mathbf{0} & 0 & \mathbf{0} & -0.167 & -0.333 & 1.2 & -0.45 \\ 0 & \mathbf{0} & 0 & \mathbf{0} & -0.333 & -0.167 & -0.45 & 1.2 \end{array}\right) \end{array} u = \begin{pmatrix} 0 \\ 0.5 \\ 0 \\ 0.62 \\ 0.25 \\ 0.25 \\ 0.75 \\ 0.75 \end{pmatrix}. \tag{2.24}$$

This way, the solution in $(\mathcal{I}_2\ \Gamma) = (3, 7, 2, 6)$ can be computed by solving the reduced system (4 last rows and columns of (2.24))

$$\begin{pmatrix} 0.667 & -0.167 & -0.167 & -0.333 \\ -0.167 & 0.667 & -0.333 & -0.167 \\ -0.167 & -0.333 & 1.2 & -0.45 \\ -0.333 & -0.167 & -0.45 & 1.2 \end{pmatrix} \begin{pmatrix} u_3 \\ u_7 \\ u_2 \\ u_6 \end{pmatrix} = \begin{pmatrix} 0.25 \\ 0.25 \\ 0.75 \\ 0.75 \end{pmatrix}.$$

This reduced system is called the *Schur complement system*.

Similarly, in the generic problem (2.7), the dependence of $u_{\mathcal{I}_1}$ on u_Γ can be expressed directly from the first row as $u_{\mathcal{I}_1} = \mathcal{K}_{\mathcal{I}_1,\mathcal{I}_1}^{-1}(f_{\mathcal{I}_1} - \mathcal{K}_{\mathcal{I}_1,\Gamma} u_\Gamma)$. Then, replacing $u_{\mathcal{I}_1}$ by this expression in the other rows gives an equivalent reduced system where $\mathcal{I}_1$ does not appear

anymore. This process is just a simple step of block-Gaussian elimination: subtracting $\mathcal{K}_{\Gamma,\mathcal{I}_1}\mathcal{K}_{\mathcal{I}_1,\mathcal{I}_1}^{-1}$ times the first row from the third row in (2.7) gives

$$\begin{pmatrix} \mathcal{K}_{\mathcal{I}_1,\mathcal{I}_1} & 0 & \mathcal{K}_{\mathcal{I}_1,\Gamma} \\ 0 & \mathcal{K}_{\mathcal{I}_2,\mathcal{I}_2} & \mathcal{K}_{\mathcal{I}_2,\Gamma} \\ 0 & \mathcal{K}_{\Gamma,\mathcal{I}_2} & \mathcal{K}_{\Gamma,\Gamma} - \mathcal{K}_{\Gamma,\mathcal{I}_1}\mathcal{K}_{\mathcal{I}_1,\mathcal{I}_1}^{-1}\mathcal{K}_{\mathcal{I}_1,\Gamma} \end{pmatrix} \begin{pmatrix} u_{\mathcal{I}_1} \\ u_{\mathcal{I}_2} \\ u_\Gamma \end{pmatrix} = \begin{pmatrix} f_{\mathcal{I}_1} \\ f_{\mathcal{I}_2} \\ f_\Gamma - \mathcal{K}_{\Gamma,\mathcal{I}_1}\mathcal{K}_{\mathcal{I}_1,\mathcal{I}_1}^{-1} f_{\mathcal{I}_1} \end{pmatrix},$$

and the solution in $\mathcal{I}_2 \cup \Gamma$ can be computed through the reduced system

$$\begin{pmatrix} \mathcal{K}_{\mathcal{I}_2,\mathcal{I}_2} & \mathcal{K}_{\mathcal{I}_2,\Gamma} \\ \mathcal{K}_{\Gamma,\mathcal{I}_2} & \mathcal{K}_{\Gamma,\Gamma} - \mathcal{K}_{\Gamma,\mathcal{I}_1}\mathcal{K}_{\mathcal{I}_1,\mathcal{I}_1}^{-1}\mathcal{K}_{\mathcal{I}_1,\Gamma} \end{pmatrix} \begin{pmatrix} u_{\mathcal{I}_2} \\ u_\Gamma \end{pmatrix} = \begin{pmatrix} f_{\mathcal{I}_2} \\ f_\Gamma - \mathcal{K}_{\Gamma,\mathcal{I}_1}\mathcal{K}_{\mathcal{I}_1,\mathcal{I}_1}^{-1} f_{\mathcal{I}_1} \end{pmatrix}. \tag{2.25}$$

Using the notations introduced in Section 2.3.2, one can define the local Schur complement matrix

$$\mathcal{S}_i = \mathcal{K}_{\Gamma,\Gamma}^{(i)} - \mathcal{K}_{\Gamma,\mathcal{I}_i}\mathcal{K}_{\mathcal{I}_i,\mathcal{I}_i}^{-1}\mathcal{K}_{\mathcal{I}_i,\Gamma},$$

and Equation (2.25) can be rewritten

$$\begin{pmatrix} \mathcal{K}_{\mathcal{I}_2,\mathcal{I}_2} & \mathcal{K}_{\mathcal{I}_2,\Gamma} \\ \mathcal{K}_{\Gamma,\mathcal{I}_2} & \mathcal{K}_{\Gamma,\Gamma}^{(2)} + \mathcal{S}_1 \end{pmatrix} \begin{pmatrix} u_{\mathcal{I}_2} \\ u_\Gamma \end{pmatrix} = \begin{pmatrix} f_{\mathcal{I}_2} \\ f_\Gamma - \mathcal{K}_{\Gamma,\mathcal{I}_1}\mathcal{K}_{\mathcal{I}_1,\mathcal{I}_1}^{-1} f_{\mathcal{I}_1} \end{pmatrix}, \tag{2.26}$$

and the Schur complement matrix $\mathcal{S}_1$ can be interpreted as a transmission matrix $\mathcal{T}_2$: the optimal BC is a (non-scalar) Robin BC whose transmission matrix is the local Schur complement of the outer domain. This BC is optimal in the sense that the local problem in Ω_2 with a Robin BC whose transmission matrix is $\mathcal{T}_2 = \mathcal{S}_1$ gives the same local solution as the global problem restricted to the subdomain Ω_2.

This Schur complement matrix $\mathcal{S}_i$ obtained by eliminating the interior unknown $u_{\mathcal{I}_i}$ from the system is a discrete version of the Dirichlet-to-Neumann operator presented above: given a Dirichlet BC v_D on Γ, the solution of the Dirichlet problem (2.10) with $f_{\mathcal{I}_i} = 0$ is $u_{\mathcal{I}_i} = -\mathcal{K}_{\mathcal{I}_i\mathcal{I}_i}^{-1}\mathcal{K}_{\mathcal{I}_i\Gamma}v_D$ in the interior and $u_\Gamma = v_D$ on Γ. Replacing $u_{\mathcal{I}_i}$ and u_Γ by this Dirichlet solution in the Neumann problem (2.16) gives a Neumann BC

$$f_N(v_N) = \begin{pmatrix} 0 \\ v_N \end{pmatrix}$$

with $v_N = \mathcal{K}_{\Gamma\mathcal{I}_i}u_{\mathcal{I}_i} + \mathcal{K}_{\Gamma_i\Gamma_i}^{(i)}u_\Gamma = \left(\mathcal{K}_{\Gamma_i\Gamma_i}^{(i)} - \mathcal{K}_{\Gamma\mathcal{I}_i}\mathcal{K}_{\mathcal{I}_i\mathcal{I}_i}^{-1}\mathcal{K}_{\mathcal{I}_i\Gamma}\right)v_D = \mathcal{S}_i v_D$.

2.5 Domain decomposition formulations

In this section, various reformulations of the coupled problem (2.4 – 2.5) are introduced. Each reformulation provides a different way of solving the same problem (1.3) through local subproblems. By changing the BC in the local subproblems, the compatibility conditions are modified accordingly and give rise to different approaches, both for the continuous PDE or for the algebraic linear system (Maday and Magoules, 2006). Imposing a Neumann or Robin BC in the local problems introduces variables λ_i on Γ. The different formulations can then be deduced from one another by eliminating, or not, the interior unknown using the Dirichlet-to-Neumann operator, and eliminating either the primal or dual variables u_i or λ_i on Γ. A tentative taxonomy is sketched in Figure 2.3, where both names and associated sections are given.

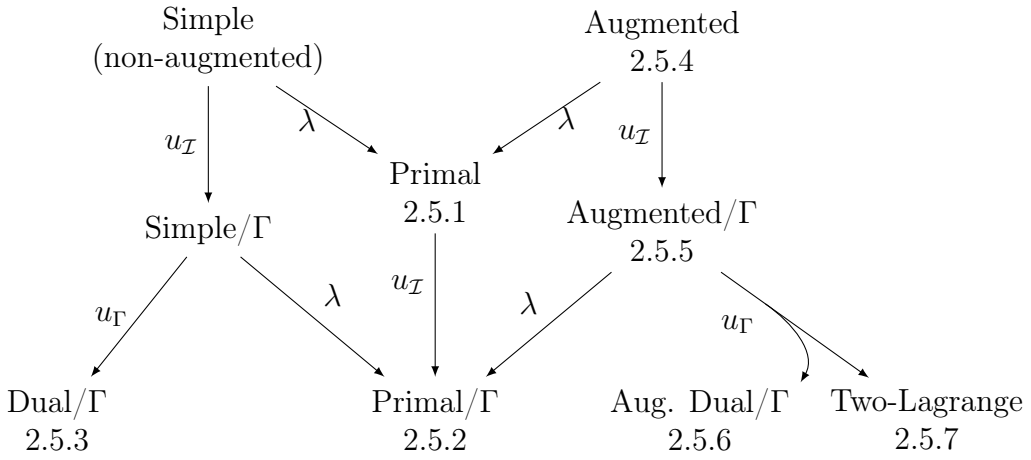


Figure 2.3: The label on each arrow represents which variable has to be eliminated to deduce formulations from one another. For each formulation, we also provide the number of the corresponding section.

2.5.1 Primal formulation

From the continuous coupled problem (2.4 – 2.5), it is possible to search for a unique global unknown u called the primal unknown: the condition $u_1 = u_2$ on Γ can be enforced by introducing a global unknown u such that $u_1 = u|_{\Omega_1}$ and $u_2 = u|_{\Omega_2}$, leading to the *primal formulation*

$$\text{find} \quad u : \Omega \to \mathbb{R} \text{ such that}$$
$$\begin{cases} \mathcal{L}(u) = f \text{ in } \Omega_1 \text{ and } \Omega_2 & (2.27) \\ \mathcal{B}(u) = g \text{ on } \partial\Omega_1 \cap \partial\Omega \text{ and } \partial\Omega_2 \cap \partial\Omega & (2.28) \\ \mathcal{N}_1(u) + \mathcal{N}_2(u) = 0 \text{ on } \Gamma. & (2.29) \end{cases}$$

Algebraically, for any decomposition $\mathcal{K}_{\Gamma\Gamma} = \mathcal{K}_{\Gamma\Gamma}^{(1)} + \mathcal{K}_{\Gamma\Gamma}^{(2)}$ and $f_\Gamma = f_\Gamma^{(1)} + f_\Gamma^{(2)}$, the global matrix $\mathcal{K}$ in (1.5) can be expressed as the sum of two local matrices $\mathcal{K}_1$ and $\mathcal{K}_2$

$$\left(\mathcal{R}_{\Omega_1}^T \mathcal{K}_1 \mathcal{R}_{\Omega_1} + \mathcal{R}_{\Omega_2}^T \mathcal{K}_2 \mathcal{R}_{\Omega_2}\right) u = \mathcal{R}_{\Omega_1}^T f_1 + \mathcal{R}_{\Omega_2}^T f_2, \tag{2.30}$$

where the restriction matrices $\mathcal{R}_{\Omega_i}$ are as defined in Equation (2.17), and

$$\mathcal{K}_i = \begin{pmatrix} \mathcal{K}_{\mathcal{I}_i\mathcal{I}_i} & \mathcal{K}_{\mathcal{I}_i,\Gamma} \\ \mathcal{K}_{\Gamma,\mathcal{I}_i} & \mathcal{K}_{\Gamma\Gamma}^{(i)} \end{pmatrix} \qquad \text{and} \qquad f_i = \begin{pmatrix} f_{\mathcal{I}_i} \\ f_\Gamma^{(i)} \end{pmatrix}.$$

Any decomposition $\mathcal{K}_{\Gamma\Gamma}^{(1)} + \mathcal{K}_{\Gamma\Gamma}^{(2)} = \mathcal{K}_{\Gamma\Gamma}$ can be used to build matrices $\mathcal{K}_1$ and $\mathcal{K}_2$: in an algebraic solver, one can, for instance, choose $\mathcal{K}_{\Gamma\Gamma}^{(1)} = \mathcal{K}_{\Gamma\Gamma}$ and $\mathcal{K}_{\Gamma\Gamma}^{(2)} = 0$. However, a case of particular interest is when $\mathcal{K}_{\Gamma\Gamma}^{(i)}$ is obtained by choosing $\mathcal{K}_i$ as the discretization of the local Neumann problem as in Section 2.3.2: this will make it easier to ensure some properties needed to build a robust coarse space correction in sections 2.7.3 and 3.

This primal formulation is exactly the same as the global formulation (1.3) or (1.5), except that the application of the linear operator corresponding to the PDE in the continuous case, or the matrix-vector product in the algebraic case, can be performed as a sum

of two local components. Each local component can be computed independently in each subdomain, and the global solution can therefore be computed using a parallel iterative method, whose convergence depends on the choice of a good preconditioner as presented in Section 2.7.

In combination with well-chosen preconditioners (see Section 2.7), this approach is used, in the Additive Schwarz method and some of its variations (Lions, 1988, 1989, 1990; Dryja and Widlund, 1987; Cai and Sarkis, 1999).

2.5.2 Primal formulation on the interface

Another approach to enforce the first compatibility condition $u_1 = u_2$ on Γ is to introduce a unique unknown $u_\Gamma : \Gamma \to \mathbb{R}$ that is the Dirichlet BC $u_i = u_\Gamma$ on Γ for both subdomains Ω_i. The unknowns u_1 and u_2 can be eliminated from the system by expressing them as the solutions of the local Dirichlet problems

$$\text{find } u_1 : \Omega_1 \to \mathbb{R} \text{ such that} \quad \begin{cases} \mathcal{L}(u_1) = f \text{ in } \Omega_1, \\ \mathcal{B}(u_1) = g \text{ on } \partial\Omega_1 \cap \partial\Omega, \\ u_1 = u_\Gamma \text{ on } \Gamma; \end{cases}$$

$$\text{and } u_2 : \Omega_2 \to \mathbb{R} \text{ such that} \quad \begin{cases} \mathcal{L}(u_2) = f \text{ in } \Omega_2, \\ \mathcal{B}(u_2) = g \text{ on } \partial\Omega_2 \cap \partial\Omega, \\ u_2 = u_\Gamma \text{ on } \Gamma. \end{cases}$$

Using the Dirichlet-to-Neumann operator $\mathcal{S}_i$ introduced in Section 2.4, the second compatibility condition $\mathcal{N}_1(u_1) + \mathcal{N}_2(u_2) = 0$ on Γ can be expressed as an interface problem for u_Γ, yielding the *primal formulation on the interface*

$$\begin{aligned} &\text{find } u_\Gamma : \Gamma \to \mathbb{R} \text{ such that} \\ &\mathcal{S}_1(u_\Gamma) + \mathcal{S}_2(u_\Gamma) = rhs_1 + rhs_2. \end{aligned}$$

Once this continuous problem is solved, the solution in Ω is deduced by solving the local problems in Ω_1 and Ω_2 with u_Γ as a Dirichlet BC on Γ.

Algebraically, eliminating the interior blocks in Equation (2.30) gives

$$\begin{aligned} (\mathcal{S}_1 + \mathcal{S}_2) u_\Gamma &= \widetilde{f}_\Gamma^{(1)} + \widetilde{f}_\Gamma^{(2)} \\ \mathcal{S} u_\Gamma &= \widetilde{f}_\Gamma, \end{aligned} \tag{2.31}$$

where $\mathcal{S}_i = \mathcal{K}_{\Gamma,\Gamma}^{(i)} - \mathcal{K}_{\Gamma,\mathcal{I}_i} \mathcal{K}_{\mathcal{I}_i,\mathcal{I}_i}^{-1} \mathcal{K}_{\mathcal{I}_i,\Gamma}$ and $\widetilde{f}_\Gamma^{(i)} = f_\Gamma^{(i)} - \mathcal{K}_{\Gamma,\mathcal{I}_i} \mathcal{K}_{\mathcal{I}_i,\mathcal{I}_i}^{-1} f_{\mathcal{I}_i}$. Once (2.31) is solved, the solution in the interior $\mathcal{I}_i$ is deduced from $u_{\mathcal{I}_i} = \mathcal{K}_{\mathcal{I}_i\mathcal{I}_i}^{-1}(f_{\mathcal{I}_i} - \mathcal{K}_{\mathcal{I}_i\Gamma_i} u_\Gamma)$.

This approach is often referred to as a (primal) substructuring approach. This approach is used, for instance, in the Neumann-Neumann method (De Roeck and Le Tallec, 1991; Mandel, 1993), in the Robin-Robin method (Achdou and Nataf, 1997; Achdou et al., 2000) or in the Additive Schwarz on the Schur method (Carvalho et al., 2001b), among many others.

2.5.3 Dual formulation on the interface

In the continuous problem (2.4 – 2.5), instead of enforcing the first continuity equation $u_1 = u_2$ on Γ and solve for u or u_Γ as in the primal formulations, it is possible to enforce

the second equation $\mathcal{N}_1(u_1) + \mathcal{N}_2(u_2) = 0$ by adding a Neumann BC $\mathcal{N}_1(u_1) = \lambda$ and $\mathcal{N}_2(u_2) = -\lambda$ on Γ for each subdomain

$$\text{find } u_1 : \Omega_1 \to \mathbb{R} \text{ such that} \quad \begin{cases} \mathcal{L}(u_1) = f \text{ in } \Omega_1, \\ \mathcal{B}(u_1) = g \text{ on } \partial\Omega_1 \cap \partial\Omega, \\ \mathcal{N}_1(u_1) = \lambda \text{ on } \Gamma; \end{cases}$$

$$\text{and } u_2 : \Omega_2 \to \mathbb{R} \text{ such that} \quad \begin{cases} \mathcal{L}(u_2) = f \text{ in } \Omega_2, \\ \mathcal{B}(u_2) = g \text{ on } \partial\Omega_2 \cap \partial\Omega, \\ \mathcal{N}_2(u_2) = -\lambda \text{ on } \Gamma; \end{cases}$$

and solve for the dual variable λ. It holds that $\mathcal{N}_i(u_i) = \pm\lambda = \mathcal{S}_i(u_i) - rhs_i$ and if the Dirichlet-to-Neumann operator is not singular, the primal unknown u_i can be eliminated from the first continuity equation $u_1 = u_2$ using $u_i = \mathcal{S}_i^{-1}(\pm\lambda + rhs_i)$. The resulting equation gives a *dual formulation*

$$\begin{aligned} &\text{find } \lambda : \Gamma \to \mathbb{R} \text{ such that} \\ &\mathcal{S}_1^{-1}(\lambda) + \mathcal{S}_2^{-1}(\lambda) = -\mathcal{S}_1^{-1}(rhs_1) + \mathcal{S}_2^{-1}(rhs_2). \end{aligned} \tag{2.32}$$

Once (2.32) is solved, the primal solution u_i can then be recovered from λ by solving the local Neumann problems above in each subdomain.

Algebraically, with the same decomposition as in Section 2.5.2, one can introduce $\lambda = \mathcal{S}_1 u_\Gamma - \widetilde{f}_\Gamma^{(1)}$, such that $u_\Gamma = \mathcal{S}_1^{-1}(\lambda + \widetilde{f}_\Gamma^{(1)})$. Then Equation (2.31) is equivalent to $\lambda + \mathcal{S}_2 u_\Gamma = \widetilde{f}_\Gamma^{(2)}$, or $u_\Gamma = \mathcal{S}_2^{-1}(\widetilde{f}_\Gamma^{(2)} - \lambda)$, yielding the following problem on λ

$$(\mathcal{S}_1^{-1} + \mathcal{S}_2^{-1})\lambda = -\mathcal{S}_1^{-1}\widetilde{f}_\Gamma^{(1)} + \mathcal{S}_2^{-1}\widetilde{f}_\Gamma^{(2)}.$$

From the solution λ, the solution can be computed as $u_\Gamma = \mathcal{S}_1^{-1}(\lambda + \widetilde{f}_\Gamma^{(1)})$ and $u_{\mathcal{I}_i} = \mathcal{K}_{\mathcal{I}_i\mathcal{I}_i}^{-1}(f_{\mathcal{I}_i} - \mathcal{K}_{\mathcal{I}_i\Gamma_i} u_\Gamma)$.
When $\mathcal{S}_i$ is singular, it is not possible to solve $\mathcal{S}_i^{-1}\lambda$. Using a pseudo-inverse instead of the inverse, the quantity $\mathcal{S}_i^\dagger\lambda$ can be defined, but only up to an element of $\ker \mathcal{S}_i$. This is one of the motivations for using a two-level method as introduced in Section 2.7.3. Another solution is to replace the Neumann BC by a Robin BC, as proposed in the next section.

Note that the dual formulation can only be used on the interface: although the primal variable u also exists inside the subdomains, the dual variable λ expresses a BC and, as such, only exists on Γ. The dual approach is at the core of the FETI method (Farhat and Roux, 1991).

2.5.4 Augmented formulation

In the *augmented formulation* here and the *augmented formulation on the interface* below, the BC is changed and neither the primal nor the dual unknown is eliminated. Their memory footprint would therefore be larger than the other formulations, if they were to be used in practice. They are only introduced here as intermediate formulations from which other formulations will be deduced in the following sections by eliminating some unknowns.

In a continuous setting, instead of a Dirichlet BC (as in the primal formulations) or a Neumann BC (as in the dual formulation), one may choose a Robin BC on Γ $\mathcal{N}_i(u_i) +$

$\mathcal{T}_i(u_i) = \lambda_i$ to the local problem. This introduces two operators $\mathcal{T}_i$, which can be chosen arbitrarily, and two functions λ_i, which are new dual unknowns in the problem. This changes the compatibility conditions of the coupled problem, yielding the *augmented formulation*

find $u_1 : \Omega_1 \to \mathbb{R}$, $\lambda_1 : \Gamma \to \mathbb{R}$ s. t.

$$\begin{cases} \mathcal{L}(u_1) = f \text{ in } \Omega_1, \\ \mathcal{B}(u_1) = g \text{ on } \partial\Omega_1 \cap \partial\Omega, \\ \mathcal{N}_1(u_1) + \mathcal{T}_1(u_1) = \lambda_1 \text{ on } \Gamma; \end{cases}$$

and $u_2 : \Omega_2 \to \mathbb{R}$, $\lambda_2 : \Gamma \to \mathbb{R}$ s. t.

$$\begin{cases} \mathcal{L}(u_2) = f \text{ in } \Omega_2, & (2.33) \\ \mathcal{B}(u_2) = g \text{ on } \partial\Omega_2 \cap \partial\Omega, & (2.34) \\ \mathcal{N}_2(u_2) + \mathcal{T}_2(u_2) = \lambda_2 \text{ on } \Gamma; & (2.35) \end{cases}$$

with the compatibility condition

$$\begin{cases} u_1 = u_2 \text{ on } \Gamma, & (2.36) \\ \mathcal{T}_1(u_1) + \mathcal{T}_2(u_2) = \lambda_1 + \lambda_2 \text{ on } \Gamma. & (2.37) \end{cases}$$

Algebraically, this augmented formulation becomes

$$\begin{cases} \begin{pmatrix} K_{\mathcal{I}_i,\mathcal{I}_i} & K_{\mathcal{I}_i,\Gamma} \\ K_{\Gamma,\mathcal{I}_i} & K^{(i)}_{\Gamma,\Gamma} + T_i \end{pmatrix} \begin{pmatrix} u_{\mathcal{I}_i} \\ u_{\Gamma_i} \end{pmatrix} = \begin{pmatrix} f_{\mathcal{I}_i} \\ f^{(i)}_{\Gamma} + \lambda_i \end{pmatrix}, & (2.38) \\ u_{\Gamma_1} = u_{\Gamma_2}, & (2.39) \\ \mathcal{T}_1 u_{\Gamma_1} + \mathcal{T}_2 u_{\Gamma_2} = \lambda_1 + \lambda_2. & (2.40) \end{cases}$$

The choice of a good transmission operator or matrix is an active research subject, see for instance Dolean et al. (2015a); Gander et al. (2002); St-Cyr et al. (2007) and references therein.

2.5.5 Augmented formulation on the interface

The augmented formulation (2.33 – 2.37) can be expressed as an interface problem by using the definition of $\mathcal{S}_i$ and rhs_i, as in Section 2.5.2. Introducing a *Dirichlet-to-Robin* operator $\widehat{\mathcal{S}}_i = \mathcal{S}_i + \mathcal{T}_i$, the elimination of interior variables gives an *augmented formulation on the interface*

find $u_1, u_2, \lambda_1, \lambda_2 : \Gamma \to \mathbb{R}$ such that

$$\begin{cases} \widehat{\mathcal{S}}_i(u_i) = rhs_i + \lambda_i \text{ on } \Gamma, & (2.41) \\ u_1 = u_2 \text{ on } \Gamma, & (2.42) \\ \mathcal{T}_1(u_1) + \mathcal{T}_2(u_2) = \lambda_1 + \lambda_2 \text{ on } \Gamma. & (2.43) \end{cases}$$

Similarly, introducing $\widehat{S}_i = S_i + T_i$, the algebraic formulation is

$$\begin{cases} \widehat{S}_i u_{\Gamma_i} = \widetilde{f}^{(i)}_{\Gamma} + \lambda_i, & (2.44) \\ u_{\Gamma_1} = u_{\Gamma_2}, & (2.45) \\ T_1 u_{\Gamma_1} + T_2 u_{\Gamma_2} = \lambda_1 + \lambda_2. & (2.46) \end{cases}$$

From these augmented formulations (on the interface or not), one could derive *primal augmented formulations* by eliminating λ_i using (2.35), (2.38), (2.41), or (2.44) in the

second compatibility condition (2.37), (2.40), (2.43), or (2.46), respectively. However, this also eliminates the $\mathcal{T}_i(u_i)$ and $\mathcal{T}_i u_i$ terms, and the resulting system is exactly the same as in the primal formulations in Section 2.5.1 or 2.5.2. On the other hand, this is not the case when eliminating u_i instead and solving for λ_i in a dual formulation as in Section 2.5.3. The transmission operator $\mathcal{T}_i$ and the transmission matrix $\mathcal{T}_i$ are supposed to be chosen such that $\widehat{\mathcal{S}}_i = \mathcal{S}_i + \mathcal{T}_i$ and $\widehat{\mathcal{S}}_i = \mathcal{S}_i + \mathcal{T}_i$ are not singular, so that the primal variable on Γ can be eliminated using $u_i = \widehat{\mathcal{S}}_i^{-1}(rhs_i + \lambda_i)$ and $u_{\Gamma_i} = \widehat{\mathcal{S}}_i^{-1}(\widetilde{f}_\Gamma^{(i)} + \lambda_i)$.

2.5.6 Dual augmented formulation on the interface ($\mathcal{T}_1 + \mathcal{T}_2 = 0$)

A case of particular interest is when $\mathcal{T}_1 + \mathcal{T}_2 = 0$. Under this condition, combining the continuous equations $[(2.43) - \mathcal{T}_1(2.42)]$ gives $\lambda_1 + \lambda_2 = (\mathcal{T}_1 + \mathcal{T}_2)(u_2) = 0$ on Γ. Introducing $\lambda = \lambda_1 = -\lambda_2$, it holds from (2.41) that $u_1 = \widehat{\mathcal{S}}_1^{-1}(rhs_1 + \lambda)$ and $u_2 = \widehat{\mathcal{S}}_2^{-1}(rhs_2 - \lambda)$. Replacing u_1 and u_2 in (2.42) gives a *Dual augmented formulation*

$$\begin{aligned} &\text{find} \quad \lambda : \Gamma \to \mathbb{R} \text{ such that} \\ &\left(\widehat{\mathcal{S}}_1^{-1} + \widehat{\mathcal{S}}_2^{-1}\right)(\lambda) = -\widehat{\mathcal{S}}_1^{-1}(rhs_1) + \widehat{\mathcal{S}}_2^{-1}(rhs_2); \end{aligned}$$

or, algebraically

$$\left(\widehat{\mathcal{S}}_1^{-1} + \widehat{\mathcal{S}}_2^{-1}\right)\lambda = -\widehat{\mathcal{S}}_1^{-1}\widetilde{f}_\Gamma^{(1)} + \widehat{\mathcal{S}}_2^{-1}\widetilde{f}_\Gamma^{(2)}.$$

This is, algebraically, a dual (non-augmented) formulation, as presented in Section 2.5.3, with a different splitting for the interface matrix $\mathcal{K}_{\Gamma\Gamma} = (\mathcal{K}_{\Gamma\Gamma}^{(1)} + \mathcal{T}) + (\mathcal{K}_{\Gamma\Gamma}^{(2)} - \mathcal{T})$ instead of $\mathcal{K}_{\Gamma\Gamma} = \mathcal{K}_{\Gamma\Gamma}^{(1)} + \mathcal{K}_{\Gamma\Gamma}^{(2)}$ with $\mathcal{T} = \mathcal{T}_1 = -\mathcal{T}_2$. Some discussion on how to choose $\mathcal{T}$ such that $\widehat{\mathcal{S}}_i$ is not singular is given in (Farhat et al., 1993).

2.5.7 Two-Lagrange formulation

In the general case, one can still eliminate $u_i = (\mathcal{S}_i + \mathcal{T}_i)^{-1}(rhs_i + \lambda_i)$ on Γ in equations (2.42) and (2.43) using Equation (2.41). The continuous system becomes

$$\begin{cases} (\mathcal{S}_1 + \mathcal{T}_1)^{-1}(rhs_1 + \lambda_1) = (\mathcal{S}_2 + \mathcal{T}_2)^{-1}(rhs_2 + \lambda_2), \\ \mathcal{T}_1(\mathcal{S}_1 + \mathcal{T}_1)^{-1}(rhs_1 + \lambda_1) + \mathcal{T}_2(\mathcal{S}_2 + \mathcal{T}_2)^{-1}(rhs_2 + \lambda_2) = \lambda_1 + \lambda_2. \end{cases}$$

Introducing $\widehat{\mathcal{S}}_i = \mathcal{S}_i + \mathcal{T}_i$ and I the identity operator $I(\lambda_i) = \lambda_i$, the previous system can be rewritten as

$$\begin{cases} \widehat{\mathcal{S}}_1^{-1}\lambda_1 - \widehat{\mathcal{S}}_2^{-1}\lambda_2 = -\widehat{\mathcal{S}}_1^{-1}(rhs_1) + \widehat{\mathcal{S}}_2^{-1}(rhs_2), & (2.47) \\ [\mathcal{T}_1\widehat{\mathcal{S}}_1^{-1} - I]\lambda_1 + [\mathcal{T}_2\widehat{\mathcal{S}}_2^{-1} - I]\lambda_2 = -\mathcal{T}_1\widehat{\mathcal{S}}_1^{-1}(rhs_1) - \mathcal{T}_2\widehat{\mathcal{S}}_2^{-1}(rhs_2). & (2.48) \end{cases}$$

Then, if $\mathcal{T}_1 + \mathcal{T}_2$ is not singular, replacing the two lines (2.47) and (2.48) by two independent linear combination theereof $[\mathcal{T}_1(2.47) - (2.48)]$ and $[-\mathcal{T}_2(2.47) - (2.48)]$ gives the *Two-Lagrange formulation*

$$\begin{cases} \lambda_1 + [I - (\mathcal{T}_1 + \mathcal{T}_2)\widehat{\mathcal{S}}_2^{-1}]\lambda_2 = (\mathcal{T}_1 + \mathcal{T}_2)\widehat{\mathcal{S}}_2^{-1}]rhs_2, & (2.49) \\ [I - (\mathcal{T}_1 + \mathcal{T}_2)\widehat{\mathcal{S}}_1^{-1}]\lambda_1 + \lambda_2 = (\mathcal{T}_1 + \mathcal{T}_2)\widehat{\mathcal{S}}_1^{-1}]rhs_1. & (2.50) \end{cases}$$

Once this system is solved, the solution u_i can be computed from λ_i by solving the Robin local problem (2.33 – 2.35).

Algebraically, introducing $\widehat{\mathcal{S}}_i = \mathcal{S}_i + \mathcal{T}_i$, the Two-Lagrange formulation becomes

$$\begin{pmatrix} I & I - (\mathcal{T}_1 + \mathcal{T}_2)\widehat{\mathcal{S}}_2^{-1} \\ I - (\mathcal{T}_1 + \mathcal{T}_2)\widehat{\mathcal{S}}_1^{-1} & I \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \lambda_2 \end{pmatrix} = \begin{pmatrix} (\mathcal{T}_1 + \mathcal{T}_2)\widehat{\mathcal{S}}_2^{-1} \widetilde{f}_\Gamma^{(2)} \\ (\mathcal{T}_1 + \mathcal{T}_2)\widehat{\mathcal{S}}_1^{-1} \widetilde{f}_\Gamma^{(1)} \end{pmatrix},$$

with $u_\Gamma = \widehat{\mathcal{S}}_1^{-1}(\lambda_1 + \widetilde{f}_\Gamma^{(1)})$ and $u_{\mathcal{I}_i} = \mathcal{K}_{\mathcal{I}_i\mathcal{I}_i}^{-1}(f_{\mathcal{I}_i} - \mathcal{K}_{\mathcal{I}_i\Gamma_i} u_\Gamma)$.

This method, introduced by Nataf et al. (1995), is applied, for instance, in (Gander et al., 2002) and (El Bouajaji et al., 2015).

2.6 Generalization to N subdomains

The various formulations presented above are generalized to the case of a domain decomposition with N subdomains. The augmented formulations are first generalized in sections 2.6.1 and 2.6.2. Eliminating the dual variable λ_i gives the primal formulations in Section 2.6.3, while eliminating the primal variable u_i or u_i gives the dual formulations in Section 2.6.4 and the N-Lagrange formulations in Section 2.6.5. As in the previous section (Figure 2.3), the different formulations are summarized in Figure 2.4. Hybrid approaches where dual and primal variables are eliminated in different parts of the interface, such as in the FETI-DP (Farhat et al., 2001) and BDDC methods (Dohrmann, 2003), are not addressed in this thesis.

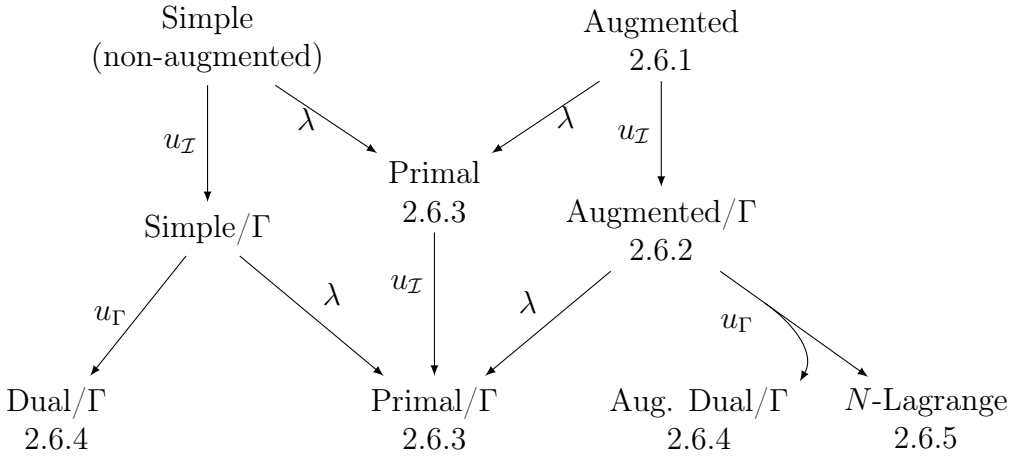


Figure 2.4: The label on each arrow represents which variable has to be eliminated to deduce formulations from one another. For each formulation, we also provide the number of the corresponding section in the N subdomains case.

2.6.1 Augmented formulation

The various formulations presented in the previous section can all be generalized to a N-subdomain decomposition: Ω is decomposed into a family of N subdomains $(\Omega_i)_{1\leq i\leq N}$

without overlap, such that

$$\bar{\Omega} = \bigcup_{1 \leq i \leq N} \bar{\Omega}_i \quad \text{and} \quad \Omega_i \cap \Omega_j = \emptyset \ \text{ if } \ i \neq j.$$

Then the local interfaces Γ_i and the global interface Γ, which is the non-disjoint union of the local interfaces, are defined as

$$\Gamma_i = \partial\Omega_i \setminus \partial\Omega \quad \text{and} \quad \Gamma = \bigcup_{1 \leq i \leq N} \Gamma_i.$$

Two subdomains Ω_i and Ω_j are said to be neighbors if their shared interface $\Gamma_{ij} = \Gamma_i \cap \Gamma_j$ is not empty. More than two subdomains Ω_i, Ω_j, ..., and Ω_k can in fact share an interface $\Gamma_{ij\dots k} = \Gamma_i \cap \Gamma_j \cap \cdots \cap \Gamma_k \neq \emptyset$, *e.g.*, crosspoints in 2D, or crosspoints and edges in 3D. Introducing for each subdomain a transmission operator $\mathcal{T}_i$ on Γ_i, the global continuous problem (1.3) is equivalent to the set of local Robin problems

$$\begin{aligned} &\forall i \in \{1, \dots, N\} \ \text{ find } \ (u_i, \lambda_i) : \Omega_i \to \mathbb{R} \text{ such that} \\ &\begin{cases} \mathcal{L}(u_i) = f \text{ in } \Omega_i, \\ \mathcal{B}(u_i) = g \text{ on } \partial\Omega_i \cap \partial\Omega, \\ \mathcal{N}_i(u_i) + \mathcal{T}_i(u_i) = \lambda_i \text{ on } \Gamma_i; \end{cases} \end{aligned} \tag{2.51}$$

with the compatibility condition

$$\begin{cases} u_i = u_j = \cdots = u_k \text{ on } \Gamma_{ij\dots k}, \\ \mathcal{T}_i(u_i) + \mathcal{T}_j(u_j) + \cdots + \mathcal{T}_k(u_k) = \lambda_i + \lambda_j + \cdots + \lambda_k \text{ on } \Gamma_{ij\dots k}. \end{cases} \tag{2.52}$$

A similar augmented formulation can be expressed algebraically in two different ways, by performing the domain decomposition either *before* or *after* the discretization process. First, the option of performing an algebraic domain decomposition on the global matrix $\mathcal{K}$ *after* the discretization process is described. Using a graph partitioner such as Scotch (Chevalier and Pellegrini, 2008) or Metis (Karypis and Kumar, 2009), one can partition the set of indices Ω into N *interiors* $\mathcal{I}_i$ and a *global interface* Γ such that $\mathcal{K}_{\ell m} = \mathcal{K}_{m\ell} = 0$ for any $\ell \in \mathcal{I}_i$ and $m \in \mathcal{I}_j$ that belong to different interiors $\mathcal{I}_i$ and $\mathcal{I}_j$ with $i \neq j$. Reordering the global matrix using the symmetric permutation $(\mathcal{I}_1 \ \dots \ \mathcal{I}_N \ \Gamma)$ gives

$$\begin{pmatrix} \mathcal{K}_{\mathcal{I}_1,\mathcal{I}_1} & 0 & \dots & 0 & \mathcal{K}_{\mathcal{I}_1,\Gamma} \\ 0 & \mathcal{K}_{\mathcal{I}_2,\mathcal{I}_2} & \ddots & \vdots & \mathcal{K}_{\mathcal{I}_2,\Gamma} \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \cdots & 0 & \mathcal{K}_{\mathcal{I}_N,\mathcal{I}_N} & \mathcal{K}_{\mathcal{I}_N,\Gamma} \\ \mathcal{K}_{\Gamma,\mathcal{I}_1} & \mathcal{K}_{\Gamma,\mathcal{I}_2} & \cdots & \mathcal{K}_{\Gamma,\mathcal{I}_N} & \mathcal{K}_{\Gamma,\Gamma} \end{pmatrix} \begin{pmatrix} u_{\mathcal{I}_1} \\ u_{\mathcal{I}_2} \\ \vdots \\ u_{\mathcal{I}_N} \\ u_\Gamma \end{pmatrix} = \begin{pmatrix} f_{\mathcal{I}_1} \\ f_{\mathcal{I}_2} \\ \vdots \\ f_{\mathcal{I}_N} \\ f_\Gamma \end{pmatrix}. \tag{2.53}$$

Each unknown in Γ is then assigned to the *local interfaces* Γ_i, Γ_j, ... of the closest (w.r.t. the graph distance) interiors $\mathcal{I}_i$, $\mathcal{I}_j$, ... (each unknown in Γ may then belong to the local interface of several subdomains). As such, Γ is equal to the (non-disjoint) union of subdomain's interfaces $\Gamma = \bigcup_{1 \leq i \leq N} \Gamma_i$. Introducing the two sets of canonical restriction

matrices $\mathcal{R}_{\Omega_i}$ from the global set of indices Ω to $\Omega_i = \mathcal{I}_i \cup \Gamma_i$, and $\mathcal{R}_{\Gamma_i}$ from Γ to Γ_i, one can partition $\mathcal{K}$ by choosing a decomposition of the interface matrix and of the right-hand side

$$\mathcal{K}_{\Gamma\Gamma} = \sum_{i=1}^{N} \mathcal{R}_{\Gamma_i}^T \mathcal{K}_{\Gamma_i\Gamma_i}^{(i)} \mathcal{R}_{\Gamma_i} \qquad \text{and} \qquad f_\Gamma = \sum_{i=1}^{N} \mathcal{R}_{\Gamma_i}^T f_{\Gamma_i}^{(i)}. \tag{2.54}$$

Then, introducing a transmission matrix $\mathcal{T}_i$ on Γ_i, local matrices $\mathcal{K}_i$ and local right-hand side f_i as

$$\mathcal{K}_i = \begin{pmatrix} \mathcal{K}_{\mathcal{I}_i\mathcal{I}_i} & \mathcal{K}_{\mathcal{I}_i\Gamma_i} \\ \mathcal{K}_{\Gamma_i\mathcal{I}_i} & \mathcal{K}_{\Gamma_i\Gamma_i}^{(i)} \end{pmatrix} \qquad \text{and} \qquad f_i = \begin{pmatrix} f_{\mathcal{I}_i} \\ f_{\Gamma_i}^{(i)} \end{pmatrix}, \tag{2.55}$$

one gets the algebraic augmented formulation, where the unknowns on Γ_i are replicated on all the neighbors that share Γ_i

$$\begin{cases} \forall i \in \{1, \ldots, N\} \quad \left[\mathcal{K}_i + \begin{pmatrix} 0 & 0 \\ 0 & \mathcal{T}_i \end{pmatrix} \right] \begin{pmatrix} u_{\mathcal{I}_i} \\ u_{\Gamma_i} \end{pmatrix} = f_i + \begin{pmatrix} 0 \\ \lambda_i \end{pmatrix}, & (2.56) \\ \displaystyle\sum_{i=1}^{N} \mathcal{B}_i u_{\Gamma_i} = 0, & (2.57) \\ \displaystyle\sum_{i=1}^{N} \mathcal{R}_{\Gamma_i}^T \mathcal{T}_i u_{\Gamma_i} = \sum_{i=1}^{N} \mathcal{R}_{\Gamma_i}^T \lambda_i, & (2.58) \end{cases}$$

where $\mathcal{B}_i$ is a *jump matrix*: each row in $\sum_{i=1}^{N} \mathcal{B}_i$ corresponds to the equality condition for one unknown shared by two subdomains $k < \ell$. The row in $\mathcal{B}_i$ has only one non-zero element on the column of the corresponding shared unknown if $i = k$, in which case the value is 1, or if $i = \ell$ in which case it is -1. All the other elements are 0. For instance, with only two subdomains, the constraint $u_{\Gamma_1} = u_{\Gamma_2}$ is ensured by choosing $\mathcal{B}_1 = I$ and $\mathcal{B}_2 = -I$. On an interface $\Gamma_{ij\ldots k}$ between Ω_i and a certain number of subdomains, several rows in $\mathcal{B}_i$ are needed for each unknown to enforce the equality between the different subdomains. Namely, for an unknown shared by n subdomains, $n-1$ rows are needed.

The global formulation (2.53) and the augmented formulation (2.56 – 2.58) are equivalent: from a solution u to (2.53) one can introduce $u_{\Gamma_i} = \mathcal{R}_{\Gamma_i} u_\Gamma$ and $\lambda_i = \mathcal{K}_{\Gamma_i\mathcal{I}_i} u_{\mathcal{I}_i} + (\mathcal{K}_{\Gamma_i\Gamma_i}^{(i)} + \mathcal{T}_i) u_{\Gamma_i} - f_{\Gamma_i}^{(i)}$. Then, the first block-row of (2.56) is exactly the i-th block row of (2.53). The second first block-row of (2.56) is true by definition of λ_i. Equation (2.57) is true by definition of the jump matrix $\mathcal{B}_i$ and the restriction matrix $\mathcal{R}_{\Gamma_i}$: $\sum_{i=1}^{N} \mathcal{B}_i \mathcal{R}_{\Gamma_i} = 0$. Equation (2.58) can be obtained from the last block-row of (2.53) by adding $\sum_{i=1}^{N} \lambda_i$ on both sides of the equation. Conversely, for the same reasons, any solution to (2.56 – 2.58) gives a global solution to (2.53) by choosing u_Γ such that $\forall i \; u_{\Gamma_i} = \mathcal{R}_{\Gamma_i} u_\Gamma$: this is possible because an unknown in Γ that appears in several local interfaces Γ_i, Γ_j, ... has the same value in all the vectors u_{Γ_i}, u_{Γ_j}, ... because of Equation (2.57).

Instead of the algebraic graph partition presented above, which needs to be performed *after* the global problem has been discretized, one can partition the meshed domain *before* discretizing the local equations. After the domain Ω has been partitioned as above in N

subdomains Ω_i, one may introduce the discretization $\mathcal{K}_i u_i = f_i$ of the local homogeneous Neumann problem as presented in Section 2.3.2

$$\forall i \in \{1, \dots, N\} \text{ find } \ u_i : \Omega_i \to \mathbb{R} \text{ such that}$$
$$\begin{cases} \mathcal{L}(u_i) = f \text{ in } \Omega_i, \\ \mathcal{B}(u_i) = g \text{ on } \partial\Omega_i \cap \partial\Omega \ , \\ \mathcal{N}_i(u_i) = 0 \text{ on } \Gamma_i. \end{cases}$$

Then, the set of local indices Ω_i can be partitioned into an *interior* $\mathcal{I}_i = \{k \in \Omega_i, \ \forall j \neq i \ k \notin \Omega_j\}$ and a *local interface* $\Gamma_i = \{k \in \Omega_i, \ \exists j \neq i \ k \in \Omega_j\}$. The local matrix and right-hand side can be reordered according to the symmetric permutation $(\mathcal{I}_i \ \Gamma_i)$

$$\mathcal{K}_i = \begin{pmatrix} \mathcal{K}_{\mathcal{I}_i\mathcal{I}_i} & \mathcal{K}_{\mathcal{I}_i\Gamma_i} \\ \mathcal{K}_{\Gamma_i\mathcal{I}_i} & \mathcal{K}^{(i)}_{\Gamma_i\Gamma_i} \end{pmatrix} \qquad \text{and} \qquad f_i = \begin{pmatrix} f_{\mathcal{I}_i} \\ f^{(i)}_{\Gamma_i} \end{pmatrix}. \tag{2.59}$$

These matrices and right-hand sides provide a suitable decomposition as needed in Equation (2.54), and the discretization of the continuous augmented formulation (2.51 – 2.52) is exactly (2.56 – 2.58). This approach has two advantages: first, the discretization process and the matrix assembly are only performed on local problems, independently from one another. As such, they can be run in parallel, and the global matrix is never explicitly assembled. The second benefit is that these local Neumann matrices may inherit some properties from the PDE, that are useful to build a more robust method as in Section 3.

2.6.2 Augmented formulation on the interface

Introducing the *Dirichlet-to-Neumann* operators $\mathcal{S}_i$ and reduced right-hand side rhs_i in each subdomain as in Section 2.4, the continuous augmented formulation can be rewritten as an interface problem

$$\forall i \in \{1, \dots, N\} \text{ find } (u_{\Gamma_i}, \lambda_i) : \Gamma_i \to \mathbb{R} \text{ such that}$$
$$\begin{cases} \mathcal{S}_i(u_{\Gamma_i}) + \mathcal{T}_i(u_{\Gamma_i}) = rhs_i + \lambda_i \text{ on } \Gamma, & (2.60) \\ u_i = u_j = \cdots = u_k \text{ on } \Gamma_{ij\dots k}, & (2.61) \\ \mathcal{T}_i(u_i) + \mathcal{T}_j(u_j) + \cdots + \mathcal{T}_k(u_k) = \lambda_i + \lambda_j + \cdots + \lambda_k \text{ on } \Gamma_{ij\dots k}. & (2.62) \end{cases}$$

Algebraically, introducing the local Schur complement matrices $\mathcal{S}_i$ and reduced right-hand side $\widetilde{f}^{(i)}_{\Gamma_i}$

$$\mathcal{S}_i = \mathcal{K}^{(i)}_{\Gamma_i\Gamma_i} - \mathcal{K}_{\Gamma_i\mathcal{I}_i}\mathcal{K}^{-1}_{\mathcal{I}_i\mathcal{I}_i}\mathcal{K}_{\mathcal{I}_i\Gamma_i} \qquad \text{and} \qquad \widetilde{f}^{(i)}_{\Gamma_i} = f^{(i)}_{\Gamma_i} - \mathcal{K}_{\Gamma_i\mathcal{I}_i}\mathcal{K}^{-1}_{\mathcal{I}_i\mathcal{I}_i} f_{\mathcal{I}_i}, \tag{2.63}$$

the interior variable $u_{\mathcal{I}_i} = \mathcal{K}^{-1}_{\mathcal{I}_i\mathcal{I}_i}(f_{\mathcal{I}_i} - \mathcal{K}_{\mathcal{I}_i\Gamma_i} u_{\Gamma_i})$ can be eliminated in the augmented formulation (2.56 – 2.58) to obtain the augmented formulation on the interface

$$\begin{cases} \forall i \in \{1, \dots, N\} \ [\mathcal{S}_i + \mathcal{T}_i] u_{\Gamma_i} = \widetilde{f}^{(i)}_{\Gamma_i} + \lambda_i & (2.64) \\ \displaystyle\sum_{i=1}^{N} \mathcal{B}_i u_{\Gamma_i} = 0, & (2.65) \\ \displaystyle\sum_{i=1}^{N} \mathcal{R}^T_{\Gamma_i} \mathcal{T}_i u_{\Gamma_i} = \sum_{i=1}^{N} \mathcal{R}^T_{\Gamma_i} \lambda_i. & (2.66) \end{cases}$$

2.6.3 Primal formulations

Eliminating λ_i from an augmented formulation (continuous or algebraic, on Ω or Γ) also eliminates the transmission operators $\mathcal{T}_i$ or $\mathcal{T}_i$ from the equations, yielding the primal formulations

$$\begin{aligned}
&\text{find} \quad u : \Omega \to \mathbb{R} \text{ such that} \\
&\begin{cases}
\mathcal{L}(u) = f \text{ in } \Omega_i, & (2.67)\\
\mathcal{B}(u) = g \text{ on } \partial\Omega_i \cap \partial\Omega \ , & (2.68)\\
\mathcal{N}_i(u) + \mathcal{N}_j(u) + \cdots + \mathcal{N}_k(u) = 0 \text{ on } \Gamma_{ij\ldots k}, & (2.69)
\end{cases}
\end{aligned}$$

and, on the interface,

$$\begin{aligned}
&\text{find} \quad u_\Gamma : \Gamma \to \mathbb{R} \text{ such that} & (2.70)\\
&\mathcal{S}_i(u_\Gamma) + \mathcal{S}_j(u_\Gamma) + \cdots + \mathcal{S}_k(u_\Gamma) = rhs_i + rhs_j + \cdots + rhs_k \text{ on } \Gamma_{ij\ldots k}, & (2.71)
\end{aligned}$$

for the continuous problem, or, algebraically,

$$\sum_{i=1}^{N} \mathcal{R}_{\Omega_i}^T \mathcal{K}_i \mathcal{R}_{\Omega_i} u = \sum_{i=1}^{N} \mathcal{R}_{\Omega_i}^T f_i, \tag{2.72}$$

and

$$\sum_{i=1}^{N} \mathcal{R}_{\Gamma_i}^T \mathcal{S}_i \mathcal{R}_{\Gamma_i} u_\Gamma = \sum_{i=1}^{N} \mathcal{R}_{\Gamma_i}^T \widetilde{f}_{\Gamma_i}^{(i)}. \tag{2.73}$$

An implementation of these two algebraic formulations is proposed in Chapter 4.

2.6.4 Dual augmented formulations

Similarly to the two-subdomain case from Section 2.5.3, if $\sum_{i=1}^{N} \mathcal{T}_i = 0$, Equation (2.62) becomes $\sum_{i=1}^{N} \lambda_i = 0$. On an interface Γ_{ij} between two neighbors, one can search a unique λ_{ij} such that $\lambda_{ij} = \lambda_i = -\lambda_j$. On an interface $\Gamma_{ij\ldots k}$ between Ω_i and two or more neighbors Ω_j, ... and Ω_k, one can introduce the same number of unknowns as neighboring subdomains $\lambda_{ij}, \ldots, \lambda_{ik}$, such that $\lambda_i = \lambda_{ij} + \cdots + \lambda_{ik}$, $\lambda_j = -\lambda_{ij}, \ldots$, and $\lambda_k = -\lambda_{ik}$.

Then, if $\widehat{\mathcal{S}}_i = \mathcal{S}_i + \mathcal{T}_i$ is not singular, one can eliminate $u_{\Gamma_i} = \widehat{\mathcal{S}}_i^{-1}(rhs_i + \lambda_i) = \widehat{\mathcal{S}}_i^{-1}(rhs_i + \lambda_{ij} + \cdots + \lambda_{ik})$, $u_{\Gamma_j} = \widehat{\mathcal{S}}_j^{-1}(rhs_j + \lambda_j) = \widehat{\mathcal{S}}_j^{-1}(rhs_j - \lambda_{ij}), \ldots$, and $u_{\Gamma_k} = \widehat{\mathcal{S}}_k^{-1}(rhs_k - \lambda_{ik})$ and the dual (augmented if $\mathcal{T}_i \neq 0$) formulation is

$$\begin{aligned}
&\text{find} \quad (\lambda_{ij}, \ldots, \lambda_{ik}) : \Gamma_{ij\ldots k} \to \mathbb{R} \text{ such that} \\
&\widehat{\mathcal{S}}_i^{-1}(rhs_i + \lambda_{ij} + \cdots + \lambda_{ik}) = \widehat{\mathcal{S}}_j^{-1}(rhs_j - \lambda_{ij}) = \cdots = \widehat{\mathcal{S}}_k^{-1}(rhs_k - \lambda_{ik}),
\end{aligned}$$

Algebraically, the same process can be applied by replacing $\lambda_i = \mathcal{B}_i^T \lambda$ and eliminating $u_{\Gamma_i} = \widehat{\mathcal{S}}_i^{-1}(\widetilde{f}_{\Gamma_i}^{(i)} + \mathcal{B}_i^T \lambda)$

$$\sum_{i=1}^{N} \mathcal{B}_i \widehat{\mathcal{S}}_i^{-1} \mathcal{B}_i^T \lambda = -\sum_{i=1}^{N} \mathcal{B}_i \widehat{\mathcal{S}}_i^{-1} \widetilde{f}_{\Gamma_i}^{(i)}.$$

The particular case where $\mathcal{T}_i = 0$ and $\widehat{\mathcal{S}}_i = \mathcal{S}_i$ gives the dual (non-augmented) formulation

$$\sum_{i=1}^{N} \mathcal{B}_i \mathcal{S}_i^{-1} \mathcal{B}_i^T \lambda = -\sum_{i=1}^{N} \mathcal{B}_i \mathcal{S}_i^{-1} \widetilde{f}_{\Gamma_i}^{(i)}. \tag{2.74}$$

2.6.5 N-Lagrange formulations

In a subdomain Ω_i with several neighbors Ω_j, ..., Ω_k, the first compatibility condition (2.61) in the continuous augmented system on the interface ensures that $u_i = u_j$ on Γ_{ij}, ... and $u_i = u_k$ on Γ_{ik}. As a result, if all points in Γ_i are also in at least one of Γ_j, ..., Γ_k, one can introduce weight functions $w_{ij}, \ldots, w_{ik} : \Gamma_i \to \mathbb{R}$ such that

$$u_i = w_{ij} u_j + \cdots + w_{ik} u_k \text{ on } \Gamma_i, \quad \text{with} \quad w_{ij} + \cdots + w_{ik} = 1 \tag{2.75}$$

where $u_j, \ldots, u_k$ have been prolonged by 0 on $\Gamma_i \backslash \Gamma_{ij}, \ldots, \Gamma_i \backslash \Gamma_{ik}$, respectively. On a part of Γ_i that is shared with only one other subdomain, for instance Ω_j, $u_i = w_{ij} u_j + \ldots 0 + w_{ik} 0 = w_{ij} u_j$, and $w_{ij} = 1$ since $u_i = u_j$. On a part of Γ_i that is shared by more subdomains, the weights can be chosen arbitrarily such that their sum is 1: on a boundary Γ_{ijk} between Ω_i and two neighbors Ω_j and Ω_k, one could impose for instance $w_{ij} = 0$ and $w_{ik} = 1$, or the reverse, or $w_{ij} = w_{ik} = \frac{1}{2}$. Then, one can eliminate u_i from the second compatibility condition (2.62) using (2.75) and the linearity of $\mathcal{T}_i$

$$\begin{aligned}
& \lambda_i - \mathcal{T}_i(u_i) + \lambda_j - \mathcal{T}_j(u_j) + \cdots + \lambda_k - \mathcal{T}_j(u_k) = 0, \\
\Rightarrow \quad & \lambda_i - \mathcal{T}_i(w_{ij} u_j + \cdots + w_{ik} u_k) + \lambda_j - \mathcal{T}_j(u_j) + \cdots + \lambda_k - \mathcal{T}_j(u_k) = 0, \\
\Rightarrow \quad & \lambda_i + \lambda_j - \mathcal{T}_i(w_{ij} u_j) - \mathcal{T}_j(u_j) + \cdots + \lambda_k - \mathcal{T}_i(w_{ik} u_k) - \mathcal{T}_k(u_k) = 0.
\end{aligned}$$

Similarly, continuing to eliminate $u_j, \ldots, u_k$ gives

$$\begin{aligned}
\lambda_i + \lambda_j - \mathcal{T}_i(w_{ij} \widehat{\mathcal{S}}_j^{-1}(\lambda_j)) - \mathcal{T}_j(\widehat{\mathcal{S}}_j^{-1}(\lambda_j)) + \cdots + & \\
\lambda_k - \mathcal{T}_i(w_{ik} \widehat{\mathcal{S}}_k^{-1}(\lambda_k)) - \mathcal{T}_k(\widehat{\mathcal{S}}_k^{-1}(\lambda_k)) = & \\
\mathcal{T}_i(w_{ij} \widehat{\mathcal{S}}_j^{-1}(rhs_j)) - \mathcal{T}_j(\widehat{\mathcal{S}}_j^{-1}(rhs_j)) + \cdots + & \\
\mathcal{T}_i(w_{ik} \widehat{\mathcal{S}}_k^{-1}(rhs_k)) - \mathcal{T}_k(\widehat{\mathcal{S}}_k^{-1}(rhs_k)). &
\end{aligned}$$

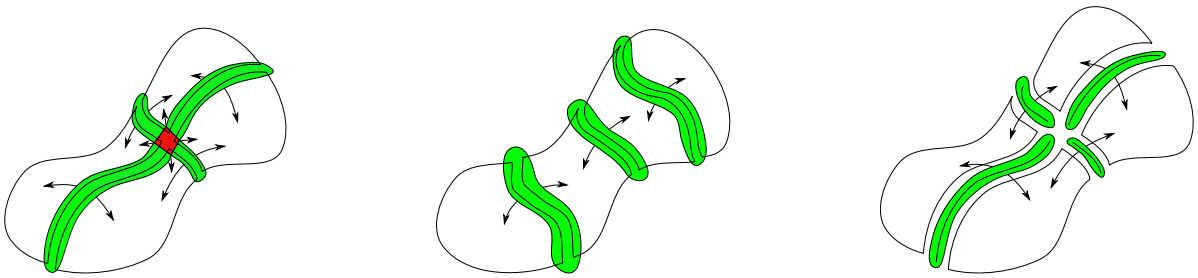
Algebraically, the same operations can be performed: in each subdomain, the interface unknowns are equal to the corresponding interface unknowns in their neighbors' interfaces. It is therefore possible to introduce a set of weight matrices $(W_{ij})_{j \neq i}$ such that $u_{\Gamma_i} = \sum_{j \in \mathcal{N}(i)} W_{ij} \mathcal{R}_{\Gamma_i} \mathcal{R}_{\Gamma_j}^T u_{\Gamma_j}$, where $\mathcal{N}(i) = \{j, \Gamma_{ij} \neq \emptyset\}$ is the set of Ω_i's neighbors. Applying the restriction $\mathcal{R}_{\Gamma_i}$ on the left of Equation (2.66) gives

$$\mathcal{R}_{\Gamma_i} \sum_{j=1}^{N} \mathcal{R}_{\Gamma_j}^T (\lambda_j - \mathcal{T}_j u_{\Gamma_j}) = 0.$$

Since $\mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_i}^T = I$ and $\mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_j}^T = 0$ if $\Gamma_{ij} = \emptyset$, the sum can be restricted to $\mathcal{N}(i)$

$$\begin{aligned}
& \lambda_i - \mathcal{T}_i u_{\Gamma_i} + \sum_{j\in\mathcal{N}(i)} \mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_j}^T\Big(\lambda_j - \mathcal{T}_j u_{\Gamma_j}\Big) = 0, \\
\Rightarrow \quad & \lambda_i - \sum_{j\in\mathcal{N}(i)} \mathcal{T}_i W_{ij}\mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_j}^T u_{\Gamma_j} + \sum_{j\in\mathcal{N}(i)} \mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_j}^T\Big(\lambda_j - \mathcal{T}_j u_{\Gamma_j}\Big) = 0, \\
\Rightarrow \quad & \lambda_i + \sum_{j\in\mathcal{N}(i)} \mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_j}^T\lambda_j - \Big(\mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_j}^T\mathcal{T}_j + \mathcal{T}_i W_{ij}\mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_j}^T\Big)u_{\Gamma_j} = 0, \\
\Rightarrow \quad & \lambda_i + \sum_{j\in\mathcal{N}(i)} \Big[\mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_j}^T - \Big(\mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_j}^T\mathcal{T}_j + \mathcal{T}_i W_{ij}\mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_j}^T\Big)\widehat{\mathcal{S}}_j^{-1}\Big]\lambda_j = \\
& \qquad\qquad \sum_{j\in\mathcal{N}(i)} \Big(\mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_j}^T\mathcal{T}_j + \mathcal{T}_i W_{ij}\mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_j}^T\Big)\widehat{\mathcal{S}}_j^{-1}\widetilde{f}_{\Gamma_j}^{(j)}.
\end{aligned}$$

Writing the same equation for each subdomain, one obtains a N-Lagrange formulation, that can be solved using an iterative method. To perform a matrix vector product, from $\lambda = (\lambda_1^T, \ldots, \lambda_N^T)^T$, a solve $u_i = -\widehat{\mathcal{S}}_i^{-1}\lambda_i = -(\mathcal{S}_i+\mathcal{T}_i)^{-1}\lambda_i$ is performed in each subdomain. Then, each subdomain Ω_i communicates with each neighbor Ω_j the restriction of u_i and $\mu_i = \lambda_i + \mathcal{T}_i u_i$ to the common interface Γ_{ij}: $\mathcal{R}_{\Gamma_j}\mathcal{R}_{\Gamma_i}^T(u_i, \mu_i)$. Then, λ_i' is computed as $\lambda_i' = \lambda_i + \sum_{j\in\mathcal{N}(i)} \mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_j}^T\mu_j + \mathcal{T}_i W_{ij}\mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_j}^T u_j$ and $\lambda' = ({\lambda_1'}^T, \ldots, {\lambda_1'}^T)^T$ is the result of the matrix-vector product. An implementation of this N-Lagrange formulation is proposed in Chapter 4 (Listing 57).



(a) general case (b) 1D decomposition (c) HDG discretization

Figure 2.5: Three decompositions in four subdomains for the same global domain. An interface unknown can be shared by two subdomains (green) or more (red), in which case it is called a *crosspoint*. Crosspoints occur where interfaces cross with one another, as in 2.5a. They can be avoided using a 1D decomposition as in 2.5b or a different discretization method such as the Hybridizable Discontinuous Galerkin (HDG) method (Cockburn et al., 2009) as in 2.5c.

In some cases, each interface unknown is shared by exactly two subdomains, as in figures 2.5b and 2.5c. In that case, $W_{ij} = I$. If furthermore, $\mathcal{T}_i$ is block-diagonal (one block per neighbor), then each subdomain Ω_j may store $\mathcal{T}_i^{(j)} = \mathcal{R}_{\Gamma_j}\mathcal{R}_{\Gamma_i}^T\mathcal{T}_i\mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_j}^T$ for each

neighbor Ω_i. The equation becomes

$$\lambda_i + \sum_{j\in\mathcal{N}(i)} \mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_j}^T(I - (\mathcal{T}_j + \mathcal{T}_i^{(j)})\widehat{\mathcal{S}}_j^{-1})\lambda_j = \sum_{j\in\mathcal{N}(i)} \mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_j}^T(\mathcal{T}_j + \mathcal{T}_i^{(j)})\widehat{\mathcal{S}}_j^{-1}\widetilde{f}_{\Gamma_j}^{(j)}.$$

Each subdomain can directly compute $\mu_i' = \left(I - \left(\mathcal{T}_i + \sum_{j\in\mathcal{N}(i)} \mathcal{T}_j^{(i)}\right)\widehat{\mathcal{S}}_i^{-1}\right)\lambda_i$ and communicate $\mathcal{R}_{\Gamma_j}\mathcal{R}_{\Gamma_i}^T\mu_i'$ to each neighbor Ω_j. The local result of the matrix-vector product can then be computed as $\lambda_i' = \lambda_i + \sum_{j\in\mathcal{N}(i)} \mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_j}^T\mu_j'$.

2.7 Domain decomposition preconditioners

2.7.1 Variations on the Schwarz Alternating method

The Schwarz alternating method (Schwarz, 1870) introduced in Section 1.1.2 was originally introduced for a global domain that is the union of only two overlapping subdomains. Lions (1988, 1989, 1990) studied and extended this method, generalizing it to a N-subdomain decomposition. Used as a preconditioner, this method is known as a *Multiplicative Schwarz* preconditioner. The main drawback of this method is that for each iteration, the solution in the N subdomains need to be performed one after another, sequentially. Lions (1988) proposed a parallel version of the method, where the solution is computed simultaneously in all subdomains, taking the previous iterate as a BC. Algebraically, this is equivalent (Gander, 2008) to the *Restricted Additive Schwarz* (RAS) preconditioner (Cai and Sarkis, 1999). Lions (1990) also generalized the method by replacing the Dirichlet BC by a Robin BC, eliminating the need for an overlap between the subdomains. Algebraically, this is equivalent to the optimized RAS method (St-Cyr et al., 2007), or, with a change of variables, to the N-Lagrange method (Nataf et al., 1995) introduced in Section 2.6.5.

2.7.2 Abstract Schwarz preconditioners

Another method, closely related to the parallel Schwarz method, is the *Additive Schwarz* (AS) method (Matsokin and Nepomnyaschikh, 1985; Dryja and Widlund, 1987) in which the corrections computed in all the subdomains are added in the overlap. This leads to a symmetric AS preconditioner

$$\mathcal{M}_{AS} = \sum_{i=1}^{N} \mathcal{R}_i^T \left(\mathcal{R}_i\mathcal{K}\mathcal{R}_i^T\right)^{-1}\mathcal{R}_i.$$

A thorough historic perspective on all these methods can be found for instance in (Gander, 2008; Gander and Wanner, 2014; Dolean et al., 2015b).

A so-called Neumann-Neumann preconditioner for the primal formulation on the interface (2.73) was introduced by De Roeck and Le Tallec (1991)

$$\mathcal{M}_{NN} = \sum_{i=1}^{N} \mathcal{R}_i^T D_i\mathcal{S}_i^{\dagger} D_i\mathcal{R}_i,$$

where $(D_i)_{i=1}^N$ is a partition of unity such that $\sum_{i=1}^N \mathcal{R}_i^T D_i \mathcal{R}_i = I_n$ and I_n is the $n \times n$ identity matrix. Similarly, a FETI method was introduced by Farhat and Roux (1991), where the preconditioner

$$\mathcal{M}_{FETI} = \sum_{i=1}^{N} \mathcal{B}_i D_i \mathcal{S}_i D_i \mathcal{B}_i^T,$$

is used to solve the dual formulation (2.74). The Neumann-Neumann preconditioner was generalized to a Robin-Robin (RR) preconditioner (Achdou and Nataf, 1997; Achdou et al., 2000).

These methods have been studied under a unified *abstract Schwarz* theory (Toselli and Widlund, 2006): the primal formulations

$$\sum_{i=1}^{N} \mathcal{R}_{\Omega_i}^T \mathcal{K}_i \mathcal{R}_{\Omega_i} u = \sum_{i=1}^{N} \mathcal{R}_{\Omega_i}^T f_i,$$
$$\sum_{i=1}^{N} \mathcal{R}_{\Gamma_i}^T \mathcal{S}_i \mathcal{R}_{\Gamma_i} u_\Gamma = \sum_{i=1}^{N} \mathcal{R}_{\Gamma_i}^T \widetilde{f}_{\Gamma_i}^{(i)},$$

and the dual formulation

$$\sum_{i=1}^{N} \mathcal{B}_i \widehat{\mathcal{S}}_i^{-1} \mathcal{B}_i^T \lambda = -\sum_{i=1}^{N} \mathcal{B}_i \widehat{\mathcal{S}}_i^{\dagger} \widetilde{f}_{\Gamma_i}^{(i)}$$

all have the same form

$$\sum_{i=1}^{N} \mathcal{R}_i^T \mathcal{A}_i \mathcal{R}_i x = b,$$

and the AS, BDD, NN, RR preconditioners above are particular choice of *abstract Schwarz* (aS) preconditioners

$$\mathcal{M}_{aS} = \sum_{i=1}^{N} \mathcal{R}_i^T \widehat{\mathcal{A}}_i^{\dagger} \mathcal{R}_i.$$

Note that the computation of a matrix-vector product or an application of the preconditioner can be performed in parallel using only neighbor-to-neighbor communications: to compute $y = \mathcal{A}x$ (where $\mathcal{A} = \sum_{i=1}^N \mathcal{R}_i^T \mathcal{A}_i \mathcal{R}_i$) or $y = \mathcal{M}_{aS} x$, knowing $x_i = \mathcal{R}_i x$ in each subdomain, one first needs to compute the local operation $y_i' = \mathcal{A}_i x_i$ or $y_i' = \widehat{\mathcal{A}}_i^{\dagger} x_i$. Then, $y_i = \mathcal{R}_i y = \mathcal{R}_i \sum_{j=1}^N \mathcal{R}_j^T y_j'$ can be computed by noticing that $\mathcal{R}_i \mathcal{R}_i^T = I$ and $\mathcal{R}_i \mathcal{R}_j^T = 0$ if $j \notin \mathcal{N}(i) \cup \{i\}$: $y_i = y_i' + \sum_{j \in \mathcal{N}(i)} \mathcal{R}_i \mathcal{R}_j^T y_j'$. Each subdomain only needs to send to its neighbors the restriction of its local component y_i' to their common interface, and sum the contributions it receives from its neighbors. This step is called *assembly*.

2.7.3 Two-level preconditioners

If the local operator $\mathcal{A}_i$ or $\widehat{\mathcal{A}}_i$ is singular, a pseudoinverse may be used in the method. However, the result of $\widehat{\mathcal{A}}_i^\dagger x_i$ is only defined up to an element of $\ker \widehat{\mathcal{A}}_i$. Furthermore, even when $\widehat{\mathcal{A}}_i$ is not singular, the convergence of these methods often deteriorates when the number of subdomains increases, due to a lack of global communication. A solution to these two problems is to use a coarse space V_0 (that contains these kernels) and to use a deflation technique introduced by Nicolaides (1987) to compute a coarse solution $x_0 = V_0(V_0^T\mathcal{A}V_0)^{-1}V_0^T b$. Then, the solution $x^* = \mathcal{A}^{-1}b$ can be computed as $x^* = x_0 + x_\perp$ where $x_\perp = x^* - x_0$. Introducing the coarse projection $\mathcal{P}_0 = V_0(V_0^T\mathcal{A}V_0)^{-1}V_0^T\mathcal{A}$, it holds that $x_0 = \mathcal{P}_0 x^*$ and $x_\perp = (I - \mathcal{P}_0)x^*$. Since $\mathcal{A}\mathcal{P}_0 = \mathcal{P}_0^T\mathcal{A}$ and $\mathcal{P}_0\mathcal{P}_0 = \mathcal{P}_0$, it holds that $\mathcal{A}x_\perp = (I - \mathcal{P}_0^T)\mathcal{A}x^* = (I - \mathcal{P}_0^T)b$. Using the abstract Schwarz preconditioner $\mathcal{M}_{aS}$ to solve this projected, or deflated equation $\mathcal{A}x_\perp = (I - \mathcal{P}_0^T)b$ is equivalent to using the *deflated abstract Schwarz* preconditioner

$$\mathcal{M}_{aS,D} = V_0(V_0^T\mathcal{A}V_0)^{-1}V_0^T + (I - \mathcal{P}_0)\mathcal{M}_{aS}(I - \mathcal{P}_0^T)$$

on the non-deflated equation $\mathcal{A}x = b$. Note that the elimination of interior unknowns presented in Section 2.4 is just a special case of deflation with $V_0 = \mathcal{R}_\mathcal{I}^T$, as proved in (Mansfield, 1990).

Another solution is to add the Coarse Space Correction (CSC) to the aS preconditioner, without the projection, leading to the *(additive) two-level* preconditioner

$$\mathcal{M}_{aS,2} = V_0(V_0^T\mathcal{A}V_0)^{-1}V_0^T + \mathcal{M}_{aS}.$$

This additive two-level variant is in fact only used when $\mathcal{M}_{aS}$ is an AS preconditioner, as explained in Section 3.4, leading to the two-level AS preconditioner $\mathcal{M}_{AS,2}$ introduced in (Dryja and Widlund, 1987).

2.7.4 Choice of a coarse space

The choice of the coarse space is a key element of the construction of a two-level preconditioner. The coarse space introduced by Nicolaides (1987) for a Poisson problem is built by partitioning the unknowns into m disjoint subsets $g_1, \ldots, g_m$, and defining V_0 such that $V_{0ij} = 1$ if $i \in g_j$ and $V_{0ij} = 0$ if $i \notin g_j$. It has been extended by Sarkis (2003) to the case where Ω is a (non-disjoint) union $\Omega = \Omega_1 \cup \cdots \cup \Omega_N$ by using a partition of unity as defined in Section 2.7.2: V_0 is defined as the $(n \times N)$ rectangular matrix whose column j is $\mathcal{R}_i^T diag(D_i)$. The construction of these two coarse spaces for Poisson's equation rely on the fact that the kernel of a local Neumann problem is the subspace of vectors that are constant in the subdomains. Similarly, coarse spaces based on the kernel of local operators have been used in the definition of the balanced NN (Mandel, 1993) and FETI (Farhat and Roux, 1991) methods.

Other coarse spaces have been constructed by separating the interface Γ into so-called faces, edges and crosspoints (or vertices) in (Bramble et al., 1986, 1989; Smith, 1991; Carvalho et al., 2001a).

The coarse spaces based on the kernels of local operators perform very well on homogeneous problems. In the heterogeneous cases, if the discontinuities are not aligned with

the interfaces between subdomains, it is possible to improve the convergence by choosing some solutions of local eigenproblems as in (Galvis and Efendiev, 2010; Nataf et al., 2011; Efendiev et al., 2012; Spillane and Rixen, 2013; Spillane et al., 2014a; Spillane, 2014; Klawonn et al., 2016b,a; Haferssas et al., 2017; Klawonn et al., 2018). A construction and proof of robustness for such a local spectral coarse space for aS preconditioners using algebraic information is proposed in Chapter 3; an implementation of the resulting two-level aS methods is proposed in Chapter 4, and the parallel design of the additive CSC for the AS method applied on the Schur complement system in a HPC hybrid solver is studied in Chapter 5.

Chapter 3

Convergence of abstract Schwarz methods

3.1 Introduction

Several classic domain decomposition methods (DDM) can be expressed as abstract Schwarz (aS) preconditioners (Section 2.7) for the primal formulations (Section 2.6.3) introduced in Chapter 2. The present chapter focuses on the convergence analysis of these aS method. It is illustrated with numerical experiments (Section 3.5). The code designed for that purpose is not presented in the present chapter, as it will instead be the object of Chapter 4 which will focus on the issues related to the design of such a code for achieving high performance in a parallel context while ensuring the required flexibility to cover all the proposed methods. Performance results are eventually provided in Section 3.5.5 to make the present chapter self-contained and illustrate the benefits of a parallel Coarse Space Correction (CSC). Nonetheless, the parallel design of the coarse grid will be tackled in details only in Chapter 5.

The domain decomposition community has developed many efficient and robust methods in the last decades. While some of these solvers fall into the aS framework mentioned above, their robustness have originally been demonstrated on a case-by-case basis, often relying on some properties of the underlying partial differential equations that is being solved. A general convergence theory of aS methods is presented in (Toselli and Widlund, 2006). In this chapter, we propose a bound for the condition number of deflated aS methods for SPD matrices provided that the coarse grid consists of the assembly of local components that contain the kernel of some local operators. We show that classical results from the literature on particular instances of aS methods can be retrieved from this bound. We then show that such a CSC can be explicitly obtained algebraically via generalized eigenproblems, leading to a condition number independent of the number of subdomains when the global SPD matrix is known as a sum of symmetric positive semi-definite (SPSD) matrices. This result can be readily applied to retrieve or improve the bounds previously obtained via generalized eigenproblems in the particular cases of Neumann-Neumann (NN), Additive Schwarz (AS) and optimized Robin but also generalizes them when applied with approximate local solvers. Interestingly, the proposed methodology turns out to be a comparison of the considered particular aS method with

generalized versions of both NN and AS for tackling the lower and upper part of the spectrum, respectively. We furthermore show that the application of the considered coarse grid corrections in an additive fashion is robust in the AS case although it is not robust for aS methods in general. In particular, the proposed framework allows for ensuring the robustness of the AS method applied on the Schur complement (AS/S), either with deflation or additively, and with the freedom of relying on an approximate local Schur complement. Numerical experiments illustrate these statements.

The linear system (Equation (1.5), page 21) to be solved is

$$\mathcal{K}u = f, \tag{3.1}$$

where $\mathcal{K}$ is a $n \times n$ sparse SPD matrix that does not need to be known explicitly. Instead, the parallel application provides $\mathcal{K}$ to the solver as a sum $\mathcal{K} = \sum_{i=1}^{N} \mathcal{K}_i^{(g)}$ of N SPSD matrices $\mathcal{K}_i^{(g)}$. The matrix $\mathcal{K}_i^{(g)}$ is similar to the $\mathcal{K}_i$ matrix introduced in Section 2.6.1 of Chapter 2, but using the global (hence the $^{(g)}$ notation) ordering instead of a local one. Even though $\mathcal{K}_i^{(g)}$ is of size $n \times n$, in practical applications it has only n_i non-zero rows (and columns), meaning that this matrix represents the interaction of only a subset of the unknowns from the global problem. We define the *global domain* $\Omega = \{1, \ldots, n\}$ as the set of row (or column) indices in $\mathcal{K}$, and the *subdomain* $\Omega_i = \{\omega_1^{(i)}, \omega_2^{(i)}, \ldots, \omega_{n_i}^{(i)}\}$ as the set of indices of the non-zero rows and columns in $\mathcal{K}_i^{(g)}$ (Ω_i is the set of vertices in the adjacency graph of $\mathcal{K}_i^{(g)}$). We introduce the $n_i \times n$ canonical restriction matrix $\mathcal{R}_{\Omega_i}$ from Ω to Ω_i, such that for any vector $u = (u_1, \ldots, u_n) \in \mathbb{R}^n$, $\mathcal{R}_{\Omega_i} u$ is the vector $(u_{\omega_1^{(i)}}, \ldots, u_{\omega_{n_i}^{(i)}}) \in \mathbb{R}^{n_i}$. Then, we define the $n_i \times n_i$ SPSD matrix $\mathcal{K}_i = \mathcal{R}_{\Omega_i} \mathcal{K}_i^{(g)} \mathcal{R}_{\Omega_i}^T$, referred to as the *local matrix* of subdomain Ω_i, leading to

$$\mathcal{K} = \sum_{i=1}^{N} \mathcal{R}_{\Omega_i}^T \, \mathcal{K}_i \, \mathcal{R}_{\Omega_i}. \tag{3.2}$$

The unknowns in any subdomain Ω_i can be partitioned into an interior $\mathcal{I}_i = \{\omega \in \Omega_i \quad s.t. \quad \forall j \neq i, \quad \omega \notin \Omega_j\}$ and an interface $\Gamma_i = \{\omega \in \Omega_i \quad s.t. \quad \exists j \neq i \quad \omega \in \Omega_j\} = \Omega_i \setminus \mathcal{I}_i$. If an unknown $\omega \in \Omega_i$ appears in at least one other subdomain, then $\omega \in \Gamma_i$, otherwise $\omega \in \mathcal{I}_i$. This yields a partition of the global domain $\Omega = \{1, \ldots, n\} = \mathcal{I}_1 \cup \cdots \cup \mathcal{I}_N \cup \Gamma$ where $\Gamma = \Gamma_1 \cup \cdots \cup \Gamma_N$ is the global interface.

Then, eliminating in parallel the interior unknowns following for instance (Section 2, Le Tallec and Vidrascu, 1998) the original system (3.1) reduces to a Schur complement problem defined on the interface Γ

$$\mathcal{S} u_\Gamma = \widetilde{f}_\Gamma, \qquad \mathcal{S} = \sum_{i=1}^{N} \mathcal{R}_{\Gamma_i}^T \, \mathcal{S}_i \, \mathcal{R}_{\Gamma_i}, \tag{3.3}$$

where the global Schur complement matrix $\mathcal{S}$ is SPD and the local Schur complement matrices $\mathcal{S}_i$ are SPSD. Using the classical index notation for referring to sub-blocks of matrices and vectors, we have $\mathcal{S} = \mathcal{K}_{\Gamma\Gamma} - \sum_{i=1}^{N} \mathcal{K}_{\Gamma\mathcal{I}_i} \mathcal{K}_{\mathcal{I}_i\mathcal{I}_i}^{-1} \mathcal{K}_{\mathcal{I}_i\Gamma}$, $\widetilde{f}_\Gamma = f_\Gamma - \sum_{i=1}^{N} \mathcal{K}_{\Gamma\mathcal{I}_i} \mathcal{K}_{\mathcal{I}_i\mathcal{I}_i}^{-1} f_{\mathcal{I}_i}$ and $\mathcal{S}_i = \mathcal{K}_{\Gamma_i\Gamma_i} - \mathcal{K}_{\Gamma_i\mathcal{I}_i} \mathcal{K}_{\mathcal{I}_i\mathcal{I}_i}^{-1} \mathcal{K}_{\mathcal{I}_i\Gamma_i}$. From the interface solution u_Γ, the solution in $\mathcal{I}_i$ can be computed as $u_{\mathcal{I}_i} = \mathcal{K}_{\mathcal{I}_i\mathcal{I}_i}^{-1} \left(f_{\mathcal{I}_i} - \mathcal{K}_{\mathcal{I}_i\mathcal{I}_i}^{-1} \mathcal{K}_{\mathcal{I}_i\Gamma} u_\Gamma \right)$.

Algebraically, the problems (3.1) and (3.3) are very similar; their only difference is that even when $\mathcal{K}_i$ is sparse, $\mathcal{S}_i$ is in general a dense matrix (as soon as $\mathcal{K}_i$ is irreducible). Although eliminating the interior unknowns is often associated with specific DDM such as Neumann-Neumann or BDD (De Roeck and Le Tallec, 1991; Mandel, 1993), it is in fact an optional step in the solution of Problem (3.1) and many domain decomposition methods can be applied either directly on $\mathcal{K}$ or, after eliminating the interior unknowns, on $\mathcal{S}$. This elimination step may take time and consume memory, but it allows one to reduce the size and the condition number of the linear system ($\mathcal{S}$) to be solved (Brenner, 1999; Mansfield, 1990), making it a useful optional preprocessing. Since the theory presented in sections 3.2 and 3.3 can be applied to solve either the original problem in (3.1) or the reduced Schur complement problem in (3.3), we write them in a general form as

$$\mathcal{A}x = b, \qquad \mathcal{A} = \sum_{i=1}^{N} \mathcal{R}_i^T \, \mathcal{A}_i \, \mathcal{R}_i, \tag{3.4}$$

where the global SPD matrix $\mathcal{A}$, the local SPSD matrices $\mathcal{A}_i$, and the restriction matrices $\mathcal{R}_i$ can represent $\mathcal{K}$, $\mathcal{K}_i$ and $\mathcal{R}_{\Omega_i}$ or $\mathcal{S}$, $\mathcal{S}_i$ and $\mathcal{R}_{\Gamma_i}$ when solving (3.1) or (3.3), respectively. When needed, a specific method M will be noted $M/\mathcal{K}$ or $M/\mathcal{S}$ to specify on which problem this method is applied. In both cases, $\mathcal{A}$ is SPD and, assuming that the $\mathcal{A}_i$ are assigned to different computing units, Problem (3.4) can be solved in parallel using the preconditioned conjugate gradient method (PCG).

A good preconditioner $\mathcal{M}$ for (3.4) should have the two following properties: (1) $\mathcal{M}$ is SPD and *close* to $\mathcal{A}^{-1}$, in the sense that the condition number $\kappa(\mathcal{M}\mathcal{A})$ should be as small as possible; (2) it is easy to compute $\mathcal{M}u$ for any vector u (at least much easier than $\mathcal{A}^{-1}u$). DDM are often used to build such preconditioners of the form

$$\mathcal{M}_{aS} = \sum_{i=1}^{N} \mathcal{R}_i^T \widehat{\mathcal{A}}_i^{\dagger} \mathcal{R}_i \tag{3.5}$$

where $\widehat{\mathcal{A}}_i$ is a local problem associated with $\mathcal{A}$ on subdomain i, and † represents a pseudo-inverse. These preconditioners have been studied for a long time using the abstract Schwarz (aS) theory (see, e.g., (Dolean et al., 2015b; Toselli and Widlund, 2006) for recent overviews). Two particular cases of these aS preconditioners are the Neumann-Neumann (NN) preconditioner (Mandel, 1993), with $\widehat{\mathcal{A}}_i = D_i^{-1}\mathcal{A}_i D_i^{-1}$, and the Additive Schwarz (AS) preconditioner, with $\widehat{\mathcal{A}}_i = \mathcal{R}_i \mathcal{A} \mathcal{R}_i^T$

$$\mathcal{M}_{NN} = \sum_{i=1}^{N} \mathcal{R}_i^T D_i \mathcal{A}_i^{\dagger} D_i \mathcal{R}_i, \qquad \mathcal{M}_{AS} = \sum_{i=1}^{N} \mathcal{R}_i^T \left(\mathcal{R}_i \mathcal{A} \mathcal{R}_i^T\right)^{-1} \mathcal{R}_i, \tag{3.6}$$

where $(D_i)_{i=1}^N$ is a partition of unity such that $\sum_{i=1}^N \mathcal{R}_i^T D_i \mathcal{R}_i = I_n$ and I_n is the $n \times n$ identity matrix. These two preconditioners are of particular importance, but any other SPSD matrix can be used as the local preconditioner $\widehat{\mathcal{A}}_i$ in (3.5).

Unless $\widehat{\mathcal{A}}_i$ perfectly mimics the global action of $\mathcal{A}$ in subdomain Ω_i, $\kappa(\mathcal{M}_{aS}\mathcal{A})$ may significantly increase with the number N of subdomains, leading to a non scalable numerical method. Furthermore, if $\widehat{\mathcal{A}}_i$ is singular, the pseudo-inverse is only defined up to

an element in its null space $\ker(\widehat{\mathcal{A}}_i)$. To solve these two problems, as discussed in Section 2.7.3, a coarse space V_0 such that $\mathcal{R}_i^T \ker(\widehat{\mathcal{A}}_i) \subset V_0$ can be introduced, leading to the deflated aS preconditioner

$$\mathcal{M}_{aS,D} = V_0(V_0^T\mathcal{A}V_0)^\dagger V_0^T + (I_n - \mathcal{P}_0)\left(\sum_{i=1}^N \mathcal{R}_i^T \widehat{\mathcal{A}}_i^\dagger \mathcal{R}_i\right)(I_n - \mathcal{P}_0)^T \tag{3.7}$$

where $\mathcal{P}_0 = V_0(V_0^T\mathcal{A}V_0)^\dagger V_0^T\mathcal{A}$ is the $\mathcal{A}$-orthogonal projection onto V_0. A simpler additive two-level preconditioner can also be obtained by just adding the coarse component to the one-level preconditioner

$$\mathcal{M}_{aS,2} = V_0(V_0^T\mathcal{A}V_0)^\dagger V_0^T + \sum_{i=1}^N \mathcal{R}_i^T \widehat{\mathcal{A}}_i^\dagger \mathcal{R}_i. \tag{3.8}$$

If V_0 is full rank, the pseudo inverse $(V_0^T\mathcal{A}V_0)^\dagger$ can be replaced by an inverse $(V_0^T\mathcal{A}V_0)^{-1}$.

While previous works had proposed bounds on the condition number $\kappa(\mathcal{M}\mathcal{A})$ on particular numerical cases, often relying on analytical assumptions, Le Tallec and Vidrascu (1998) derived an algebraic bound for a new class of preconditioners, relying on the generalized Rayleigh quotient of two local matrices. These preconditioners are called *generalized NN* in the original article; however, because the generalization consists of handling an approximate matrix, we will instead refer to them as *approximate NN* preconditioners in this thesis. The approximation is not related to the use of inexact solvers to compute the preconditioner, but to the use of an approximation matrix $\widetilde{\mathcal{A}}$ instead of $\mathcal{A}$ in the construction of the preconditioner. The approximate NN preconditioner is in fact an exact algebraic NN preconditioner for $\widetilde{\mathcal{A}}$. Then, this approximate preconditioner is used to accelerate the convergence of PCG applied on the exact matrix $\mathcal{A}$, guaranteeing a convergence towards the actual solution of Equation (3.4).

This class of approximate NN preconditioners generalizes classical NN but does not cover the whole aS class of preconditioners. Note, for instance, that AS cannot be expressed as such an approximate NN preconditioner. The first contribution (Section 3.2) of this chapter is to extend the result from (Le Tallec and Vidrascu, 1998) by using a generic local preconditioner and cover a broader range of aS methods, which we name *approximate deflated aS* methods and consist of all deflated aS methods whose coarse space consists of the assembly of local components that contain the kernel of some local operators (that are formally introduced below, in Definition 1). Interestingly, the bound we exhibit (Theorem 1) highlights the key position of NN and AS among other local preconditioners in the Schwarz framework: they provide two bounds on the spectrum of the preconditioned operator, and the convergence of any aS local preconditioner can be evaluated by comparing it to these two well-known methods.

This bound depends on generalized Rayleigh quotients which are traditionally estimated using functional analysis. Alternatively, we propose to control these Rayleigh quotients algebraically by building the coarse space using eigenvectors of well chosen generalized eigenproblems (Theorem 2). For that, we follow the Generalized Eigenvalue in the Overlap (GenEO) procedure (Spillane et al., 2014a). This second contribution (Section 3.3) results in an explicit procedure for building a robust CSC for any approximate

deflated aS method leading to a bound on the condition number (hence on the number of iterations of PCG) independent of the number of subdomains. This result can be readily applied to retrieve or improve the bounds previously obtained via generalized eigenproblems in the particular cases of AS/$\mathcal{K}$ (Spillane et al., 2014a), NN/$\mathcal{S}$ (Spillane and Rixen, 2013) and optimized Robin (SORAS/$\mathcal{K}$) (Haferssas et al., 2017). It also generalizes these results to the approximate case. The idea of building a coarse space by solving local eigenproblems in each subdomain was introduced in (Galvis and Efendiev, 2010; Nataf et al., 2011); coarse spaces following this general idea for FETI-DP (Farhat et al., 2001) or BDDC (Dohrmann, 2003) were introduced in (Klawonn et al., 2016a, 2018, 2016b).

The third contribution (Section 3.4) in this chapter is that the application of the considered CSC in an additive fashion is robust in the approximate AS case (although it is not robust for aS methods in general). The bound we obtain (Theorem 3) can be applied for retrieving the bound obtained in (Spillane et al., 2014a), when the coarse correction is applied additively to the AS method on the original matrix (AS/$\mathcal{K}$). When working on the Schur complement matrix (AS/$\mathcal{S}$) (Carvalho et al., 2001b), the bound is still valid and leads, as commented in (Galvis and Efendiev, 2010), to a smaller coarse space compared to AS/$\mathcal{K}$.

Numerical experiments illustrate our discussion in Section 3.5. A HPC implementation of the CSC of one particular, robust method (AS/$\mathcal{S}$) has furthermore been implemented in the HPC MaPHyS[1] hybrid (direct/iterative) sparse linear solver (Agullo et al., 2011, 2016b) to eventually assess its performance on a modern parallel computer (Section 3.5.5) and make this scalable method available to the scientific community. The presentation of the `ddmpy` code employed for illustrating the numerical behavior of the proposed methods in sections 3.5.2, 3.5.3 and 3.5.4 is not addressed in the present chapter and will be the object of Chapter 4. Similarly, the presentation of the parallel design of CSC into the MaPHyS fully-featured sparse hybrid solver will be the object of Chapter 5 and Section 3.5.5 only provides a brief preview of it.

The chapter is organized as follows. Section 3.2 introduces a new class of approximate (deflated) aS preconditioners and provides a bound on their condition number, which depends on generalized Rayleigh quotients. Applying the GenEO procedure on two well chosen generalized eigenproblems, Section 3.3 proposes a procedure to explicitly compute the coarse space while bounding these Rayleigh quotients leading to a bound on the condition number (hence on the number of iterations of PCG) independent of the number of subdomains. Section 3.4 shows that a similar result (and procedure) can be obtained when the CSC is additively applied, in the case of approximate AS problems. Numerical experiments illustrate our discussion in Section 3.5 before concluding in Section 3.6.

3.2 Approximate abstract Schwarz preconditioners

In this section, we first define a class of approximate aS preconditioners, which combine a local preconditioner $\widehat{\mathcal{A}}_i$, an approximate matrix $\widetilde{\mathcal{A}}$ and a coarse space V_0 in Section 3.2.1. We then provide a bound on the condition number of this class of methods in Section 3.2.2, whose proof is detailed in Section 3.2.3.

[1] See `https://gitlab.inria.fr/solverstack/maphys/`

3.2.1 Context

Definition 1 (Approximate abstract Schwarz preconditioner $\widetilde{\mathcal{M}}_{aS,D}$)**.** *In order to build such a preconditioner for Problem* (3.4)*, we need the three following ingredients:*

1. *a set of symmetric positive semi-definite (SPSD) local preconditioners* $\widehat{\mathcal{A}}_i$*,*
2. *an approximation* $\widetilde{\mathcal{A}}$ *of* $\mathcal{A}$ *such that*

$$\exists\ (\widetilde{\mathcal{A}}_i)_{i=1}^N, \quad \widetilde{\mathcal{A}} = \sum_{i=1}^N \mathcal{R}_i^T \widetilde{\mathcal{A}}_i \mathcal{R}_i \text{ and } \widetilde{\mathcal{A}}_i \text{ is SPSD}, \tag{3.9}$$

$$\exists\ \omega_-, \omega_+ > 0, \quad \forall v \in V \quad \omega_-\ v^T \mathcal{A} v \leq v^T \widetilde{\mathcal{A}} v \leq \omega_+\ v^T \mathcal{A} v, \tag{3.10}$$

3. *and a coarse space* V_0 *such that*

$$\exists\ (V_0^i)_{i=1}^N, \quad V_0 = \sum_{i=1}^N \mathcal{R}_i^T V_0^i \quad \text{with} \quad \ker(\widehat{\mathcal{A}}_i) + \ker(\widetilde{\mathcal{A}}_i^{(NN)}) \subset V_0^i, \tag{3.11}$$

where $\widetilde{\mathcal{A}}_i^{(NN)} = D_i^{-1} \widetilde{\mathcal{A}}_i D_i^{-1}$.

We can then define a coarse matrix $\widetilde{\mathcal{A}}_0 = V_0^T \widetilde{\mathcal{A}} V_0$*, a coarse projection* $\widetilde{\mathcal{P}}_0 = V_0 \widetilde{\mathcal{A}}_0^\dagger V_0^T \widetilde{\mathcal{A}}$ *and the* approximate aS preconditioner *is then defined as*

$$\widetilde{\mathcal{M}}_{aS,D} = V_0 \widetilde{\mathcal{A}}_0^\dagger V_0^T + (I_n - \widetilde{\mathcal{P}}_0) \left(\sum_{i=1}^N \mathcal{R}_i^T \widehat{\mathcal{A}}_i^\dagger \mathcal{R}_i \right) (I_n - \widetilde{\mathcal{P}}_0)^T. \tag{3.12}$$

Note that the matrix $\widetilde{\mathcal{A}}_i^{(NN)}$ introduced in (3.11) is the local matrix in the approximate NN preconditioner $\widetilde{\mathcal{M}}_{NN,D}$ with the algebraic decomposition from (3.9). The matrices D_i can be any partition of unity as in (3.6). $\widetilde{\mathcal{A}}_i^{(NN)}$ is a scaled version of the local matrix $\widetilde{\mathcal{A}}_i$ in the approximation $\widetilde{\mathcal{A}}$ of $\mathcal{A}$.

When no approximation is used, after a suitable initialization, $\widetilde{\mathcal{M}}_{aS,D}$ can be replaced by $(I_n - \widetilde{\mathcal{P}}_0) \left(\sum_{i=1}^N \mathcal{R}_i^T \widehat{\mathcal{A}}_i^\dagger \mathcal{R}_i \right)$ in the PCG iterations, as noted in (Mandel, 1993).

3.2.2 Convergence result for $\widetilde{\mathcal{M}}_{aS,D}$

In each subdomain, we note $N_i = \#\{j \neq i,\ \mathcal{R}_i \widetilde{\mathcal{A}} \mathcal{R}_j^T \neq 0\}$ the number of neighbors through the connectivity graph of $\widetilde{\mathcal{A}}$. We also define two local subspaces $\widehat{V}_i^\perp$ and $\widetilde{V}_i^\perp$ as the orthogonal spaces of V_0^i for the inner products inferred by $\widehat{\mathcal{A}}_i$ in range$(\widehat{\mathcal{A}}_i)$ and $\widetilde{\mathcal{A}}_i^{(NN)}$ in range$(\widetilde{\mathcal{A}}_i^{(NN)})$ respectively. Then,

$$\text{range}(\mathcal{R}_i) = \widehat{V}_i^\perp \oplus V_0^i = \widetilde{V}_i^\perp \oplus V_0^i, \tag{3.13}$$

$$\forall u \in V_0^i,\ \forall v \in \widehat{V}_i^\perp,\ \forall w \in \widetilde{V}_i^\perp \qquad u^T \widehat{\mathcal{A}}_i v = u^T \widetilde{\mathcal{A}}_i^{(NN)} w = 0. \tag{3.14}$$

Finally, for any SPSD matrix $\mathcal{B}$ and vector u, we note $|u|_{\mathcal{B}} = \sqrt{u^T \mathcal{B} u}$ the $\mathcal{B}$-seminorm of u; if $\mathcal{B}$ is SPD, we note it $||u||_{\mathcal{B}}$.

Theorem 1 (Convergence result for approximate aS)**.** *The condition number of the preconditioned matrix $\widetilde{\mathcal{M}}_{aS,D}\mathcal{A}$ is bounded by*

$$\kappa(\widetilde{\mathcal{M}}_{aS,D}\mathcal{A}) \leq \frac{\omega_+}{\omega_-}\left(1+\max_{1\leq i\leq N}\sup_{v\in\widetilde{V}_i^\perp}\frac{|v|^2_{\widehat{\mathcal{A}}_i}}{|v|^2_{\widetilde{\mathcal{A}}_i^{(NN)}}}\right)\max\left(1,\max_{1\leq i\leq N}(N_i+1)\sup_{v\in\widehat{V}_i^\perp}\frac{|v|^2_{\widetilde{\mathcal{A}}_i^{(AS)}}}{|v|^2_{\widehat{\mathcal{A}}_i}}\right),$$

where $\widetilde{\mathcal{A}}_i^{(NN)} = D_i^{-1}\widetilde{\mathcal{A}}_i D_i^{-1}$ and $\widetilde{\mathcal{A}}_i^{(AS)} = \mathcal{R}_i\widetilde{\mathcal{A}}\mathcal{R}_i^T$.

We see three factors in this bound:

- The first one, with ω_+ and ω_-, controls the quality of the approximation $\widetilde{\mathcal{A}}$. If no approximation is used, then $\widetilde{\mathcal{A}} = \mathcal{A}$ and $\omega_- = \omega_+ = 1$.
- The second one is a generalized Rayleigh quotient between the local preconditioner $\widehat{\mathcal{A}}_i$ and the approximate NN preconditioner $\widetilde{\mathcal{A}}_i^{(NN)} = D_i^{-1}\widetilde{\mathcal{A}}_i D_i^{-1}$ defined in (Le Tallec and Vidrascu, 1998).
- The last one is a generalized Rayleigh quotient between the local preconditioner $\widehat{\mathcal{A}}_i$ and an approximate AS preconditioner $\widetilde{\mathcal{A}}_i^{(AS)} = \mathcal{R}_i\widetilde{\mathcal{A}}\mathcal{R}_i^T$.

As for $\widetilde{\mathcal{A}}_i^{(NN)}$ above with NN, $\widetilde{\mathcal{A}}_i^{(AS)} = \mathcal{R}_i\widetilde{\mathcal{A}}\mathcal{R}_i^T$ is an algebraic generalization of the local matrix in the AS preconditioner in Equation (3.6), built upon the approximation $\widetilde{\mathcal{A}}$ instead of $\mathcal{A}$.

Proof. The proof of Theorem 1 is a direct consequence of lemmas 3 and 5 in Section 3.2.3, using the definition of

$$\kappa(\widetilde{\mathcal{M}}_{aS,D}\mathcal{A}) = \frac{\lambda_{\max}(\widetilde{\mathcal{M}}_{aS,D}\mathcal{A})}{\lambda_{\min}(\widetilde{\mathcal{M}}_{aS,D}\mathcal{A})}.$$ □

Corollary 1 (Convergence results for approximate AS and approximate NN)**.** *We define the approximate AS and NN preconditioners $\widetilde{\mathcal{M}}_{AS,D}$ and $\widetilde{\mathcal{M}}_{NN,D}$ by replacing $\widehat{\mathcal{A}}_i$ with $\widetilde{\mathcal{A}}_i^{(AS)}$ or $\widetilde{\mathcal{A}}_i^{(NN)}$ respectively in Equation* (3.12)*. We also define $N_c = \max_{1\leq i\leq N}(N_i+1)$.*

Then, the condition numbers of $\widetilde{\mathcal{M}}_{NN,D}\mathcal{A}$ and $\widetilde{\mathcal{M}}_{AS,D}\mathcal{A}$ are bounded by

$$\kappa(\widetilde{\mathcal{M}}_{AS,D}\mathcal{A}) \leq \frac{\omega_+}{\omega_-}\left(1+\max_{1\leq i\leq N}\sup_{v\in\widetilde{V}_i^\perp}\frac{|v|^2_{\widetilde{\mathcal{A}}_i^{(AS)}}}{|v|^2_{\widetilde{\mathcal{A}}_i^{(NN)}}}\right)\ N_c,$$

$$\kappa(\widetilde{\mathcal{M}}_{NN,D}\mathcal{A}) \leq \frac{\omega_+}{\omega_-}\max\left(1,\sup_{v\in\widehat{V}_i^\perp}\frac{|v|^2_{\widetilde{\mathcal{A}}_i^{(AS)}}}{|v|^2_{\widetilde{\mathcal{A}}_i^{(NN)}}}\right)\ N_c.$$

Proof. The proof of Corollary 1 is a consequence of lemmas 3 and 4 for AS, and lemmas 2 and 5 for NN. □

Note that the bound for $\widetilde{\mathcal{M}}_{NN,D}$ in Corollary 1 is the same as in (Theorem 1, Le Tallec and Vidrascu, 1998). This bound is tighter than the bound obtained by setting $\widehat{\mathcal{A}}_i = \widetilde{\mathcal{A}}_i^{(NN)}$ in Theorem 1; this comes from the fact that the bound in Lemma 2 is also tighter than its generalization in Lemma 3.

The similarity of the bounds for AS and NN in Corollary 1 shows that the convergence of these two methods are governed by the same quantity $\sup_{v\in\widetilde{V}_i^{\perp}} |v|^2_{\widetilde{\mathcal{A}}_i^{(AS)}}/|v|^2_{\widetilde{\mathcal{A}}_i^{(NN)}}$. As a result, with the same coarse space, we expect the AS/$\mathcal{S}$ method (Carvalho et al., 2001b) to show the same convergence behavior as the BDD method (NN/$\mathcal{S}$) (Mandel, 1993) or its dual counterpart FETI (Farhat and Roux, 1991). Although AS require more communication than NN (each subdomain i has to send the matrix block $\mathcal{R}_j\mathcal{R}_i^T\widetilde{\mathcal{A}}_i\mathcal{R}_i\mathcal{R}_j^T$ to each neighbor j) to setup the preconditioner, one advantage of using AS over NN is that the local preconditioner $\widetilde{\mathcal{A}}_i^{(NN)}$ is often singular in some subdomains while $\widetilde{\mathcal{A}}_i^{(AS)}$ remains SPD, and $\widetilde{\mathcal{A}}_i^{(AS)-1}u_i$ is often easier and faster to compute than $\widetilde{\mathcal{A}}_i^{(NN)\dagger}u_i$.

3.2.3 Proof of Theorem 1

To estimate the condition number of $\widetilde{\mathcal{M}}_{aS,D}\mathcal{A}$, we need to bound the spectrum of this operator from above (Lemma 5) and below (Lemma 3). The lower bound is a consequence of the Stable Decomposition Lemma as stated in (Toselli and Widlund, 2006).

Lemma 1 (Stable decomposition lemma)**.** *If there exists a constant C_0, local matrices $\mathcal{B}_i$ and extension operators $\mathcal{I}_i$, such that* $\ker(\mathcal{B}_i) \subset \ker(\mathcal{I}_i)$ *and every $u \in V$ admits a decomposition*

$$u = \sum_{i=0}^{N} \mathcal{I}_i u_i, \quad \{u_i \in V_i, 0 \le i \le N\} \qquad \textit{that satisfies} \qquad \sum_{i=0}^{N} |u_i|^2_{\mathcal{B}_i} \le C_0^2 \ ||u||^2_{\mathcal{A}}.$$

Then

$$\lambda_{\min}(\mathcal{M}\mathcal{A}) \ge C_0^{-2}, \quad \textit{where} \quad \mathcal{M} = \sum_{i=0}^{N} \mathcal{I}_i \mathcal{B}_i^{\dagger} {\mathcal{I}_i}^T.$$

Proof. see, e.g., Lemma 2.5 in (Toselli and Widlund, 2006). □

Then, although it is not directly used in the proof of Theorem 1, we first expose in Lemma 2 a lower bound for the spectrum of NN ($\widehat{\mathcal{A}}_i = \widetilde{\mathcal{A}}_i^{(NN)}$) as it provides a good insight on the reason behind the Rayleigh quotients in the bound presented in Lemma 3 for the general case.

Lemma 2 (Lower bound for the approximate Neumann-Neumann preconditioner)**.** *Let*

$$\widetilde{\mathcal{M}}_{NN,D} = V_0\widetilde{\mathcal{A}}_0^{\dagger}V_0^T + (I_n - \widetilde{\mathcal{P}}_0)\left(\sum_{i=1}^{N} \mathcal{R}_i^T \widetilde{\mathcal{A}}_i^{(NN)\dagger}\mathcal{R}_i\right)(I_n - \widetilde{\mathcal{P}}_0)^T.$$

Then,

$$\lambda_{\min}(\widetilde{\mathcal{M}}_{NN,D}\mathcal{A}) \ge \frac{1}{\omega_+}.$$

Proof. This is a consequence of Lemma 1 (see Theorem 1 in (Le Tallec and Vidrascu, 1998)). □

If, instead of $\widetilde{\mathcal{A}}_i^{(NN)}$, another local preconditioner $\widehat{\mathcal{A}}_i$ is used, there is no change on the bound if we restrict the operators to the coarse space V_0 since the application of the local preconditioner is preceded and followed by projections $(I_n - \widetilde{\mathcal{P}}_0)$ and $(I_n - \widetilde{\mathcal{P}}_0)^T$. However, in the orthogonal of the coarse space, the bound has to change and reflect the difference between $\widetilde{\mathcal{A}}_i^{(NN)}$ and $\widehat{\mathcal{A}}_i$. As is proved in Lemma 3, the lower bound on the spectrum of $\widetilde{\mathcal{M}}_{aS,D}\mathcal{A}$ can be deduced from the bound for $\widetilde{\mathcal{M}}_{NN,D}\mathcal{A}$ in Lemma 2 by adding a correction related to the generalized Rayleigh quotient between $\widetilde{\mathcal{A}}_i^{(NN)}$ and $\widehat{\mathcal{A}}_i$ in the orthogonal of the coarse space.

Lemma 3 (Lower bound for the approximate abstract Schwarz preconditioner)**.**

$$\lambda_{\min}(\widetilde{\mathcal{M}}_{aS,D}\mathcal{A}) \geq \frac{1}{\omega_+}\left(1 + \max_{1\leq i\leq N}\sup_{v\in\widetilde{V}_i^{\perp}}\frac{|v|^2_{\widehat{\mathcal{A}}_i}}{|v|^2_{\widetilde{\mathcal{A}}_i^{(NN)}}}\right)^{-1}.$$

Proof. We want to split u into a sum of local contributions, while being able to uniformly control the $\widehat{\mathcal{A}}_i$-norm of these contributions u_i with the global $\mathcal{A}$-norm of u to apply Lemma 1. For any u and $i \geq 1$, we decompose $D_i\mathcal{R}_i u = u_i^0 + u_i^{\perp}$ where $u_i^0 \in V_0^i$ and $u_i^{\perp} \in \widetilde{V}_i^{\perp}$. We then define $u_0 = (V_0^T\widetilde{\mathcal{A}}V_0)^{\dagger}V_0^T\mathcal{A}u$ such that $V_0u_0 = \widetilde{\mathcal{P}}_0 u$. We can use the facts that $\sum_{i=1}^N \mathcal{R}_i^T D_i\mathcal{R}_i = I_n$ and $\sum_{i=0}^N \mathcal{R}_i^T u_i^0 \in V_0 \subset \ker(I_n - \widetilde{\mathcal{P}}_0)$ to obtain the decomposition

$$\begin{aligned}
u &= \widetilde{\mathcal{P}}_0 u + (I_n - \widetilde{\mathcal{P}}_0)u = V_0u_0 + (I_n - \widetilde{\mathcal{P}}_0)\sum_{i=1}^N \mathcal{R}_i^T D_i\mathcal{R}_i u \\
&= V_0u_0 + (I_n - \widetilde{\mathcal{P}}_0)\sum_{i=1}^N \mathcal{R}_i^T(u_i^0 + u_i^{\perp}) = V_0u_0 + (I_n - \widetilde{\mathcal{P}}_0)\sum_{i=1}^N \mathcal{R}_i^T u_i^{\perp} \\
&= \sum_{i=0}^N \mathcal{I}_i u_i \quad \text{where} \quad \mathcal{I}_0 = V_0, \quad \mathcal{I}_i = (I_n - \widetilde{\mathcal{P}}_0)\mathcal{R}_i \quad \text{and} \quad u_i = u_i^{\perp}.
\end{aligned}$$

Since $\widetilde{\mathcal{P}}_0$ is a $\widetilde{\mathcal{A}}$-orthogonal projection, it holds that:

$$|u_0|^2_{\widetilde{\mathcal{A}}_0} = |u_0|^2_{V_0^T\widetilde{\mathcal{A}}V_0} = |V_0u_0|^2_{\widetilde{\mathcal{A}}} = |\widetilde{\mathcal{P}}_0 u|^2_{\widetilde{\mathcal{A}}} \leq |u|^2_{\widetilde{\mathcal{A}}} \tag{3.15}$$

Let

$$C = \max_{1\leq i\leq N}\sup_{v\in\widetilde{V}_i^{\perp}}\frac{|v|^2_{\widehat{\mathcal{A}}_i}}{|v|^2_{\widetilde{\mathcal{A}}_i^{(NN)}}} = \max_{1\leq i\leq N}\sup_{v\in\widetilde{V}_i^{\perp}}\frac{|v|^2_{\widehat{\mathcal{A}}_i}}{|v|^2_{D_i^{-1}\widetilde{\mathcal{A}}_i D_i^{-1}}}.$$

We can then use equations (3.14), (3.9) and (3.15):

$$
\begin{aligned}
&|u_i^{\perp}|^2_{\widehat{\mathcal{A}}_i} \leq C|u_i^{\perp}|^2_{D_i^{-1}\widetilde{\mathcal{A}}_i D_i^{-1}} \leq C|u_i^{\perp}+u_i^0|^2_{D_i^{-1}\widetilde{\mathcal{A}}_i D_i^{-1}} = C|\mathcal{R}_i u|^2_{\widetilde{\mathcal{A}}_i},\\
&\sum_{i=1}^{N}|u_i^{\perp}|^2_{\widehat{\mathcal{A}}_i} \leq C\sum_{i=1}^{N}|\mathcal{R}_i u|^2_{\widetilde{\mathcal{A}}_i} = C|u|^2_{\sum_{i=1}^{N}\mathcal{R}_i^T\widetilde{\mathcal{A}}_i\mathcal{R}_i} = C|u|^2_{\widetilde{\mathcal{A}}},\\
&|u_0|^2_{\widetilde{\mathcal{A}}_0}+\sum_{i=1}^{N}|u_i^{\perp}|^2_{\widehat{\mathcal{A}}_i} \leq (1+C)\,|u|^2_{\widetilde{\mathcal{A}}} \leq \omega_+(1+C)|u|^2_{\mathcal{A}},
\end{aligned}
\tag{3.16}
$$

and the local norms are controlled by the global norm. Then, applying Lemma 1, we get

$$
\lambda_{\min}(\widetilde{\mathcal{M}}_{aS,D}\mathcal{A}) \geq \frac{1}{\omega_+}\left(1+\max_{1\leq i\leq N}\sup_{v\in\widetilde{V}_i^{\perp}}\frac{|v|^2_{\widehat{\mathcal{A}}_i}}{|v|^2_{\widetilde{\mathcal{A}}_i^{(NN)}}}\right)^{-1} \square
$$

Now that we have proved a lower bound for the spectrum of $\widetilde{\mathcal{M}}_{aS,D}\mathcal{A}$, we will prove an upper bound in Lemma 5. We first recall a classic upper bound for AS preconditioners in Lemma 4 since it explains the origin of the Rayleigh quotient in the bound for the general case.

Lemma 4 (Upper bound for the approximate Additive Schwarz preconditioner)**.** *Let* $\widetilde{\mathcal{M}}_{AS,D} = V_0\widetilde{\mathcal{A}}_0^{\dagger}V_0^T + (I_n-\widetilde{\mathcal{P}}_0)\left(\sum_{i=1}^{N}\mathcal{R}_i^T\widetilde{\mathcal{A}}_i^{(AS)^{-1}}\mathcal{R}_i\right)(I_n-\widetilde{\mathcal{P}}_0)^T$.

Then,

$$
\lambda_{\max}(\widetilde{\mathcal{M}}_{AS,D}\mathcal{A}) \leq \frac{1}{\omega_-}\max_{1\leq i\leq N}(N_i+1).
$$

Proof. This lemma is a particular case of Lemma 5 which is proven below. □

Lemma 5 (Upper bound for the approximate abstract Schwarz preconditioner)**.**

$$
\lambda_{\max}(\widetilde{\mathcal{M}}_{aS,D}\mathcal{A}) \leq \frac{1}{\omega_-}\max\left(1,\max_{1\leq i\leq N}(N_i+1)\sup_{v\in\widehat{V}_i^{\perp}}\frac{|v|^2_{\widetilde{\mathcal{A}}_i^{(AS)}}}{|v|^2_{\widehat{\mathcal{A}}_i}}\right).
$$

Proof. First, let us remark that

$$
\begin{aligned}
\widetilde{\mathcal{M}}_{aS,D}\widetilde{\mathcal{A}}u &= V_0\widetilde{\mathcal{A}}_0^{\dagger}V_0^T\widetilde{\mathcal{A}}u + (I_n-\widetilde{\mathcal{P}}_0)\sum_{i=1}^{N}\mathcal{R}_i^T\widehat{\mathcal{A}}_i^{\dagger}\mathcal{R}_i(I_n-\widetilde{\mathcal{P}}_0)^T\widetilde{\mathcal{A}}u\\
&= u_0 + (I_n-\widetilde{\mathcal{P}}_0)\sum_{i=1}^{N}\mathcal{R}_i^T u_i
\end{aligned}
$$

where $u_0 = \widetilde{\mathcal{P}}_0 u$ and u_i is the orthogonal projection of $\widehat{\mathcal{A}}_i^{\dagger}\mathcal{R}_i(I_n-\widetilde{\mathcal{P}}_0)^T\widetilde{\mathcal{A}}u$ onto range$(\widehat{\mathcal{A}}_i)$ along $\ker(\widehat{\mathcal{A}}_i) \subset V_0^i \subset \ker\left[(I_n-\widetilde{\mathcal{P}}_0)\mathcal{R}_i^T\right]$.

As a consequence, $u_i \in \widehat{V}_i^{\perp}$:

$$u_i^T \widehat{\mathcal{A}}_i V_0^i = u^T \widetilde{\mathcal{A}}(I_n - \widetilde{\mathcal{P}}_0)\mathcal{R}_i^T \widehat{\mathcal{A}}_i^{\dagger} \widehat{\mathcal{A}}_i V_0^i = u^T \widetilde{\mathcal{A}}(I_n - \widetilde{\mathcal{P}}_0)\mathcal{R}_i^T V_0^i = 0.$$

Then,

$$\begin{aligned}|\widetilde{\mathcal{M}}_{aS,D}\widetilde{\mathcal{A}}u|_{\widetilde{\mathcal{A}}}^2 &= |u_0|_{\widetilde{\mathcal{A}}}^2 + |(I_n - \widetilde{\mathcal{P}}_0)\sum_{i=1}^N \mathcal{R}_i^T u_i|_{\widetilde{\mathcal{A}}}^2 \leq |u_0|_{\widetilde{\mathcal{A}}}^2 + |\sum_{i=1}^N \mathcal{R}_i^T u_i|_{\widetilde{\mathcal{A}}}^2 \\ &\leq |u_0|_{\widetilde{\mathcal{A}}}^2 + \sum_{i=1}^N (N_i+1)|\mathcal{R}_i^T u_i|_{\widetilde{\mathcal{A}}}^2 = |u_0|_{\widetilde{\mathcal{A}}}^2 + \sum_{i=1}^N (N_i+1)|u_i|_{\mathcal{R}_i\widetilde{\mathcal{A}}\mathcal{R}_i^T}^2\end{aligned}$$

where we used the fact that

$$\begin{aligned}0 \leq \sum_{\substack{1\leq i,j\leq N \\ \mathcal{R}_i^T\widetilde{\mathcal{A}}\mathcal{R}_j\neq 0}} |\mathcal{R}_i^T u_i - \mathcal{R}_j^T u_j|_{\widetilde{\mathcal{A}}}^2 &= 2\left(\sum_{\substack{1\leq i,j\leq N \\ \mathcal{R}_i^T\widetilde{\mathcal{A}}\mathcal{R}_j\neq 0}} |\mathcal{R}_i^T u_i|_{\widetilde{\mathcal{A}}}^2 - \sum_{\substack{1\leq i,j\leq N \\ \mathcal{R}_i^T\widetilde{\mathcal{A}}\mathcal{R}_j\neq 0}} u_i^T \mathcal{R}_i \widetilde{\mathcal{A}} \mathcal{R}_j^T u_j\right) \\ &\leq 2\left(\sum_{i=1}^N (N_i+1)|\mathcal{R}_i^T u_i|_{\widetilde{\mathcal{A}}}^2 - |\sum_{i=1}^N \mathcal{R}_i^T u_i|_{\widetilde{\mathcal{A}}}^2\right). \end{aligned} \tag{3.17}$$

Let us define

$$C = \max\left(1, \max_{1\leq i\leq N}(N_i+1)\sup_{v\in\widehat{V}_i^{\perp}} \frac{|v|_{\widetilde{\mathcal{A}}_i^{(AS)}}^2}{|v|_{\widehat{\mathcal{A}}_i}^2}\right) = \max\left(1, \max_{1\leq i\leq N}(N_i+1)\sup_{v\in\widehat{V}_i^{\perp}} \frac{|v|_{\mathcal{R}_i\widetilde{\mathcal{A}}\mathcal{R}_i^T}^2}{|v|_{\widehat{\mathcal{A}}_i}^2}\right).$$

We can now write

$$\begin{aligned}|\widetilde{\mathcal{M}}_{aS,D}\widetilde{\mathcal{A}}u|_{\widetilde{\mathcal{A}}}^2 &\leq C|u_0|_{\widetilde{\mathcal{A}}}^2 + C\sum_{i=1}^N |u_i|_{\widehat{\mathcal{A}}_i}^2 = Cu^T\widetilde{\mathcal{P}}_0^T\widetilde{\mathcal{A}}u_0 + C\sum_{i=1}^N u^T\widetilde{\mathcal{A}}(I_n-\widetilde{\mathcal{P}}_0)\mathcal{R}_i^T\widehat{\mathcal{A}}_i^{\dagger}\widehat{\mathcal{A}}_i u_i \\ &= Cu^T\widetilde{\mathcal{A}}\widetilde{\mathcal{M}}_{aS,D}\widetilde{\mathcal{A}}u \leq C|u|_{\widetilde{\mathcal{A}}}|\widetilde{\mathcal{M}}_{aS,D}\widetilde{\mathcal{A}}u|_{\widetilde{\mathcal{A}}} \\ |\widetilde{\mathcal{M}}_{aS,D}\widetilde{\mathcal{A}}u|_{\widetilde{\mathcal{A}}} &\leq C|u|_{\widetilde{\mathcal{A}}},\end{aligned}$$

and use the same strategy as in (Le Tallec and Vidrascu, 1998) to obtain our result:

$$\begin{aligned}\lambda_{\max}(\widetilde{\mathcal{M}}_{aS,D}\mathcal{A}) &= \max_{v\in V}\frac{|v|_{\mathcal{A}}^2}{|v|_{\widetilde{\mathcal{M}}_{aS,D}^{-1}}^2} \leq \max_{v\in V}\frac{1}{\omega_-}\frac{|v|_{\widetilde{\mathcal{A}}}^2}{|v|_{\widetilde{\mathcal{M}}_{aS,D}^{-1}}^2} \leq \max_{v\in V}\frac{1}{\omega_-}\frac{|\widetilde{\mathcal{M}}_{aS,D}\widetilde{\mathcal{A}}v|_{\widetilde{\mathcal{A}}}}{|v|_{\widetilde{\mathcal{A}}}} \leq \frac{C}{\omega_-}, \\ \lambda_{\max}(\widetilde{\mathcal{M}}_{aS,D}\mathcal{A}) &\leq \frac{1}{\omega_-}\max\left(1, \max_{1\leq i\leq N}(N_i+1)\sup_{v\in\widehat{V}_i^{\perp}}\frac{|v|_{\widetilde{\mathcal{A}}_i^{(AS)}}^2}{|v|_{\widehat{\mathcal{A}}_i}^2}\right).\end{aligned}$$

□

3.3 Building the coarse space via generalized eigenproblems

The bound in Theorem 1 has originally been estimated through functional analysis after a coarse space has been chosen (Le Tallec and Vidrascu, 1998). A more algebraic approach

is to build the coarse space V_0 by solving a generalized eigenproblem in each subdomain in order to control the Rayleigh quotient as proposed by (Spillane et al., 2014a; Spillane and Rixen, 2013) for AS/$\mathcal{K}$ and NN/$\mathcal{S}$, respectively. This approach has also been successfully applied to other aS variants such as the SORAS method (Haferssas et al., 2017), in which case two eigenproblems are needed. The case where the correction is applied additively as in (Efendiev et al., 2012; Galvis and Efendiev, 2010; Spillane et al., 2014a) for AS is addressed in Section 3.4.

The connection between the GenEO method and Theorem 1 comes from the following lemma:

Lemma 6 (Bound on the Rayleigh quotient)**.** *Let $\mathcal{B}$ be a SPSD matrix, $\mathcal{C}$ a SPD matrix and $\eta > 0$ be a parameter.*

If $V_\eta = span\big(\{p, \quad \mathcal{B}p = \lambda \mathcal{C}p, \quad \lambda \leq \eta\}\big)$ and $V_\eta^{\perp_\mathcal{B}} = \{u \in range(\mathcal{B}), \forall v \in V_\eta, u^T\mathcal{B}v = 0\}$,

$$then \quad \sup_{u \in V_\eta^{\perp_\mathcal{B}}} \frac{|u|_\mathcal{C}^2}{|u|_\mathcal{B}^2} \leq \frac{1}{\eta}.$$

Proof. Since $\mathcal{C}$ is SPD, the generalized eigenproblem $\mathcal{B}p = \lambda\mathcal{C}p$ has solutions (λ_k, p_k) with $p_k^T\mathcal{C}p_l = \delta_{kl}$ and $p_k^T\mathcal{B}p_l = \lambda_k\delta_{kl}$.

Now, let $u \in V_\eta^{\perp_\mathcal{B}}$. We can project u on the basis $(p_k)_k$: $u = \sum_k \alpha_k p_k$.

If k is such that $\lambda_k \leq \eta$, then $p_k \in V_\eta$ and $0 = u^T\mathcal{B}p_k = \lambda_k\alpha_k$. As a consequence, $\alpha_k = 0$ because if $\lambda_k = 0$, $p_k \in \ker(\mathcal{B}) = \big(\text{range}(\mathcal{B})\big)^\perp \perp u$ and $\alpha_k = u^T p_k = 0$. This leads to

$$\frac{|u|_\mathcal{C}^2}{|u|_\mathcal{B}^2} = \frac{\sum_{\lambda_k > \eta} \alpha_k^2}{\sum_{\lambda_k > \eta} \lambda_k \alpha_k^2} \leq \frac{1}{\eta}. \qquad \square$$

Following the GenEO methodology, we propose to build the coarse space V_0 by solving two generalized eigenproblems to control the condition number of approximate aS preconditioners through two parameters $\alpha > 0$ and $\beta \geq 1$.

Theorem 2 (Condition number of aS preconditioners)**.** *If $\widehat{\mathcal{A}}_i$ is SPD and the coarse space is defined as $V_0 = \sum_{i=1}^{N} \mathcal{R}_i^T V_0^i$ with*

$$\begin{aligned} V_0^i = span\big(\{p_k^i, \quad \widetilde{\mathcal{A}}_i^{(NN)} p_k^i &= \lambda_k^i \widehat{\mathcal{A}}_i p_k^i, \quad \lambda_k^i \leq \alpha^{-1}\} \\ \cup \{p_k^i, \quad \widehat{\mathcal{A}}_i p_k^i &= \lambda_k^i \widetilde{\mathcal{A}}_i^{(AS)} p_k^i, \quad \lambda_k^i \leq (N_i+1)\beta^{-1}\}\big), \end{aligned}$$

then, we can bound the condition number

$$\kappa(\widetilde{\mathcal{M}}_{aS,D}\mathcal{A}) \leq \frac{\omega_+}{\omega_-}(1+\alpha)\,\beta.$$

Proof. Using Lemma 6 and the definition of $\widetilde{V}_i^\perp$ and $\widehat{V}_i^\perp$ in 3.2.2, we can bound the Rayleigh quotients

$$\sup_{v \in \widetilde{V}_i^\perp} \frac{|v|_{\widehat{\mathcal{A}}_i}^2}{|v|_{\widetilde{\mathcal{A}}_i^{(NN)}}^2} \leq \alpha, \qquad \sup_{v \in \widehat{V}_i^\perp} \frac{|v|_{\widetilde{\mathcal{A}}_i^{(AS)}}^2}{|v|_{\widehat{\mathcal{A}}_i}^2} \leq \frac{\beta}{N_i+1}.$$

Replacing these bounds in Theorem 1 gives the result. $\square$

Corollary 2. *In the NN or AS cases, for any $\alpha \geq 1$, we can define*

$$V_0^i = span\big(\{p_k^i, \quad \widetilde{\mathcal{A}}_i^{(NN)} p_k^i \;=\; \lambda_k^i \widetilde{\mathcal{A}}_i^{(AS)} p_k^i, \quad \lambda_k^i \leq \alpha^{-1}\}\big).$$

Then, Corollary 1 and Lemma 6 give

$$\kappa(\widetilde{\mathcal{M}}_{AS,D}\mathcal{A}) \leq \frac{\omega_+}{\omega_-}(1+\alpha)\;\; N_c, \qquad\qquad \kappa(\widetilde{\mathcal{M}}_{NN,D}\mathcal{A}) \leq \frac{\omega_+}{\omega_-}\;\alpha\;N_c.$$

If $\alpha^{-1} = \min_{\lambda_k^i \neq 0}(\lambda_k^i)$, then $V_0^i = \ker(\widetilde{\mathcal{A}}_i^{(NN)}) = D_i \ker(\widetilde{\mathcal{A}}_i)$ and the resulting coarse space for NN is exactly the same as in the BDD algorithm.

With small variations in the generalized eigenproblems considered, Theorem 2 and Corollary 2 retrieve or improve previous GenEO results and generalize them to the approximate case: AS/$\mathcal{K}$ (Spillane, 2014; Spillane et al., 2014a), NN/$\mathcal{S}$ (Spillane and Rixen, 2013) and SORAS (Haferssas et al., 2017).

3.4 Additive coarse correction

3.4.1 Context

The preconditioner $\widetilde{\mathcal{M}}_{aS,D}$ separates the part of the solution that is in V_0 (on which a direct coarse solve is performed through $\widetilde{\mathcal{A}}_0^{\dagger}$), from its $\widetilde{\mathcal{A}}$-orthogonal part (on which the local preconditioner $\mathcal{M}_{aS} = \sum_{i=1}^N \mathcal{R}_i^T \widehat{\mathcal{A}}_i^{\dagger} \mathcal{R}_i$ is used to accelerate convergence). Eigenvalues or Rayleigh quotients λ corresponding to vectors in the coarse space V_0 are shifted to 1 by the coarse solve, and to 0 by the projection steps $(I_n - \widetilde{\mathcal{P}}_0)$ and $(I_n - \widetilde{\mathcal{P}}_0)^T$, so the overall effect of the deflated preconditioner is to shift them to 1 exactly. If we skip these projection steps, we obtain an approximate additive two-level preconditioner $\widetilde{\mathcal{M}}_{aS,2}$ similar to $\mathcal{M}_{aS,2}$ presented in Equation (3.8). In this case, without the projection steps eigenvalues are shifted to $1+\lambda$. As a result, this coarse correction applied on big eigenvalues only makes them bigger, thus hampering convergence. This additive coarse correction can only be effective to tackle the lower part of the spectrum since small eigenvalues $\lambda \ll 1$ are shifted to $1 + \lambda \approx 1$.

The one-level AS method has already an upper bound on the spectrum (see Lemma 4), and only the lower bound needs to be recovered, making it an ideal candidate for an additive coarse correction. In this section, we show that in the approximate AS case, when $\widehat{\mathcal{A}}_i = \widetilde{\mathcal{A}}_i^{(AS)} = \mathcal{R}_i \widetilde{\mathcal{A}} \mathcal{R}_i^T$, the projection steps can be removed without losing robustness. Namely, we still have a bound for the condition number of the additive two-level AS method independent of the number of subdomains.

3.4.2 Convergence result for $\widetilde{\mathcal{M}}_{AS,2}$

Theorem 3 (Condition number of the two-level approximate AS preconditioner)**.** *Let* $\mathcal{M}_{AS,2} = V_0 \widetilde{\mathcal{A}}_0^{\dagger} V_0^T + \sum_{i=1}^N \mathcal{R}_i^T \widetilde{\mathcal{A}}_i^{(AS)-1} \mathcal{R}_i$ *and* $N_c = \max_{1 \leq i \leq N}\,(N_i + 1)$.

Then, we can bound the condition number

$$\kappa(\mathcal{M}_{AS,2}\mathcal{A}) \leq \frac{\omega_+}{\omega_-}\left[N_c + 1 + (N_c+2)\max_{1\leq i\leq N}\sup_{v\in\widetilde{V}_i^\perp}\frac{|v|^2_{\widetilde{\mathcal{A}}_i^{(AS)}}}{|v|^2_{\widetilde{\mathcal{A}}_i^{(NN)}}}\right](N_c+1).$$

For any $\alpha > 0$, if we choose

$$V_0^i = span\big(\{p_k^i, \quad \widetilde{\mathcal{A}}_i^{(NN)}p_k^i = \lambda_k^i\widetilde{\mathcal{A}}_i^{(AS)}p_k^i, \quad \lambda_k^i \leq \alpha^{-1}\}\big),$$

it holds that

$$\kappa(\mathcal{M}_{AS,2}\mathcal{A}) \leq \frac{\omega_+}{\omega_-}\left[N_c + 1 + \alpha(N_c+2)\right](N_c+1).$$

Theorem 3 generalizes (Theorem 4.40, Spillane et al., 2014a) to the approximate case, while improving the bound.

A spectral coarse space composed of eigenvectors of a generalized eigenproblem was earlier proposed in (Efendiev et al., 2012; Galvis and Efendiev, 2010). In those studies, the authors also discuss the analytical and numerical interest of using AS,2/$\mathcal{S}$ instead of the more traditional AS,2/$\mathcal{K}$ to reduce the size of the coarse space. In comparison, our method is more algebraic in the sense that it does not need a stable interpolation operator, nor the mass matrix.

3.4.3 Proof of Theorem 3

Proof. If we apply Lemma 4 without a coarse space and consider V_0 as another subdomain in the decomposition, we get

$$\lambda_{\max}(\mathcal{M}_{AS,2}\mathcal{A}) \leq \frac{1}{\omega_-}(N_c+1).$$

The lower bound is a consequence of Lemma 1. We define $u_i^0 \in V_0^i$ and $u_i^\perp \in \widetilde{V}_i^\perp$ such that $D_i\mathcal{R}_i u = u_i^0 + u_i^\perp$ as in the proof of Lemma 3. We now introduce u_0 such that $V_0u_0 = \sum_{i=1}^N \mathcal{R}_i^T u_i^0$, and $u = V_0u_0 + \sum_{i=1}^N \mathcal{R}_i^T u_i^\perp$.

We get from Equation (3.16) that

$$\sum_{i=1}^N |u_i^\perp|^2_{\widetilde{\mathcal{A}}_i^{(AS)}} = \sum_{i=1}^N |u_i^\perp|^2_{\widehat{\mathcal{A}}_i} \leq C|u|^2_{\widetilde{\mathcal{A}}} \qquad \text{with} \qquad C = \max_{1\leq i\leq N}\sup_{v\in\widetilde{V}_i^\perp}\frac{|v|^2_{\widetilde{\mathcal{A}}_i^{(AS)}}}{|v|^2_{\widetilde{\mathcal{A}}_i^{(NN)}}} = \frac{|v|^2_{\mathcal{R}_i\widetilde{\mathcal{A}}\mathcal{R}_i^T}}{|v|^2_{D_i^{-1}\widetilde{\mathcal{A}}_iD_i^{-1}}}.$$

Then, we can use the same method as in Equation (3.17):

$$\begin{aligned}
|u_0|^2_{\widetilde{\mathcal{A}}} &= |u - \sum_{i=1}^N \mathcal{R}_i^T u_i^\perp|^2_{\widetilde{\mathcal{A}}} \leq (N_c+1)\left(|u|^2_{\widetilde{\mathcal{A}}} + \sum_{i=1}^N |\mathcal{R}_i^T u_i^\perp|^2_{\widetilde{\mathcal{A}}}\right) \\
&= (N_c+1)\left(|u|^2_{\widetilde{\mathcal{A}}} + \sum_{i=1}^N |u_i^\perp|^2_{\mathcal{R}_i\widetilde{\mathcal{A}}\mathcal{R}_i^T}\right) \leq (N_c+1)(1+C)|u|^2_{\widetilde{\mathcal{A}}} \\
|u_0|^2_{\widetilde{\mathcal{A}}} + \sum_{i=1}^N |u_i^\perp|^2_{\mathcal{R}_i\widetilde{\mathcal{A}}\mathcal{R}_i^T} &\leq \left[N_c + 1 + (N_c+2)C\right]|u|^2_{\widetilde{\mathcal{A}}} \leq \omega_+\left[N_c + 1 + (N_c+2)C\right]|u|^2_{\mathcal{A}}.
\end{aligned}$$

We then use Lemma 1 with $\mathcal{I}_0 = V_0$, $\mathcal{I}_i = \mathcal{R}_i^T$ and $\mathcal{B}_i = {\mathcal{I}_i}^T \widetilde{\mathcal{A}} \mathcal{I}_i$ to get the bound

$$\lambda_{\min}(\mathcal{M}_{AS,2}\mathcal{A}) \geq \frac{1}{\omega_+} \left[N_c + 1 + (N_c+2) \max_{1\leq i\leq N} \sup_{v \in \widetilde{V}_i^{\perp}} \frac{|v|^2_{\mathcal{R}_i \widetilde{\mathcal{A}} \mathcal{R}_i^T}}{|v|^2_{\widetilde{\mathcal{A}}_i^{(NN)}}} \right]^{-1}.$$

We can then conclude with Lemma 6. □

3.5 Numerical experiments

3.5.1 Experimental setup

The methods introduced in sections 3.2, 3.3 and 3.4 are tested on a Darcy *bâton* problem similar to what is presented in (Spillane et al., 2014a). We use the Finite Element Method (FEM) with Q1 elements to solve a heterogeneous diffusion equation $\nabla \cdot (k \nabla u) = 1$ in a 3D stratified medium. The domain $[0, N] \times [0, 6] \times [0, 1]$ is discretized on a regular mesh of $(5N+1) \times 31 \times 6$ nodes. The domain is divided into N identical subdomains along the first axis. Along the second axis, it is divided into 10 layers (of $5N \times 3 \times 5$ elements each) of alternating conductivity $k = 1$ and $k = K$ (K is a heterogeneity parameter). A Dirichlet boundary condition is applied on the left of the domain ($x = 0$), a Neumann condition on every other boundary. These test cases are generated using the `genfem` python module[2] developed for the purpose of this thesis. Using a FEM discretization on each subdomain gives rise naturally to a set of local SPSD matrices and a global matrix that is SPD. The geometry and 1D partitioning of this test case are chosen so as to emphasize the effects of using a CSC: indeed, without a coarse correction, the number of iterations grows as $O(N^{1/d})$ where d is the dimension of the partitioning. Using a 3D partitioning of the global domain, one would need more than 7M subdomains (192^3) to illustrate the same effect as in the experiments presented here with a 1D partitioning and 192 subdomains. The layered structure of the domain is introduced to deteriorate the condition number of the local subproblems. Since all subdomains (except the first and last ones) are identical, the bound on the condition number of the method in Theorem 1 is independent of N if at least the kernels of $\widetilde{\mathcal{A}}_i^{(NN)}$ and $\widehat{\mathcal{A}}_i$ are included in V_0^i; a coarse space that only includes these kernels (as in BDD for instance) thus yields a method that can be considered as robust in this regard, while being considerably simpler to compute than the coarse space proposed in this chapter. However, the condition number still depends on the inverse of the smallest eigenvalues not included in the coarse space, which can be quite close to 0 if the local problems are ill-conditioned (*i.e.*, if K is big). As a result, the condition number, although independent of N, can still be too large for the iterative solver to converge in a reasonable number of iterations. Building the coarse space by solving the generalized eigenproblems as proposed in Section 3.3 yields a more robust method in the sense that the condition number of the method can be controlled independently of both N, K, and the particular choice of a local preconditioner. We consider three aS methods: the AS and NN preconditioners introduced in Equation (3.6) and a Shifted (Sh) preconditioner

[2] `https://gitlab.inria.fr/solverstack/genfem`

whose local matrix is obtained by shifting the diagonal of $\widetilde{\mathcal{A}}_i$ by 1 to remove its potential singularity: $\widetilde{\mathcal{M}}_{Sh} = \sum_{i=1}^{N} \mathcal{R}_i^T (\widetilde{\mathcal{A}}_i + I_{n_i})^{\dagger} \mathcal{R}_i$ where I_{n_i} is the identity matrix of same size as $\mathcal{A}_i$. If built on the Schur complement matrix, $\widetilde{\mathcal{M}}_{Sh}$ is a (non-optimized) Robin preconditioner. The optimization of the Robin condition as proposed in (Gander, 2006) is not considered here as it is out of the scope of this study. This Sh preconditioner is introduced as an example of a more generic aS preconditioner than AS and NN; as such, two generalized eigenproblems need to be solved to compute the coarse space for Sh as opposed to only one for AS and NN. Each of these method is assessed with $\mathcal{A} = \mathcal{K}$ or $\mathcal{A} = \mathcal{S}$. Equation (3.4) can therefore either result from:

- the FEM discretization (3.1) of the global problem, in which case the preconditioner is said to be applied on the original matrix $\mathcal{K}$ and the abstract Schwarz method is noted aS/$\mathcal{K}$;
- or the substructuring system (3.3) obtained by eliminating the interior variables from Equation (3.1), in which case the preconditioner is said to be applied on the Schur complement matrix $\mathcal{S}$ and the method is noted aS/$\mathcal{S}$.

The partition of unity D_i is computed using the diagonal values of $\mathcal{A}_i$. The condition numbers of the preconditioned matrices are estimated using the eigenvalues of the tridiagonal Lanczos matrix computed during the PCG iterations (see, e.g., (Frayssé and Giraud, 2000)). The stopping criterion is based on the normwise backward error $||b - \mathcal{A}x_k||/||b|| \leq 10^{-6}$.

Using the `ddmpy` toolbox presented later in Chapter 4, we study the numerical behavior of these methods under the constraint of a bounded condition number or an imposed coarse space size in sections 3.5.2 and 3.5.3, respectively. We then study the approximate case with an empirical approach in Section 3.5.4, using a so-called *sparsification* technique. Our numerical results overall confirm (Efendiev et al., 2012; Galvis and Efendiev, 2010) regarding the numerical interest of using AS,2/$\mathcal{S}$ instead of the more traditional AS,2/$\mathcal{K}$ method to reduce the size of the coarse space. Section 3.5.5 eventually illustrates the parallel behavior of that promising variant implemented as an extension to the HPC MaPHyS solver[3]. The parallel design of this extension is presented in more details in Section 5.

3.5.2 Imposing an *a priori* bound on the condition number

We proved in Section 3.3 that it is possible to control the condition number $\kappa(\widetilde{\mathcal{M}}_{aS,D}\mathcal{A})$ of aS methods through some parameters α and β. For now, we do not use any approximation (whose effects are the object of Section 3.5.4), hence $\widetilde{\mathcal{A}}_i = \mathcal{A}_i$ and $\omega_- = \omega_+ = 1$. In order to compare the three methods, we first choose a bound χ and we then choose α and β such that $\kappa \leq \chi$:

- for AS (respectively NN), Corollary 2 states that $\kappa \leq (1 + \alpha)N_c$ (respectively $\kappa \leq \alpha N_c$). We choose $\alpha = \chi/N_c - 1$ (respectively $\alpha = \chi/N_c$).

[3] `https://gitlab.inria.fr/solverstack/maphys/maphys`

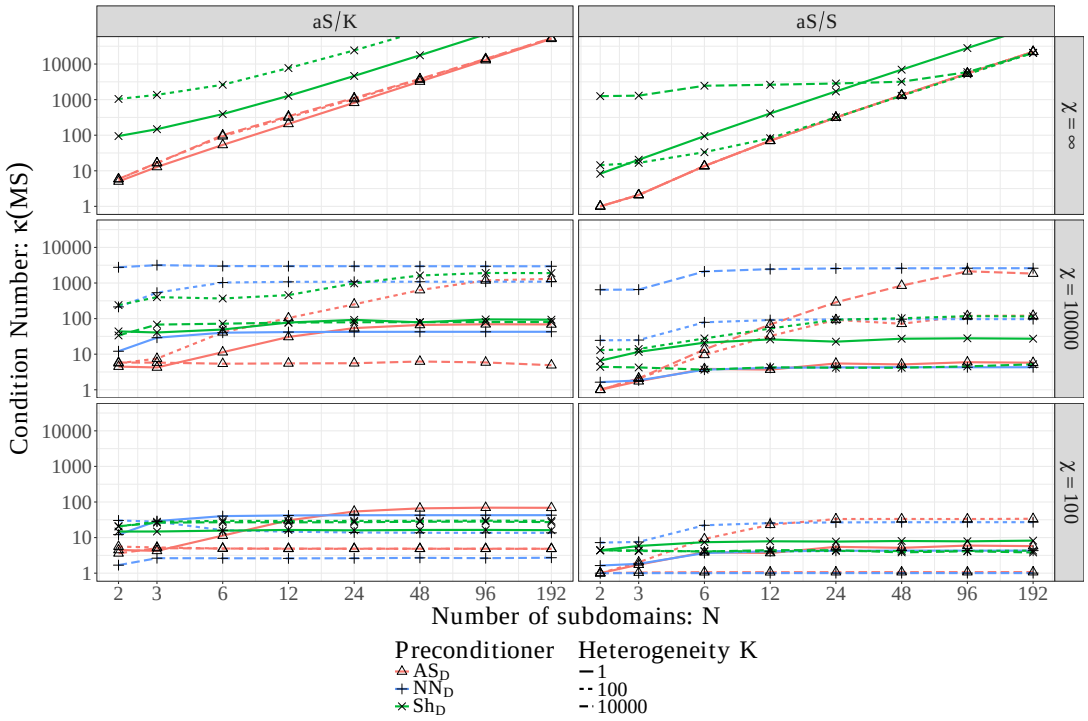


Figure 3.1: Imposing an *a priori* bound χ on the condition number using deflation. Whatever the chosen target χ, we ensure that the condition number of the iterative problem $\kappa(\mathcal{MA})$ remains below χ. Each preconditioner (AS, NN, Sh) can be applied either on the original matrix $\mathcal{K}$ (aS/$\mathcal{K}$), left, or in a substructuring context on $\mathcal{S}$ (aS/$\mathcal{S}$), right.

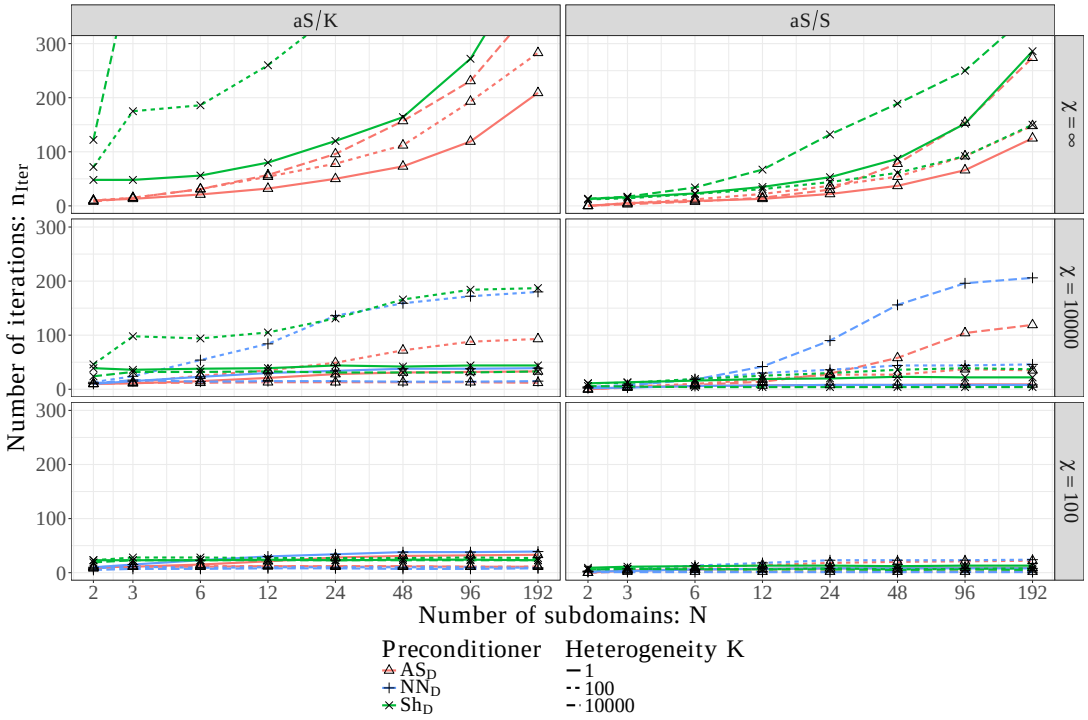


Figure 3.2: Number of iterations when imposing an *a priori* bound χ on the condition number.

- for Sh (or any other aS preconditioner), Theorem 2 states that $\kappa \leq (1+\alpha)\beta$ and we choose $\alpha = \beta = \sqrt{1/4+\chi} - 1/2$.

When we do not impose an upper bound ($\chi = \infty$), no CSC is used and results are presented only for AS and Sh. We observe (Figure 3.1) that the condition number κ grows quadratically with the number of subdomains N and that the number of iterations to reach convergence (Figure 3.2) is proportional to the number of subdomains (note the log scale for the x-axis). This lack of scalability is the main motivation for using a two-level method. We also note that, without CSC, our AS preconditioner outperforms the Sh preconditioner, especially when the heterogeneity K is high: the AS preconditioner performs a more appropriate local solve than the very basic Sh preconditioner. As expected, the condition number is also lower when working on the Schur complement matrix $\mathcal{S}$ instead of $\mathcal{K}$, since all the interior unknowns are solved using a direct method and do not appear anymore in the iterative process.

When we impose an upper bound on the condition number ($\chi = 10,000$ or $\chi = 100$), we observe that the condition number κ does indeed drop below the prescribed bound χ, independently of the number of subdomains N, the local preconditioner AS, NN or Sh, the heterogeneity K and the choice of operating on $\mathcal{K}$ or $\mathcal{S}$. However, this *a priori* control on the condition number comes at the expense of having to use a direct solve on a coarse space V_0 whose dimension can be quite large. Each subdomain computes a local coarse space V_0^i of dimension $n_v^{(i)}$ (Figure 3.3) and the size of the global coarse space therefore grows linearly with the number of subdomains. Since without deflation ($\chi = \infty$) the Sh preconditioner applied to the original matrix $\mathcal{K}$ does not perform very well in the

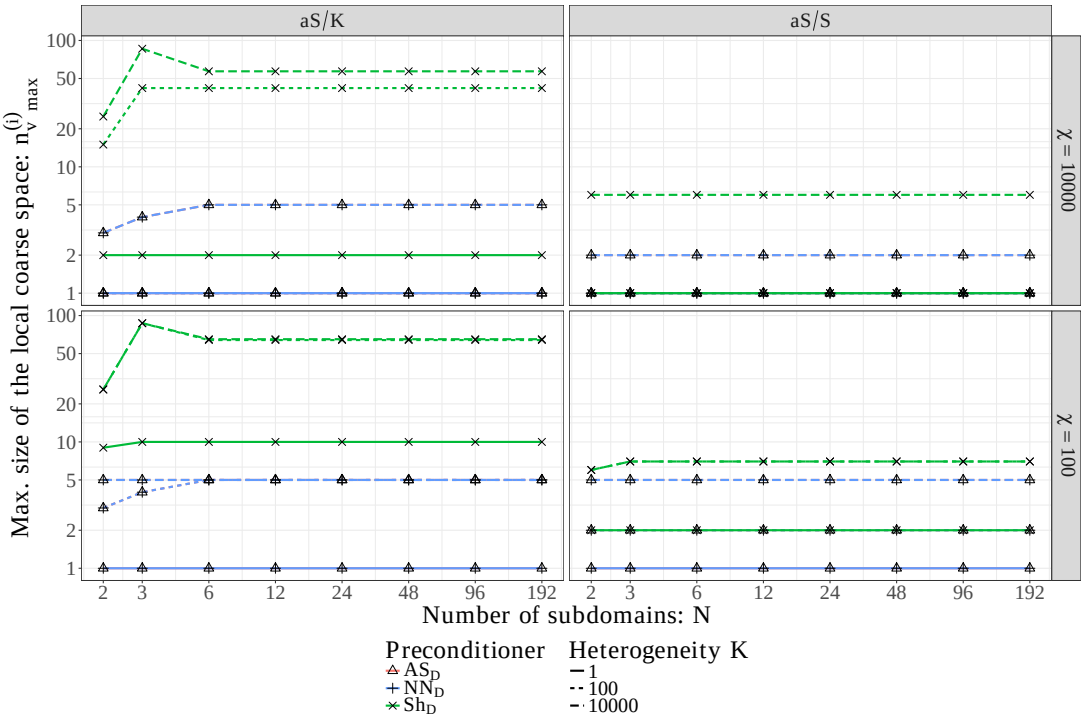


Figure 3.3: Maximum size of the local coarse space when imposing an *a priori* bound χ on the condition number. Note that AS and NN overlap with each other. In most cases, only few vectors per subdomain are enough but the least robust methods can induce a relatively large local coarse space V_0^i in some cases.

heterogeneous case, the size of the coarse space necessary to obtain a condition number below the target χ is very large (up to 87 vectors per subdomain). However, using a better local preconditioner such as AS or NN can greatly reduce the size of the coarse space, as well as working on the Schur complement matrix $\mathcal{S}$ instead of $\mathcal{K}$.

3.5.3 Imposing an *a priori* coarse space size

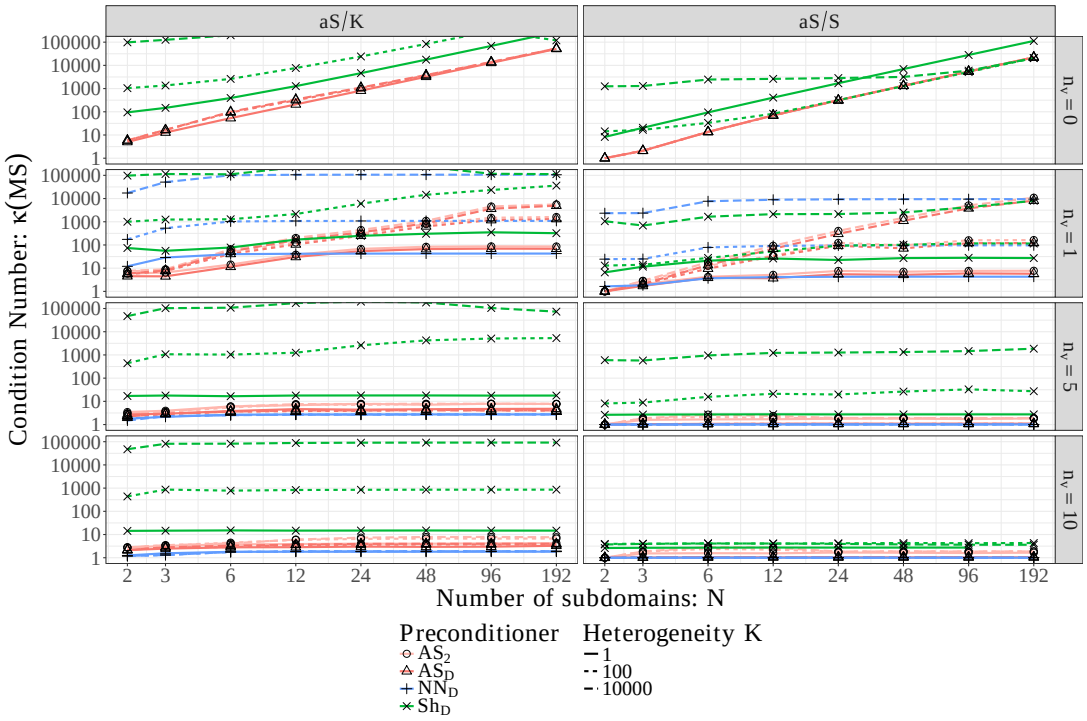


Figure 3.4: Condition number when imposing an *a priori* size n_v for the local coarse space V_0^i. We are still able to significantly reduce the condition number of the methods. The best convergence results are obtained with the AS,D/$\mathcal{S}$ method.

We showed in the previous section that we can effectively control the condition number κ of the method by building the coarse space using two parameters α and β as presented in Theorem 2. However, this can lead to an impractically large coarse space and we now consider the context where the size n_v of the local subspace in each subdomain is chosen *a priori*. Instead of choosing the coarse space by comparing the eigenvalues to a threshold, we thus keep the eigenvectors associated with the n_v smallest eigenvalues. Once the coarse space is computed, we know what threshold would have led us to keep the same number of vectors and we can get, *a posteriori*, a bound on the condition number of the method: if λ_{n_v+1} is the lowest eigenvalue corresponding to a vector not in the coarse space, Theorem 2 ensures that $\kappa(\mathcal{M}_{Sh,D}\ \mathcal{A}) \leq N_c(1 + 1/\lambda_{n_v+1})/\lambda_{n_v+1}$. As in Section 3.5.2, this bound can be improved for NN and AS preconditioners using Corollary 2 and Theorem 3:

- $\kappa(\mathcal{M}_{NN,D}\ \mathcal{A}) \leq N_c/\lambda_{n_v+1}$;
- $\kappa(\mathcal{M}_{AS,D}\ \mathcal{A}) \leq N_c(1 + 1/\lambda_{n_v+1})$;

- $\kappa(\mathcal{M}_{AS,2}\ \mathcal{A}) \leq (N_c+1)\,[N_c+1+(N_c+2)/\lambda_{n_v+1}]$.

The Schur complement matrix $\mathcal{S}$ is smaller and better conditioned (Brenner, 1999; Mansfield, 1990) than the original matrix $\mathcal{K}$. Furthermore, in a two-level domain decomposition framework, eliminating the interior unknowns significantly improves the convergence by reducing the size of the coarse space needed to take into account the physical heterogeneity in the domain (Galvis and Efendiev, 2010). In accordance with these theoretical results, Figure 3.4 highlights the benefits of operating on $\mathcal{S}$ (Figure 3.4, right) instead of $\mathcal{K}$ (left): the condition number is smaller when applying any aS method on $\mathcal{S}$ instead of $\mathcal{K}$. Without CSC ($n_v = 0$, top), the results are consistent with Figure 3.1, top ($\chi = \infty$): the condition number κ increases with the number of subdomains N. Choosing $n_v = 1$, our coarse space reduces to a classical partition-of-unity coarse space (Sarkis, 2003) and is sufficient in the homogeneous case ($K = 1$, plain lines); we notice that NN,D/$\mathcal{S}$ then reduces to classical BDD where the condition number does not depend on N but remains fairly large for large values of K. However, in the heterogeneous cases ($K = 100$ or $10,000$, dashed lines), this simpler coarse space is not enough to get a scalable method: one eigenvector per high-conductivity inclusion is needed in the coarse space to build a robust method (Galvis and Efendiev, 2010). In our case, with 5 high-conductivity layers passing through all the subdomains, $n_v = 5$ eigenvectors are enough to bound the condition number for AS/$\mathcal{S}$ and NN/$\mathcal{S}$. Using the Sh/$\mathcal{S}$ method, since two eigenproblems are solved in each subdomain, 10 vectors are needed to get a good convergence (bottom right).

With a large enough coarse space, the three methods NN,D/$\mathcal{S}$, AS,2/$\mathcal{S}$ and AS,D/$\mathcal{S}$ perform quite similarly, with a slight advantage for NN. However, when the coarse space is too small ($n_v = 1$ and $K = 10,000$ for instance), AS,2/$\mathcal{S}$ and AS,D/$\mathcal{S}$ have a significantly smaller condition number than NN,D/$\mathcal{S}$, and they appear more robust. A parallel implementation of all these methods will be presented in Chapter 4 along with an experimental study focusing on performance issues. Because of its robustness, we furthermore decided to further study the scalability of the two-level AS/$\mathcal{S}$ methods by integrating an AS,2/$\mathcal{S}$ methods within the MaPHyS fully-featured sparse hybrid solver; Section 3.5.5 provides an insight on the performance impact while Chapter 5 will present in details the various parallel strategies that have been designed to handle the CSC.

3.5.4 Approximate case: Empirical study of the impact of sparsification

The convergence results for approximate aS methods in sections 3.2, 3.3 and 3.4 apply for both aS/$\mathcal{K}$ and aS/$\mathcal{S}$ cases. However, for a matter of conciseness, we now only focus on the latter context for illustrating the impact of approximation, as the above experiments showed the numerical benefits of operating on the Schur complement. For that, we approximate the dense matrix $\mathcal{S}_i$ with a sparse matrix $\widetilde{\mathcal{S}}_i$, by dropping some entries in the matrix. This process is called *sparsification*. In a very heterogeneous medium ($K \gg 1$), some entries in $\mathcal{S}$ corresponding to couplings between unknown separated by a low-conductivity layer, are negligible. We use the symmetry-preserving strategy of dropping s_{ij} if $|s_{ij}| \leq \epsilon(s_{ii} + s_{jj})$, where ϵ is a parameter that controls the sparsity (see, e.g.,

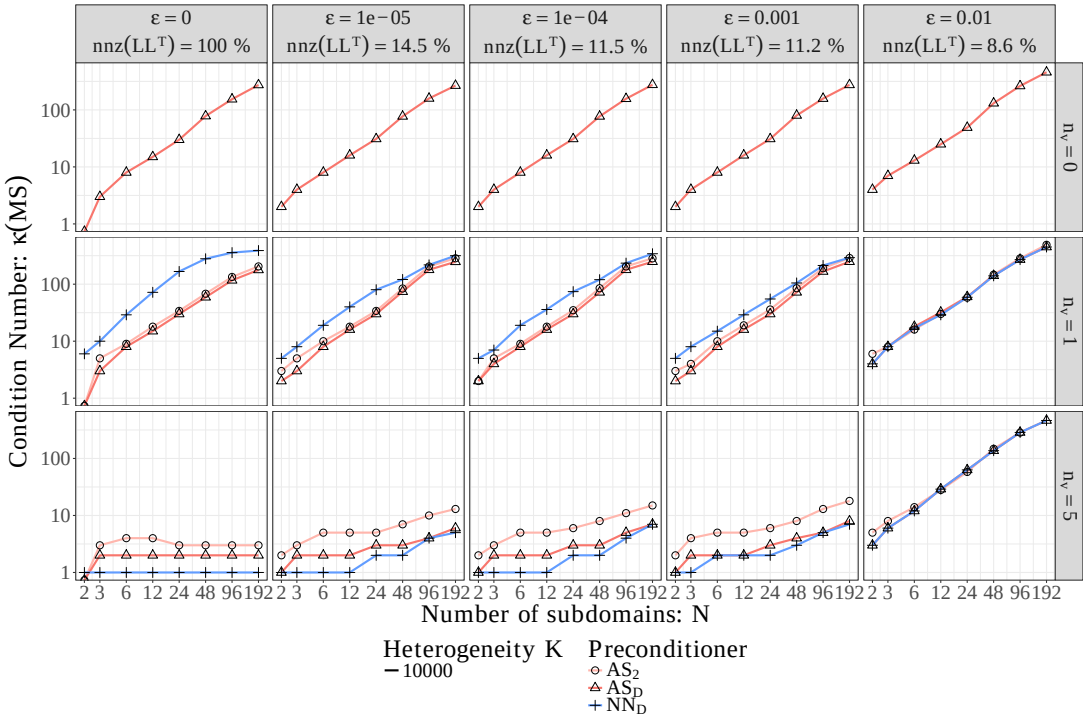


Figure 3.5: Up to a certain level, the sparsification does not break the robustness of the method: using a big enough coarse space ($n_v = 5$), it is possible to discard 88.8% of the entries in the factorization of the preconditioner without losing convergence.

(Carvalho et al., 2001b)).

The benefits of sparsification are evaluated by assessing the proportion $nnz(LL^T)$ of non-zero elements in the Cholesky factorization $\widehat{\mathcal{S}}_i = LL^T$ of the local preconditioner. In Figure 3.5, we evaluate the impact of sparsification on the robustness of the method. It appears that, up to a certain level, we are still able to find a robust coarse space despite having significantly reduced the memory footprint of the preconditioner. For instance, with a sparsity parameter of $\epsilon = 0.001$, although 88.8% of the entries in the factorization of the preconditioner are dropped, our coarse space with $n_v = 5$ vectors per subdomain still significantly improves the convergence.

These results are very promising as they show we can efficiently apply an approximate scheme to reduce the complexity of two-level aS methods. However, the considered sparsification technique is delicate for ensuring an *a priori* condition number. Approximation through hierarchical matrices (Grasedyck and Hackbusch, 2003) might better fit this objective, for bounding ω_- and ω_+ and ensure theorems 2 and 3 apply. This is left for future work (see (Agullo et al., 2018) for preliminary investigations in this direction) and we do not consider approximation techniques in the rest of this thesis.

3.5.5 Performance of AS,2/$\mathcal{S}$ on a modern parallel computer

The excellent numerical properties exhibited above by the AS,2/$\mathcal{S}$ method motivated the design of an HPC code of that variant. For that, we relied on the MaPHyS package and we

added a coarse grid correction to the baseline, one-level AS/$\mathcal{S}$ variant (Agullo et al., 2011, 2016b; Giraud et al., 2008) for the purpose of the present study. A detailed analysis of this implementation and of the resulting experimental results is provided in Chapter 5, a subset of which is presented in the current section for illustrating the potential impact on performance of the numerical issues discussed in this chapter. MaPHyS is a parallel hybrid (direct/iterative) sparse linear solver. Its *Setup* step relies on third-party sparse direct solvers for efficiently performing the elimination of the interior variables and computing the local Schur complement $\mathcal{S}_i$. Subdomains are processed concurrently, each subdomain being associated with a process. The computation of the one-level preconditioner (still within the *Setup* step) is then performed with neighbor-to-neighbor communications. The *Solve* step consists of classical preconditioned conjugate gradient iterations. In particular, global synchronizations are only required for computing dot products while the matrix-vector product and preconditioner application can be performed concurrently on each subdomain with neighbor-to-neighbor communications only. We extended MaPHyS to ensure a CSC as follows. In the *Setup* step, the generalized eigenproblems are processed concurrently on each subdomain; the matrix associated with the resulting coarse space is then assembled and factorized using a third-party parallel sparse direct solver. In the *Solve* step, a coarse solve is added in the application of the preconditioner at each iteration. Due to the nature of the coarse space, these operations add global communications and synchronizations in the algorithm and particular care must be taken in their implementation in order to achieve good scalability and parallel efficiency: several parallelization strategies for this coarse correction are discussed in Chapter 5. In the current experiment, the coarse matrix $\mathcal{A}_0$ is assembled and factorized redundantly on disjoint sub-communicators (obtained by splitting the global one) in order to reduce the number of global communications during the solve step.

We now present a weak scalability study conducted on test cases similar to the ones introduced in Section 3.5.1, but with larger subdomains. Each subdomain is indeed a cube discretized on a $31 \times 31 \times 31$ mesh with 29,791 unknowns. There are now 6 alternating conductivity layers ($K = 10,000$), and we consider a scenario with an imposed coarse space size (as in Section 3.5.3) using 3 vectors per subdomain. No approximation is performed. The same stopping criterion as above is used. The experiments have been conducted on the Occigen machine at CINES. Each node is composed of two Haswell (E5-2690V3) 12-core processors running at 2.6 GHz. A subdomain is associated with a process, binded on a CPU core. MaPHyS was compiled with Intel 17.0 and Intel MPI 2017.0.098. All dense operations are performed with the Intel Math Kernel Library (MKL) 2017 (including the Lapack `dsygvx` routine for solving the eigenproblems, that allows one to only compute a targeted subset of eigenpairs). Sparse factorizations are performed with the MUMPS 5.0.2 sparse direct solver (Amestoy et al., 2001) together with the ParMetis 4.0.3 partitioner (Karypis and Kumar, 2009).

Table 3.1 compares the behavior of our extension of MaPHyS relying on the proposed CSC described above (AS,2/$\mathcal{S}$) with the baseline, one-level version of MaPHyS. The number of subdomains N, which is equal to the number of MPI processes and CPU cores used for the respective computation, the total number of unknowns $n = (30N+1)\times 31\times 31$ and the size of the coarse space n_0 are provided in the table along with the maximum (among all subdomains) time in seconds needed to perform the *Setup* step, the *Solve* step

or both steps (*Total*) and the number of PCG iterations performed during the *Solve* step, for both the AS/$\mathcal{S}$ method (left) and the AS,2/$\mathcal{S}$ method (right). The *Setup* step includes the time spent in the factorization of the local matrices and the computation of the local Schur complement matrix using a sequential sparse direct solver, the assembly and factorization of the local Schur complement, the solution of the generalized eigenproblems, the construction and the factorization of the coarse matrix. The *Solve* step corresponds to the PCG iterations and the final computation of the interior unknowns. We observe that the addition of the coarse correction increases the *Setup* time and the individual cost of each iteration (up to a factor 2), mainly due to the induced global communications. On the other hand, the number of iterations of AS,2/$\mathcal{S}$ remains stable, leading to a drastically overall reduced *Solve* time compared to the baseline AS/$\mathcal{S}$ method (up to a factor 37 when the 44,283,841 unknowns are distributed among 1,536 subdomains). As a consequence, in a scenario consisting of solving a linear system with a single right-hand side, using a coarse space reduces the total time to solution (*Setup* + *Solve*) when the number of subdomains (and CPU cores) is equal to or higher than 384. In another common application scenario where multiple (say, p), successive, right-hand sides must be solved, the total time to solution (*Setup* + p *Solve*) may then essentially be governed by the *Solve* step if p is large. In that latter case, the benefits of the CSC may then thus be tremendous on large scale computers.

Table 3.1: A weak scalability study was performed using the MaPHyS parallel solver. The Setup, Solve and Total times are the max among all subdomains, in seconds (s). Each subdomain is associated with one MPI process binded onto one CPU core. N is the number of subdomains, n is the size of $\mathcal{K}$ and n_0 is the size of the coarse space. Without coarse correction, the *Setup* time remains stable, whereas the *Solve* time grows linearly with the number of subdomains. The coarse correction adds to the *Setup* time but keeps the number of iterations constant, thus improving the scalability. Without coarse correction, no convergence was achieved on 3,072 subdomains.

			AS/$\mathcal{S}$				AS,2/$\mathcal{S}$			
N	n	n_0	*Setup*	*Solve*	*Total*	# iter	*Setup*	*Solve*	*Total*	# iter
24	692k	72	3,64	0,47	4,12	*33*	6,13	0,30	6,44	*15*
48	1.4M	144	3,67	0,87	4,54	*62*	6,52	0,30	6,83	*15*
96	2.8M	288	3,79	1,62	5,41	*119*	6,52	0,31	6,84	*15*
192	5.6M	576	3,75	3,17	6,92	*233*	6,59	0,33	6,92	*15*
384	11.1M	1.1k	3,87	5,02	8,90	*371*	6,61	0,32	6,93	*14*
768	22.1M	2.3k	3,78	8,30	12,1	*609*	6,61	0,33	6,95	*14*
1536	44.3M	4.6k	4,13	15,1	19,2	*1,077*	6,96	0,40	7,38	*14*
3072	88.6M	9.2k	-	-	-	-	7,24	0,42	7,70	*14*

3.6 Conclusion

In this chapter, we have proposed a new class of aS preconditioners, so-called approximate aS preconditioners. These preconditioners are fully algebraic in the sense that they

do not require any other information apart from SPSD subdomain matrices. This class is wide as it consists of all aS preconditioners, provided that their coarse space results from the assembly of local components that contain the kernel of some local operators (Definition 1). In particular, it generalizes the class of approximate NN preconditioners introduced in (Le Tallec and Vidrascu, 1998) (named *generalized NN* in the original paper). We exhibited a bound on the condition number of all approximate deflated aS preconditioners (Theorem 1). This bound depends on generalized Rayleigh quotients and generalizes the result from (Le Tallec and Vidrascu, 1998) beyond the class of approximate NN methods. Applying a GenEO procedure on two well chosen generalized eigenproblems, we proposed to explicitly compute the coarse space while bounding these Rayleigh quotients leading to a bound on the condition number (hence on the number of iterations of PCG) independent of the number of subdomains. We also showed that a similar bound can be obtained when the coarse space is applied additively for the subclass of newly introduced approximate AS methods.

The results presented in this chapter can be readily derived to retrieve the bounds previously obtained via generalized eigenproblems in the particular cases of AS/$\mathcal{K}$ (Efendiev et al., 2012; Spillane et al., 2014a), NN/$\mathcal{S}$ (Spillane and Rixen, 2013) and optimized Robin (SORAS) (Haferssas et al., 2017). It also generalizes these results when used with approximate local solvers. Furthermore, they allowed us to define a coarse space for the AS method applied on the Schur complement (AS/$\mathcal{S}$) (Carvalho et al., 2001b), leading to an extremely robust substructuring method, for which the CSC can be applied either with deflation or additively, and with the freedom of relying on an approximate local Schur complement. A challenge opened by the present study is to determine an explicit procedure to perform the approximation while achieving a given *a priori* bound on the condition number. We also plan to study the effects of the method on the spectrum and on the empirical convergence of non symmetric test cases.

These statements were illustrated by numerical experiments, that were made possible by a parallel implementation of these methods in python, presented in detail in Chapter 4. These experiments motivated an HPC design of a CSC for AS/$\mathcal{S}$ within the MaPHyS package as presented later in Chapter 5. The presented sample of these parallel experiments results showed the significant benefits that the resulting AS,2/$\mathcal{S}$ solver could bring.

Chapter 4

Design of a domain decomposition toolbox in python

4.1 Introduction

In Chapter 2, we introduced various Domain Decomposition (DD) formulations and preconditioners that can be used to solve a PDE or a linear system. Some of these Domain Decomposition Methods (DDM), which can be expressed as abstract Schwarz (aS) methods, were studied in Chapter 3 where we gave a proof of convergence of two-level aS methods with an adaptive Coarse Space Correction (CSC). In the present chapter, we present a parallel implementation of aS methods and other DDM. The parallel design of the CSC in an HPC hybrid solver is discussed in Chapter 5.

In order to design a DD solver various questions have to be addressed: shall a primal or dual formulation be used? Shall the problem be reduced to an interface problem (Schur complement matrix)? Will a N-Lagrange formulation perform better? Once these questions are answered and a particular DDM is chosen, other questions regarding the implementation arise: should the DD solver be developed from scratch, or, can an external specialized library be used? How should the parallelism be handled? The overall objective is to have a parallel solver that is both numerically and computationally effective.

These choices give rise to two competing objectives for designing an efficient DD solver: one needs to optimize the mathematical method on the one hand, and, on the other hand, its HPC implementation. The first objective requires an ability to develop several DDM and compare them with their variants, in order to tune their numerical features. The required flexibility to easily implement these various numerical methods may contradict the second objective of optimizing the code to improve its computational performance. We can often observe a significant gap between DD numerical prototypes built by researchers in interpreted languages such as Matlab, and DD codes embedded within industrial applications in compiled languages such as C, C++ or Fortran because of the separation of concerns that prevent to address the numerical and computational efficiencies at once.

In this chapter, we demonstrate that it is possible to get both flexibility and performance using the python language to build DDM: these methods (presented in Chapter 2) can be expressed as well-chosen combinations of individual operations (solving a Dirich-

let, Neumann or Robin problem, computing a Schur complement matrix or a preconditioner, ...). If the DDM are applied after discretization and follow the hypotheses in Section 3.2.1, these individual operations can be expressed algebraically as linear algebra operations, or basic kernels, such as matrix-vector products or matrix factorizations for instance. The optimization of such kernels is in fact out of the scope of DDM implementations since they appear in many other applications, and it is faster and safer (both regarding the implementation time and the execution time) to call external optimized implementations for these generic algebraic operations. It is therefore possible to implement a DD solver such that most of the time spent in the solver is actually not spent executing the solver's code *per se* but inside these compute-intensive external kernels. As such, the optimization of a DD solver's implementation mainly consists in making sure that state-of-the-art HPC implementations for these kernels are used; this implies an ability to test and compare these various implementations and the second objective expressed above now aligns with the first objective: more *flexibility* can improve *performance*.

We developed a parallel DD toolbox in python named `ddmpy` and focused on these two objectives: DD algorithms and high-level structures benefit from the expressivity of the language while computational operations are performed using optimized external libraries in order not to compromise on performance on the range of interest. The relevance of this strategy is assessed through a performance study of two basic linear algebra operations (vector addition and matrix multiplication) and two DD operators (Schur complement matrix computation and Conjugate Gradient (CG) algorithm) in Section 4.2. Then the design and implementation of the `ddmpy` DD toolbox in python is presented in Section 4.3, and its parallel performance on several test cases is studied in Section 4.4.

4.2 Performance of some elemental operators in python

It is often accepted as a fact that python is not a fast language: even the creator of the language Guido van Rossum wrote: *I will gladly admit that Python is not the fastest running scripting language. It is a good runner-up though. With ever-increasing hardware speed, the accumulated running time of a program during its lifetime is often negligible compared to the programmer time needed to write and debug it. This, of course, is where the real savings can be made. While this is hard to assess objectively, Python is considered a winner in coding time by most who have tried it. In addition, many consider using Python a pleasure – a better recommendation is hard to imagine.*

Although the python language was not meant to be fast, we show in sections 4.2.1 to 4.2.3 that it is relevant for HPC: several implementations, in C and in python, for computing a vector addition (Section 4.2.1.1) and a matrix multiplication (Section 4.2.1.2) are compared: native implementations with loops in C and python are compared with calls to the Intel MKL BLAS library from both languages, natively or through the `numpy` package in python. The performance of python code can also be improved by using a compiler such as cython (Behnel et al., 2011). For each of the two considered operations, the native python implementation with loops (executed with the standard CPython interpreter) is compared to the same code compiled with cython and with a hybrid C/python version, also compiled with cython, where the internal variables in the functions are declared with

C types. Other python compilers such as pythran (Guelton et al., 2015) or numba (Lam et al., 2015) are not considered in this study. A performance comparison of two implementations for computing a Schur complement matrix (Section 4.2.2) and solving a linear system using the Conjugate Gradient algorithm (Section 4.2.3) complete this analysis.

4.2.1 Comparison of C and python for basic linear algebra operations

4.2.1.1 Vector addition in C and python (daxpy)

When adding two vectors of size n, $2n$ read operations, n write operations, and n additions are performed: each computation requires 3 data accesses. The performance is here essentially governed by the time needed to transfer data between the CPU and the memory: it is said to be *memory-bound*, by opposition to compute-intensive operations such as the matrix product in Section 4.2.1.2 which is said to be *compute-bound*. Several implementations for the addition are presented in listings 1 to 6 and compared.

```
1 void loop__C(const double* x, double* y, const unsigned int n) {
2   for (unsigned int i = 0; i < n; i++) {
3     y[i] += x[i];
4   }
5 }
```

Listing 1: `loop/C`: straightforward implementation of $y \leftarrow x + y$ using loops in C.

```
1 void blas__C(const double* x, double* y, const unsigned int n) {
2   cblas_daxpy(n, 1., x, 1., y, 1);
3 }
```

Listing 2: `blas/C`: HPC implementation of $y \leftarrow x+y$ using the optimized BLAS function (`daxpy`) as provided by the Intel MKL library called from C.

```
1 def loop__python(x, y):
2     for i in xrange(len(x)):
3         y[i] += x[i]
4     return y
```

Listing 3: `loop/python`: straightforward implementation of $y \leftarrow x + y$ using loops in python. This code can be compiled with cython, yielding the `loop/cython` version.

The time needed for each of these implementations to perform the operation $y \leftarrow y+x$ where x and y are two vectors of size n are presented in Figure 4.1, and the number of

```
def loop_type__cython(x, y):
    cdef int i, n = len(x)
    cdef double *x_ = <double *>malloc(n * sizeof(double))
    cdef double *y_ = <double *>malloc(n * sizeof(double))
    for i in xrange(n):
        x_[i] = x[i]
        y_[i] = y[i]
    for i in xrange(n):
        y_[i] += x_[i]
    y = [y_[i] for i in xrange(n)]
    free(x_)
    free(y_)
    return y
```

Listing 4: `loop_type/cython`: mixed implementation of $y \leftarrow x + y$ using loops with typed data using cython (the python list are converted to C arrays before and after the computations).

```
def numpy__python(x, y):
    y += x
    return y
```

Listing 5: `numpy/python`: implementation of $y \leftarrow x + y$ using the `numpy` module in python, where the input are of type `numpy.ndarray` instead of `list`.

```
def blas__python(x, y):
    return scipy.linalg.blas.daxpy(x, y)
```

Listing 6: `blas/python`: implementation of $y \leftarrow x + y$ using the low-level `scipy.linalg.blas` wrapper, linked with the Intel MKL library, in python, with `numpy.ndarray` as input.

floating-point operations per second (flop/s) are presented in Figure 4.2 (both figures use log scale on both axes). The relative performance of the different implementations compared to the `blas/C` implementation are given in Table 4.1.

The comparison of the straightforward implementations with loops (in green in the figures) shows, as should be expected, that the C version (`loop/C` from Listing 1, plain line) far outperforms the python version (`loop/python` from Listing 3, dashed line) by a factor 142 ($n = 1.0\ 10^6$). The same python code, when compiled using cython (`loop/cython`, still from Listing 3, dotted line), is 7.2 times faster (but still 20 times slower than C). Adding type information in cython and converting the python list to a C array (`loop_type/cython` from Listing 4, blue line) only slightly improves the performance because the time needed to go through the python lists remains the same; furthermore, the resulting hybrid code is much less readable than both the corresponding python and C code. The overhead of all these python versions compared to the C versions comes from the fact that they operate on native python lists that are not contiguous in memory as C vectors are.

Instead of coding the addition using a loop, it is possible to call an external optimized library such as the Intel MKL BLAS library: the vector addition is available as a `daxpy` function (in red in the figures). This external function can be called from C (`blas/C` from Listing 2, plain line); although calling this external function introduces some overhead for small vectors, the `blas/C` function outperforms the `loop/C` implementation for vectors of intermediate size that fit in the cache memory. For larger vectors, both implementations achieve the same performance bounded by the memory speed. The blas function can also be called from python on `numpy.ndarray` vectors, either explicitly through a wrapper in the `scipy.linalg.blas` module (`blas/python` from Listing 6, dashed line) or implicitly by adding the arrays (`numpy/python` from Listing 5, purple line). Both these python implementations perform as well as the `blas/C` version.

Table 4.1: Relative performance of the different implementations for the vector addition for the largest size n available.

blas C	1
blas python	0.98524
loop C	0.9957
loop cython	0.05078
loop python	0.00703
loop_type cython	0.03139
numpy python	0.99193

4.2.1.2 Dense matrix multiplication in C and python (dgemm)

The operation $C \leftarrow AB + C$, where A, B and C are three (n, n) matrices requires $3n^2$ read operations, n^2 write operations, n^3 additions and n^3 multiplications. Each computation thus requires $2/n$ data transfers. If n is large enough, these data transfers become negligible and the performance of this operation is governed by the computational power of the CPU (Irony et al., 2004; Jia-Wei and Kung, 1981; Langou, 2014): it is said

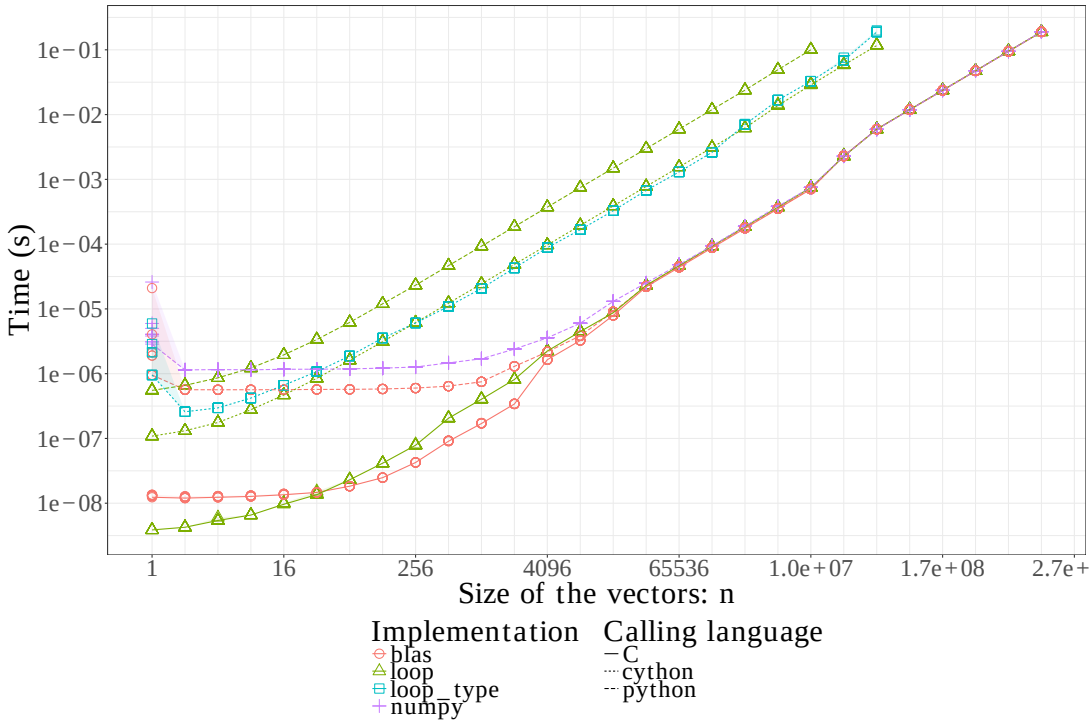


Figure 4.1: Time needed to perform a vector addition (`daxpy`).

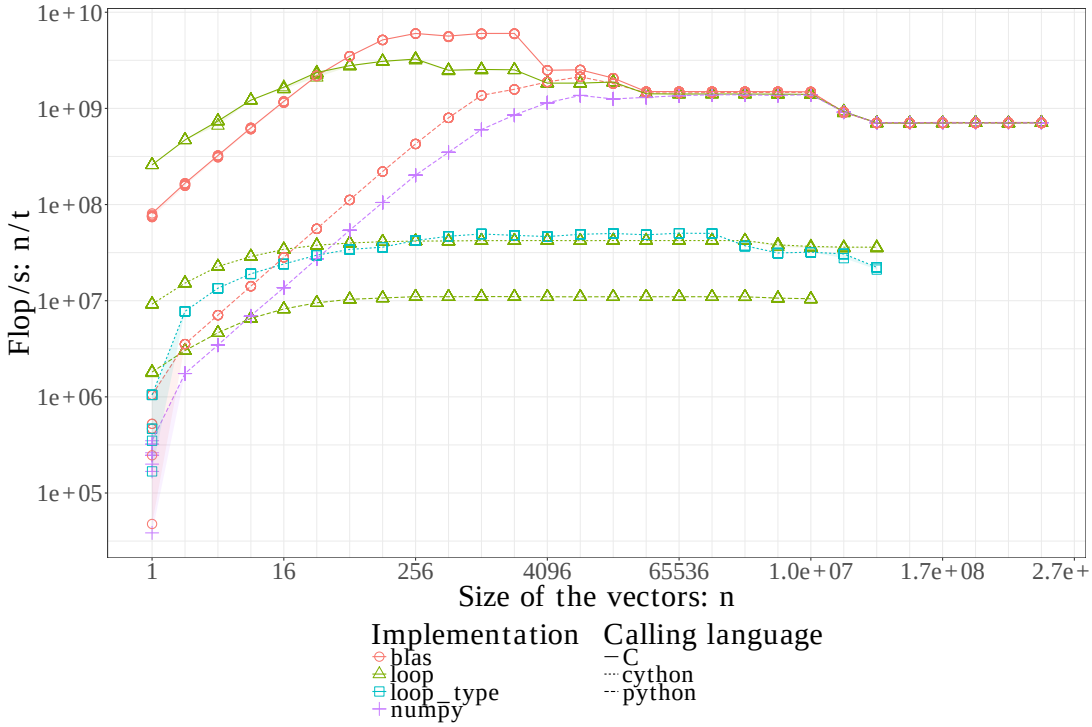


Figure 4.2: Performance of the vector addition (`daxpy`).

to be *compute-bound*. As such, there is more room for optimization by reordering the operations to minimize the data transfers between the CPU, the different levels of cache and the main memory.

The different implementation presented in listings 7 to 12 are very similar to their counterpart for the vector addition. Note that in the `numpy/python` implementation, the `@` operator can be used to represent the matrix multiplication operation in python3[1]. The notation `A @ B` is not available in `python2` and it can be replaced by `A.dot(B)`.

```
void loop__C(const double* A, const double* B, double* C, const unsigned int n) {
  for (unsigned int i = 0; i < n; i++){
    for (unsigned int j = 0; j < n; j++){
      for (unsigned int k = 0; k < n; k++) {
        C[j + n*i] += A[k + n*i] * B[j + n*k];
      }
    }
  }
}
```

Listing 7: `loop/C`: straightforward implementation of $C \leftarrow AB + C$ using loops in C.

```
void blas__C(const double* A, const double* B, double* C, const unsigned int n) {
  cblas_dgemm(CblasRowMajor,
              CblasNoTrans, CblasNoTrans,
              n, n, n, 1.,
              A, n, B, n,
              1., C, n);
}
```

Listing 8: `blas/C`: HPC implementation of $C \leftarrow AB + C$ using the optimized BLAS function (`dgemm`) as provided by the Intel MKL library called from C.

The results presented in figures 4.3 and 4.4 and in Table 4.2 show that the optimized MKL `dgemm` routine used in the `blas/C` version (Listing 8) is 60 times faster than the `loop/C` version (Listing 7), 990 times faster than the `loop_cython` version (Listing 9), 2,857 times faster than the `loop/python` version (Listing 9). The `loop_type/cython` implementation (Listing 10) performs quite similarly to the `loop/C` version: the overhead of converting the python list to C arrays is negligible for large matrices since it involves $O(n^2)$ data movements and the matrix product involves $O(n^3)$ operations. The `numpy/python` version (Listing 11) is only 5% slower than the `blas/C` version. However, the `blas/python` (Listing 12) version is 30% slower. This may come from some known restrictions on the wrapper issue[2] (this performance overhead was not investigated further).

[1] see `https://www.python.org/dev/peps/pep-0465/`

[2] see issue `https://github.com/scipy/scipy/issues/6779`

```python
def loop__python(A, B, C):
    n = len(A)
    for i in xrange(n):
        for j in xrange(n):
                for k in xrange(n):
                    C[i][j] += A[i][k] * B[k][j]
```

Listing 9: `loop/python`: straightforward implementation of $C \leftarrow AB + C$ using loops in python. This code can be compiled with cython, yielding the `loop/cython` version.

```python
def loop_type__cython(A, B, C):
    cdef int i, j, k, n = len(A)
    cdef double *A_ = <double *>malloc(n*n * sizeof(double))
    cdef double *B_ = <double *>malloc(n*n * sizeof(double))
    cdef double *C_ = <double *>malloc(n*n * sizeof(double))
    for i in xrange(n):
        for j in xrange(n):
            A_[j + n*i] = A[i][j]
            B_[j + n*i] = B[i][j]
    for i in xrange(n):
        for j in xrange(n):
            for k in xrange(n):
                C_[j + n*i] += A_[k + n*i] * B_[j + n*k]
    for i in xrange(n):
        for j in xrange(n):
            C[i][j] = C_[j + n*i]
    free(A_)
    free(B_)
    free(C_)
```

Listing 10: `loop_type/cython`: mixed implementation of $C \leftarrow AB + C$ using loops with typed data using cython (the python list are converted to C arrays before and after the computations).

```python
def numpy__python(A, B, C):
    C += A.dot(B) # C += A @ B in python3
    return C
```

Listing 11: `numpy/python`: implementation of $C \leftarrow AB + C$ using the `numpy` module in python, where the input are of type `numpy.ndarray` instead of `list`.

```
def blas__python(A, B, C):
    return scipy.linalg.blas.dgemm(1., A, B, 1., C, 1, 0, 0)
```

Listing 12: `blas/python`: implementation of $C \leftarrow AB + C$ using the low-level `scipy.linalg.blas` wrapper, linked with the Intel MKL library, in python, with `numpy.ndarray` as input.

These two experiments on $y \leftarrow x + y$ and $C \leftarrow AB + C$ show that, as expected, the implementation in python of an intensive computational operation is much slower than the corresponding code in C. However, independently of the choice of the language, the best performance is achieved when calling an external optimized library for computation-intensive kernels. The overhead of calling such a library from python, which is of order 10^{-6} s. in our experiments, becomes negligible when the time spent inside the kernel is large enough. As a result, the python language is relevant for implementing DDM, as long as these methods can be expressed as a combination of elemental operations that can be performed using such optimized kernels. The `numpy` and `scipy` packages provide HPC implementations for many of these elemental operations. Using a python compiler such as cython would be a promising option for developing other kernels in python if needed. However, we choose not to implement any low-level operation and instead rely on available state-of-the-art HPC libraries: although compiled python code can be as fast as native C code (as shown for instance in Table 4.2), it is much more efficient (both in implementation time and computation time) to rely on a highly optimized third party library such as the Intel MKL library.

Table 4.2: Relative performance of the different implementations for the matrix multiplication for the largest size n available.

blas C	1
blas python	0.72309
loop C	0.01661
loop cython	0.00101
loop python	0.00035
loop_type cython	0.02936
numpy python	0.94456

4.2.2 Comparison of Scipy and Pastix for computing the Schur complement matrix

A key step of some DDM is the computation of a Schur complement matrix using the formula $\mathcal{S}_i = \mathcal{K}^{(i)}_{\Gamma_i\Gamma_i} - \mathcal{K}_{\Gamma_i\mathcal{I}_i}\mathcal{K}^{-1}_{\mathcal{I}_i\mathcal{I}_i}\mathcal{K}_{\mathcal{I}_i\Gamma_i}$ as explained in Section 2.4. An implementation using the `scipy` module is compared with an implementation that internally uses the `Pastix` sparse direct solver (Hénon et al., 2002). These implementations are compared on two test cases: a 2D square domain of n^2 unknowns, whose interface Γ has $4n - 4$ unknowns,

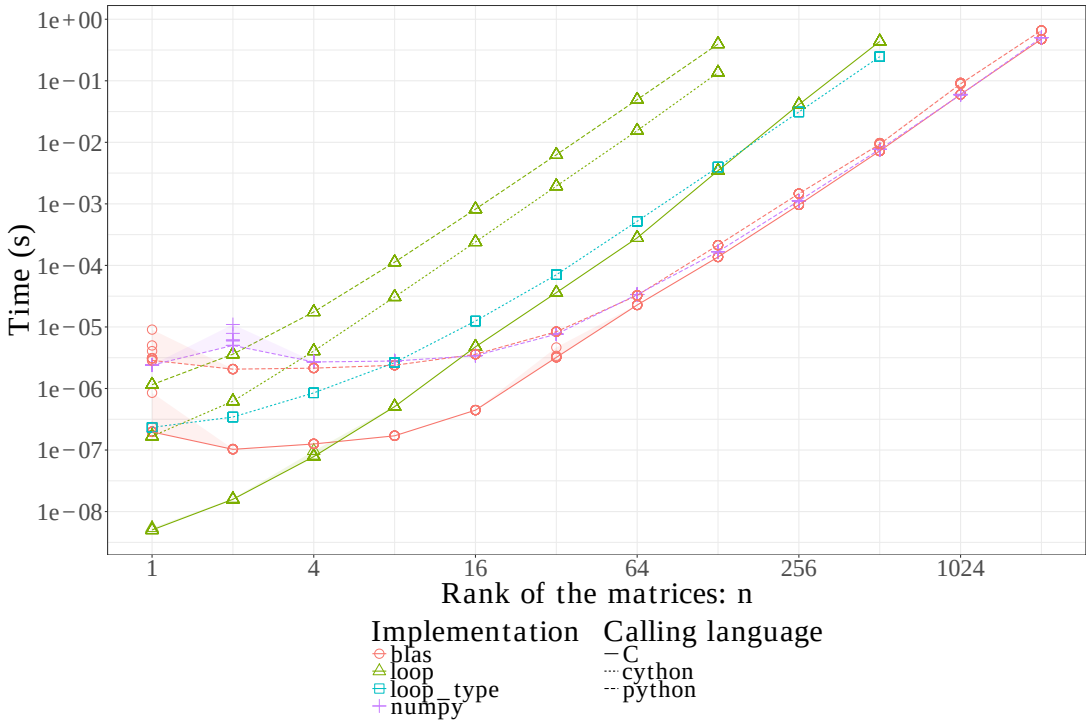


Figure 4.3: Time needed to perform a matrix multiplication (`dgemm`).

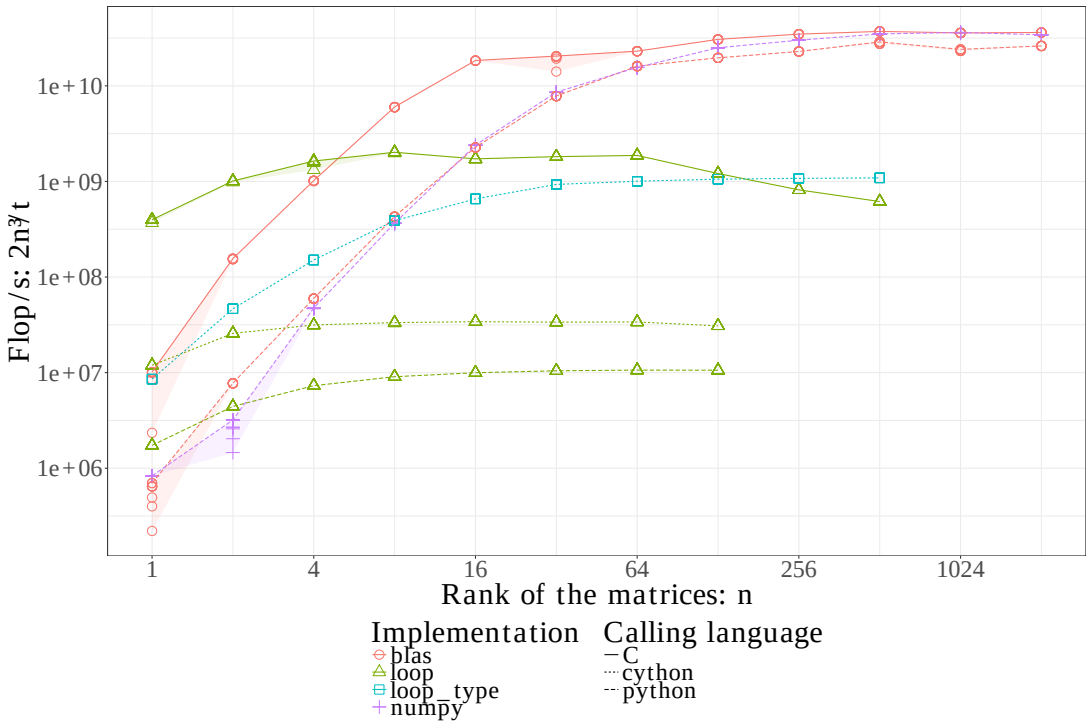


Figure 4.4: Performance of the matrix multiplication (`dgemm`)

and a 3D cubic domain of n^3 unknowns, whose interface Γ is of size $6n^2 - 12n + 8$.

Since there is no built-in Schur complement function in `scipy`, it is implemented in three steps: first, the `scipy.sparse.linalg.factorized()` method is used to factorize $\mathcal{K}_{\mathcal{I}_i\mathcal{I}_i}$ (internally calling the SuperLU sparse direct solver (Demmel et al., 1999)). Then, a `solve` operation is performed on $\mathcal{K}_{\mathcal{I}_i\Gamma_i}$ to compute $\mathcal{K}_{\mathcal{I}_i\mathcal{I}_i}^{-1}\mathcal{K}_{\mathcal{I}_i\Gamma_i}$. The Schur complement matrix can then be computed with one matrix-matrix product and a matrix subtraction. In the second step, the `solve()` method only accepts a dense argument and the sparse block $\mathcal{K}_{\mathcal{I}_i\Gamma_i}$ has to be converted to a dense representation.

In the `pastix` implementation, the whole matrix $\mathcal{K}_i$ is provided to the sparse direct solver along with a list of interface unknowns, and the Schur complement matrix $\mathcal{S}_i$ is computed as a byproduct of the partial factorization of $\mathcal{K}_i$. Both implementations are presented in more details in sections 4.3.8.1, 4.3.8.4 and 4.3.10.1.

The results presented in Figure 4.5 show that the `pastix` version is consistently faster than the `scipy` version: computing the Schur complement matrix of a 3D $32 \times 32 \times 32$ cube takes 4 min. 40 s. with scipy, compared to only 8.6 s. using Pastix. In the following, the Pastix solver is used for all sparse factorizations and Schur complement matrix computations.

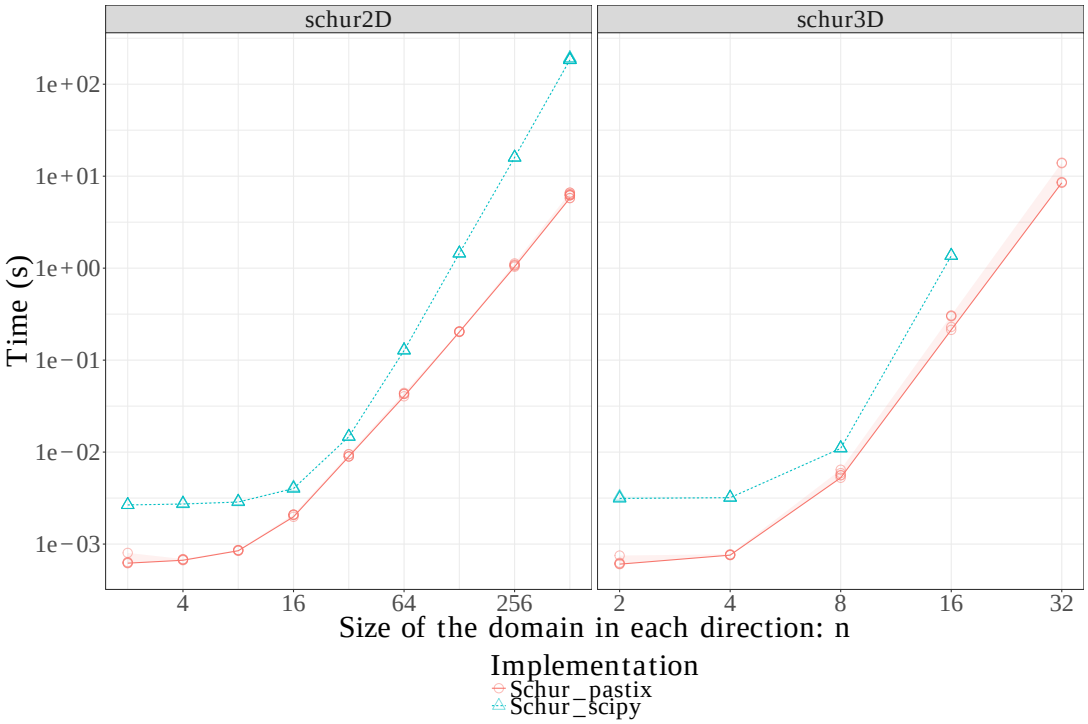


Figure 4.5: Time needed to compute the Schur complement matrix. The matrix is the FEM discretization of a 2D with n^2 unknowns or a 3D domain with n^3 unknowns. The Schur complement matrix is of size $4(n-1)$ or $6n^2 - 12n + 8$, respectively.

Table 4.3: Relative performance of the different implementations for the Schur complement matrix computation.

Schur_pastix python schur2D	1
Schur_scipy python schur2D	0.0315
Schur_pastix python schur3D	1
Schur_scipy python schur3D	0.1566

4.2.3 Comparison of Scipy and a custom implementation of the conjugate gradient

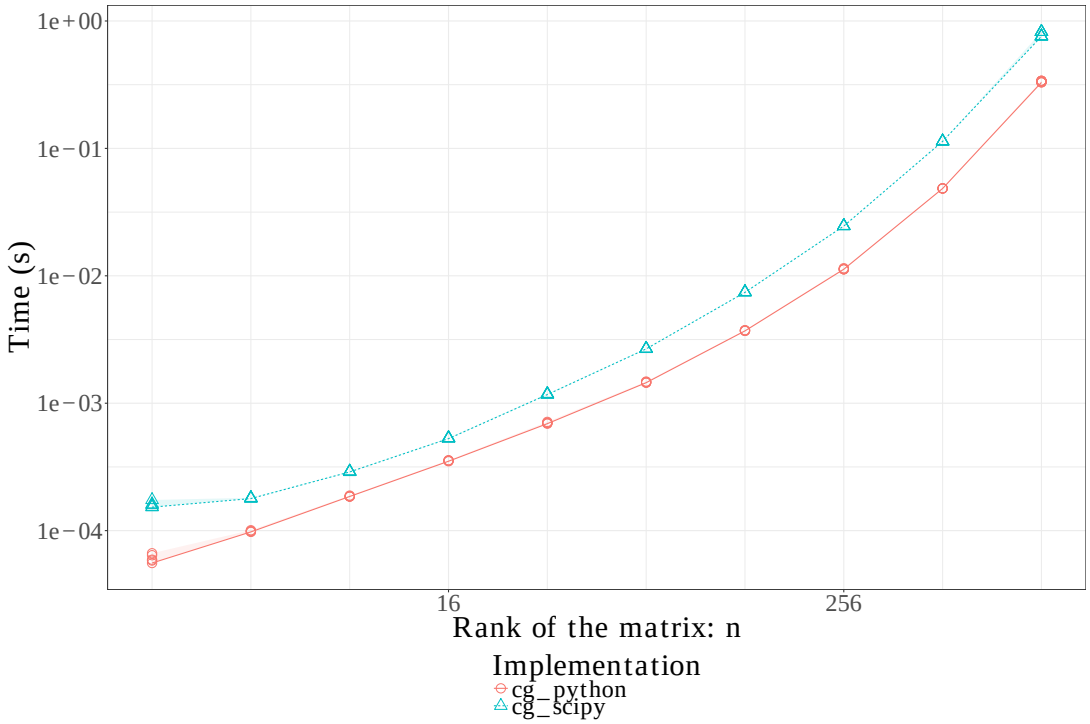


Figure 4.6: Comparison of the `scipy.sparse.linalg.cg` that calls a CG implementation in Fortran and a custom python implementation.

Another missing feature in `scipy` is the ability to use a parallel iterative solver such as the CG algorithm. Indeed, the provided routine, which calls an external Fortran routine (Barrett et al., 1994), was not designed to be used in a distributed environment: although the user can provide a distributed implementation for the matrix-vector product and the preconditioner application (through the `.dot()` methods of the relevant arguments parameters), the dot products between vectors are performed in sequential inside the routine by calling the relevant Blas routines and cannot be overridden by the user.

In order to use the CG algorithm on distributed matrices and vectors, we rewrote it in python, as shown in Listing 13. This function has the exact same signature as the `scipy.sparse.linalg.cg` function, and its implementation seems purely sequential (no explicit parallelism). However, the matrix-vector products and the scalar products

Table 4.4: Relative performance of the different implementations for the Conjugate Gradient algorithm.

cg_python python	1
cg_scipy python	0.435

performed in lines 3, 6, 7, 9, 11, 13, 14, 20, 22 and 23 can be performed in parallel by overriding the `dot` methods used by the types of `A` and `b`.

A comparison of both implementations (in sequential) for solving a dense system without a preconditioner shows that our implementation (`cg_python`) is faster than the Fortran version (`cg_scipy`), as presented in Figure 4.6. This might be explained by the fact that the `cg_scipy` implementation is always switching back-and-forth between python and Fortran: when the user enters `x = scipy.linalg.cg(A, b)`, the data are transferred to the Fortran routine. Each time a matrix-vector product should be computed or the preconditioner should be applied, the Fortran routine terminates, and the python wrapper calls the user-provided implementations for these operations before calling again the Fortran routine. If `A` and `b` are `numpy` arrays, these operations are in fact performed using the same BLAS routines as in the Fortran version. Furthermore, our function is more permissive on the input since it relies (implicitly) on the built-in python exception system to handle all errors, whereas the Fortran wrapper has to perform more checks on the data. These results confirm the overall relevance of the python language for writing a DD toolbox, as proposed in Section 4.3.

```
1  def cg(A, b, x0=None, tol=1e-5, maxiter=None,
2         xtype=None, M=None, callback=None):
3      bb = b.T @ b)
4      maxiter = len(b) if maxiter is None else maxiter
5      x = 0 * b if x0 is None else x0
6      r = b - A @ x
7      if r.T @ r <= tol * tol * bb:
8          return x, 0
9      z = r if M is None else M @ r
10     p = z.copy()
11     rz = r.T @ z
12     for i in range(maxiter):
13         Ap = A @ p
14         pAp = p.T @ Ap
15         alpha = (rz / pAp)[0,0]
16         x += alpha * p
17         r -= alpha * Ap
18         if callback is not None:
19             callback(x)
20         if r.T @ r <= tol * tol * bb:
21             return x, 0
22         z = r if M is None else M @ r
```

```
23          rz, rzold = r.T @ z, rz
24          beta = (rz / rzold)[0,0]
25          p = z + beta * p
26      return x, i
```

Listing 13: A simple CG implementation in python (`cg_python`) with the same signature as the `scipy.sparse.linalg.cg` function.

4.3 Design of the ddmpy domain decomposition toolbox in python

4.3.1 Introduction

The `ddmpy` package is an algebraic DD toolbox written in python. It is compatible with python 2.7 and python 3.4+. We chose to write this package using a programming paradigm known as *literate programming* (Knuth, 1984) in which the focus is shifted from writing the code to explaining the code, using *org-mode* (Dominik, 2010; Schulte and Davison, 2011). Following a workflow inspired by the work of Stanisic and Legrand (2014), the experiments in this chapter are reproducible: the `.org` source of this document (which is just a text file following a special markup syntax) contains the code of `ddmpy` presented in the present section and the scripts needed to launch the experiments, analyze the results and produce the figures in Section 4.4. Note that the experiments in Chapter 3 were performed using a previous version of the `ddmpy` toolbox and are not fully reproducible as the experiments in the present chapter are.

The `ddmpy` module, provided in this document, is available under the CeCILL-C license (see source).

Some specific vocabulary is introduced in Section 4.3.2 before exposing the `ddmpy` toolbox. In Section 4.3.3, the required and optional dependencies are introduced (`import` statements). Then, a function for handling neighbor-to-neighbor communications and a hierarchical profiler are presented in sections 4.3.4 and 4.3.5. The classed used to handle the domain decomposition and distributed data (matrices and vectors) are defined in Section 4.3.6, before the various solvers in `ddmpy` are introduced in sections 4.3.7 to 4.3.11. One-level and two-level DD preconditioners are then presented in Section 4.3.12.

4.3.2 Some important concepts in the Python language

The code of the `ddmpy` package uses several advances python features that are briefly described below. More information can be found in the official python documentation[3].

- an *iterator* is any object that behaves like a list and that can be iterated upon, implicitly in a `for` loop or explicitly using the `next()` function.

[3] `https://docs.python.org/3/glossary.html`

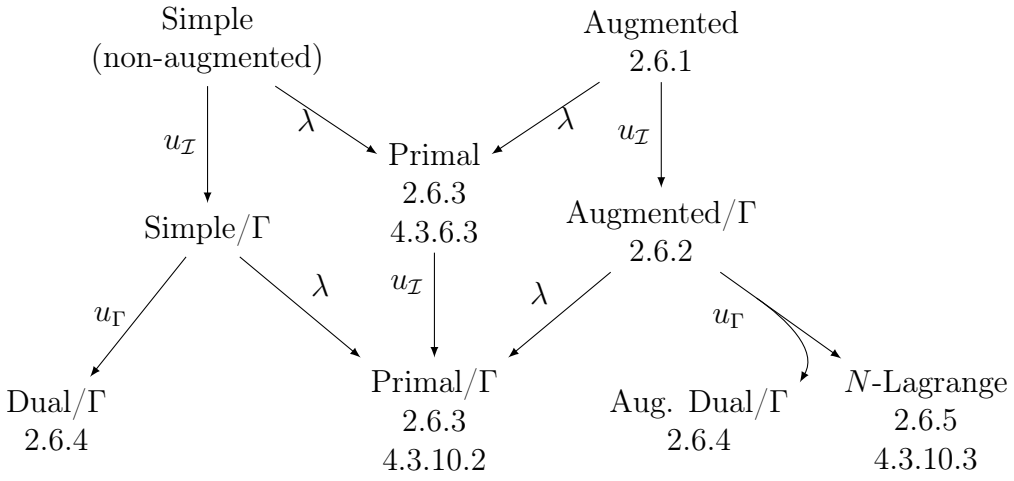


Figure 4.7: The Primal, Primal/Γ and N-Lagrange formulations presented in Figure 2.3 are implemented in `ddmpy`; aS preconditioners (Section 2.7.2) for both primal formulations are presented in Section 4.3.12.

- a *generator function* is a function that (implicitly) returns an iterator. Instead of using the `return` statement to produce one single value, it uses (several) `yield` statements to produce a series of values: each `yield` produces a value and suspends the execution of the function until the next iteration. Our `neighborSendRecv()` function presented in Section 4.3.4 is a generator function.

- a *decorator* is a function that takes a function as input and returns a function. A function `f` can be replaced by the decorated function `g(f)` by adding the line `@g` above the definition of `f`. Many functions in `ddmpy` are timed using the `TimeIt` decorator defined in Section 4.3.5.

- a *context manager* is an object that defines operations to be performed when entering and exiting a particular block of code starting with a `with` statement. The `TimeIt` decorator presented above is also used as a context manager in the code.

- a *special method* is a method that is called implicitly by python to perform a certain operation. For instance, one can overload the `+` operator by defining a `__add__()` method in a class. Then, `a + b` is resolved as `a.__add__(b)`. Several arithmetic operators are overloaded following this approach in the `DistVector` and `DistMatrix` classes in sections 4.3.6.2 and 4.3.6.3. The `TimeIt` decorator mentioned above is in fact not a function but a class with a `__call__()` method, that can therefore be called like a function.

- *duck-typing* is a way of handling type safety: instead of explicitly checking the type of function argument, the presence of some methods and properties used in the function is enough for the argument to be accepted. For instance, the matrix `A` and preconditioner `M` used in the `cg` method (Listing 13) can be any objects with a suitable `__matmul__()` method (special method for the `@` operator).

4.3.3 Dependencies (required and optional)

All the operations in the ddmpy package can be performed using only common python packages available in any scientific distribution: numpy, scipy and mpi4py.

```
from __future__ import print_function
from collections import OrderedDict
import traceback
import time
import numpy as np
import scipy.sparse as ssp
import scipy.linalg as la
import scipy.sparse.linalg as sla
from mpi4py import MPI
```

Listing 14: Required dependencies.

However, better performance can optionally be obtained for some operations by relying on external libraries such as Pastix (Hénon et al., 2002), Mumps (Amestoy et al., 2001) or MaPHyS (Agullo et al., 2016b; Giraud et al., 2008) that are used through python wrappers. The import for these modules are wrapped in `try ... except` blocks so that ddmpy can be imported easily with or without these optional dependencies.

```
try:
    import pypastix
except ImportError:
    traceback.print_exc()

try:
    import pymumps
except ImportError:
    traceback.print_exc()

try:
    import pymaphys
except ImportError:
    traceback.print_exc()
```

Listing 15: Optional dependencies. Without these optional dependencies, ddmpy falls back onto the required dependencies to perform all operations.

4.3.4 An abstraction layer over the MPI for a domain decomposition methods dd

The `neighborSendRecv` generator function. The `ddmpy` module is designed for testing and comparing parallel DDM in a distributed environment. One process per subdomain is created. It handles all local computations relative to the subdomain, and the *Message Passing Interface* (MPI) (The MPI Forum, 1993) is used to handle communications through the `mpi4py` wrapper (Dalcin et al., 2011). `mpi4py` provides both low-level functions that communicate efficiently memory-contiguous buffers, such as `numpy` arrays, and a higher-level interface that makes it easy to communicate any python object, with a performance penalty.

A common communication pattern in DDM is for each subdomain to send messages to all his neighbors and receive the corresponding messages. This is handled in our code by a `neighborSendRecv` *generator function* as defined in Section 4.3.2. A generator function is a function that can return several successive values (through the `yield` keyword) and can be used in a loop. In our case, `neighborSendRecv` takes a dictionnary of messages to send, and returns the received messages in the order they arrive. The arguments of this generator function are:

- `messages`, a dictionnary of messages: `messages[i]` is sent to process `i`. Each message can be any python object.
- `comm`, the MPI communicator to be used for the communication
- `n_recv`, the number of expected incoming messages. This argument is optional if the number of expected messages is equal to the number of sent messages (`n_recv = len(messages)`).
- `is_ndarray`, `True` if the specialized `mpi4py` routines for fast communication of memory contiguous arrays should be used, `False` for using the generic routines in `mpi4py` that can handle any python object. By default, a strategy is chosen based on the type of the objects in `messages`.
- `dtype`, used only if `is_ndarray`: `numpy` datatype of the arrays in the incoming messages.
- `debug`, boolean: print debug messages. Default: `False`.

```
1  def neighborSendRecv(messages, comm=MPI.COMM_WORLD, n_recv=None,
2                       is_ndarray=None, dtype=None, debug=False):
3      if n_recv is None:
4          n_recv = len(messages)
5      if is_ndarray is None:
6          is_ndarray = all(isinstance(v, np.ndarray)
7                           for v in messages.values())
8      if is_ndarray and dtype is None:
9          assert(len(messages) > 0)
10         dtype = list(messages.values()).pop().dtype
11         assert(all(v.dtype == dtype
12                    for v in messages.values()))
```

Listing 16: neighborSendRecv (1/5).

In order not to mix messages from different neighborSendRecv calls, an incremental tag is used to discriminate between different calls to the generator function.

```
    tag = getattr(neighborSendRecv, "tag", 0)
    neighborSendRecv.tag = tag + 1
```

Listing 17: neighborSendRecv (2/5).

Then, all the messages are sent using the optimized non-blocking Isend function for numpy arrays or the generic non-blocking isend function for other python objects.

```
    # Send
    requests = []
    for dest, message in messages.items():
        if debug:
            print("{}.-> {}: {} / {} {}".format(
                comm.Get_rank(), dest, message,
                tag, is_ndarray))
        if is_ndarray:
            req = comm.Isend(message, dest, tag)
        else:
            req = comm.isend(message, dest, tag)
        requests.append(req)
```

Listing 18: neighborSendRecv (3/5).

After this step, the n_recv incoming messages are returned in the order they arrive. For this, a call to Probe waits for an incoming message. This message is then received using the Recv (for arrays) or recv (for objects) function in mpi4py, and a tuple =(source, message) is returned where source is the MPI rank of the source message.

```
    # Probe and Receive
    status = MPI.Status()
    while n_recv > 0:
        comm.Probe(MPI.ANY_SOURCE, tag, status)
        source = status.Get_source()
        if is_ndarray:
            size = status.Get_count(
                MPI._typedict[np.dtype(dtype).char])
            message = np.empty(size, dtype)
            comm.Recv(message, source, tag)
        else:
```

```
12              message = comm.recv(None, source, tag)
13          if debug:
14              print("{} ->.{}: {} / {}{}".format(
15                  source, comm.Get_rank(), message,
16                  tag, is_ndarray))
17          yield source, message
18          n_recv -= 1
```

Listing 19: `neighborSendRecv` (4/5).

After all messages are received, one waits for all the send requests to terminate.

```
1       # Wait for Send to terminate
2       for req in requests:
3           req.wait()
```

Listing 20: `neighborSendRecv` (5/5).

Testing the `neighborSendRecv` function. The previous function can be tested by exporting the following code in a file `test_neighborSendRecv.py`

```
import ddmpy as dd

n = dd.MPI.COMM_WORLD.Get_size()
rank = dd.MPI.COMM_WORLD.Get_rank()

send_messages = {i: "message from {} to {}".format(rank, i)
                 for i in range(n) if i != rank}
SendRecv = dd.neighborSendRecv(send_messages)
recv_messages = {src: message for src, message in SendRecv}

if rank==0:
    print("send:{}\nrecv:{}".format(
        send_messages, recv_messages))
```

and running

```
mpirun -np 3 python test_neighborSendRecv.py
```

```
send:{1: 'message from 0 to 1', 2: 'message from 0 to 2'}
recv:{1: 'message from 1 to 0', 2: 'message from 2 to 0'}
```

Process 0 sends and receives one message from each of the two other processes.

4.3.5 A hierarchical profiler suited for python

The TimeIt context manager and decorator. In order to facilitate debugging and performance analysis, a TimeIt class enables to time the execution of selected functions and blocks of code. The advantage compared to using the standard cProfile module in python is the ability to choose which functions should be profiled and to instrument individual blocks of code inside a function. Since timed functions or code blocks can call other timed functions and code blocks, TimeIt events are created in a hierarchical fashion. Hierarchical traces generated by a distributed execution of a ddmpy DD solver are presented and analyzed in Section 4.4.4.

Each instance of this class contains the information for one instrumented function or block of code. Global information is stored in two class attributes. First, events contains a *history* of the execution: a list of all events that have been created since the reset() function has been called. Second, the stack class attribute contains a list of events that have started but are not yet finished (the *parents* of the current event in the hierarchy). One can set the debug flag to True to have the trace printed on the standard output during execution. One can also limit the size of the events list by setting the max_events class attribute.

```
class TimeIt:

    debug = False
    max_events = None
    events = []
    stack = []
    t0 = time.time()
```

Listing 21: TimeIt (1/5).

The TimeIt class can be used as a *context manager* to time individual blocks of code.

```
    def __init__(self, name=""):
        self.name = name

    def __enter__(self):
        self.begin = time.time() - self.t0
        self.end = np.nan
        self.duration = np.nan
        self.level = len(self.stack)
        self.events.append(self)
        self.stack.append(len(self.events)-1)
        if TimeIt.debug:
            print(self.__str__(which="current"))
        return self

```

```
15      def __exit__(self, type, value, traceback):
16          self.end = time.time() - self.t0
17          self.duration = self.end - self.begin
18          if TimeIt.debug:
19              print(self.__str__(which="current"))
20          if self.stack:
21              self.stack.pop()
22          return False
```

Listing 22: `TimeIt` (2/5).

It can also be used as a decorator. When called on a method of a `LinearOperator` instance (see Section 4.3.7), the time spent in the method is stored in an attribute of the `LinearOperator` instance.

```
1       def __call__(self, f):
2           def timed_f(*args, **kwargs):
3               # We create a default name
4               name = self.name
5               if name=="":
6                   name = type(args[0]).__name__ + " " + f.__name__
7               # We time and run the function
8               # We create a new TimeIt instance because self is created
9               # at function declaration and is the same object for all
10              # executions of the function
11              with TimeIt(name) as t:
12                  res = f(*args, **kwargs)
13              # Store the duration in the object
14              if len(args)>0 and isinstance(args[0], LinearOperator):
15                  solver = args[0]
16                  key = "t_" +  name.replace(" ", "_")
17                  if key in solver.parameters:
18                      solver.parameters[key] += t.duration
19                  else:
20                      solver.parameters[key] = t.duration
21              return res
22          return timed_f
```

Listing 23: `TimeIt` (3/5).

One can print the history by calling `print(TimeIt())`.

```
1       def __str__(self, which="all"):
2           if which=="current":
3               s = "{:50s} | {:2d} | {:12.7f} | {:12.7f} |\
4               {:12.7f}".format( "!  "*self.level + self.name,
```

```
                                self.level, self.begin,
                                self.duration, self.end)
        elif which=="all":
            s = "\n".join(
                [t.__str__(which="current")
                 for t in self.events[:TimeIt.max_events]])
        else: # "stack"
            s = "\n".join(
                [self.events[i].__str__(which="current")
                 for i in self.stack])
        return s
```

Listing 24: `TimeIt` (4/5).

The origin of time can be reset, along with the `events` and `stack` class attributes by calling the `reset` class method. If running in parallel, a MPI `barrier()` call is used to synchronize the origins of all processes.

```
    @classmethod
    def reset(cls, comm=MPI.COMM_WORLD):
        cls.events = []
        cls.stack = []
        if comm is not None:
            comm.barrier()
        cls.t0 = time.time()
```

Listing 25: `TimeIt` (5/5).

Testing the `TimeIt` class. The `TimeIt` class can be tested by executing the following code

```
from ddmpy import TimeIt, np

@TimeIt("my sum")
def my_sum(a, b):
    with TimeIt("compute the sum"):
        c = a + b
    return c

@TimeIt("f")
def f():
    with TimeIt("concatenate strings"):
        s1 = "Hello, world!"
        s2 = "How are you?"
        s = "\n".join((s1, s2))
```

```
15      c = my_sum(1, 2)
16      with TimeIt("sum of squares"):
17          n = sum(i**2 for i in range(10000))
18
19  TimeIt.reset()
20  a = my_sum(42, 0)
21  f()
22  print(TimeIt())
```

```
my sum                  | 0 | 0.0054998 | 0.0000172 | 0.0055170
!  compute the sum      | 1 | 0.0055099 | 0.0000031 | 0.0055130
f                       | 0 | 0.0107698 | 0.0010610 | 0.0118308
!  concatenate strings  | 1 | 0.0107780 | 0.0000041 | 0.0107820
!  my sum               | 1 | 0.0107880 | 0.0000100 | 0.0107980
!  !  compute the sum   | 2 | 0.0107930 | 0.0000019 | 0.0107949
!  sum of squares       | 1 | 0.0108030 | 0.0010269 | 0.0118299
```

Each row in the result represents an event (a TimeIt block or a TimeIt-decorated function call). The five columns of an event are its name, level, starting time, duration and ending time. At the highest level (level 0), two decorated functions are called (lines 20 and 21), resulting in the first and third line in the results. During the execution of these functions, other decorated functions and blocks are executed, resulting in TimeIt events of level 1, which may, recursively, generate other events of higher level.

4.3.6 Distributed DD

In a distributed DD framework as presented in Section 3, each MPI process is responsible for one subdomain and stores the data related to this subdomain. Namely, each unknown is assigned to one or several subdomains, such that, from the matrix adjacency graph point of view the two vertices of each edge are assigned to at least one common subdomain. In the example presented in Figure 4.8, the 7 nodes are assigned to the following subdomains:

Node	0	1	2	3	4	5	6
Subdomain(s)	1, 3	1	1, 2	1, 2, 3	2	2, 3	3

The MPI ranks are numbered from 0 to N - 1 consistently with the default MPI numbering. However, subdomains are numbered from 1 to N to stay consistent with previous chapters where index 0 is reserved for the coarse space. In the code, the subdomain Ω_i is therefore identified by the rank $i-1$ of the MPI process it is associated with. A local numbering of the unknowns is used in each subdomain: in the example, the local indices [0, 1, 2, 3] in Ω_1 correspond to the same global indices [0, 1, 2, 3]. The global indices corresponding to local indices [0, 1, 2, 3] of Ω_2 and Ω_3 are [3, 2, 4, 5] and [5, 6, 0, 3], respectively. Each subdomain has to know which unknowns it shares with which other subdomains (dotted lines in Figure 4.8 (c)). In our example, for instance, Ω_1 shares [2, 3] (in local ordering) with Ω_2 and [0, 3] with Ω_3. Ω_2 shares [1, 0] with Ω_1 and [0, 3] with Ω_3. Ω_3 shares [2, 3] with Ω_1 and [3, 0] with Ω_2.

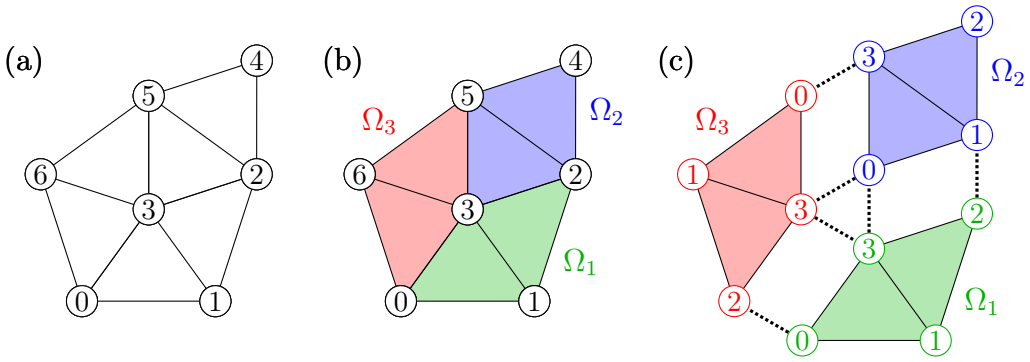


Figure 4.8: The global domain in (a) is partitioned in three subdomains Ω_1, Ω_2 and Ω_3 in (b). Each subdomain can use a local ordering of its unknowns as in (c); duplicate unknowns in different subdomains are linked by a dotted line.

Consistently with the assumption in Chapter 3 that the matrix is provided in a distributed fashion (2. in Definition 1), `ddmpy` does not partition matrices or vectors to distribute them (from 4.8 (a) to 4.8 (c)), but uses distributed data as an input (4.8 (c)).

There is no constraint on the relative position of interior and interface indices in the local ordering. For instance, in Ω_2, the interior node (2 in local ordering) is not at the beginning or the end of the local unknown list. However, the orders of the local orderings of two neighboring subdomains on their common interface depend on each other. For instance, nodes 2 and 3 in global ordering have the same index (2 and 3) in the local ordering of Ω_1, whereas in the local ordering of Ω_2, they are in reverse order (1 and 0). As a result, when communicating data between Ω_1 and Ω_2, one has to take into account the ordered list of nodes in their common interface ([2, 3] for Ω_1 and [1, 0] for Ω_2) instead of the (unordered) set of common unknowns ($\{2, 3\}$ and $\{0, 1\}$) in order not to associate node 2 of Ω_1 with node 0 of Ω_2 instead of node 1, and node 3 of Ω_1 with 1 instead of 0.

This connectivity information and the MPI communicator are handled through a `DomainDecomposition` class, that is presented in Section 4.3.6.1. The `DistVector` and `DistMatrix` classes used to distribute linear algebra operations are then presented in sections 4.3.6.1 and 4.3.6.3.

4.3.6.1 The `DomainDecomposition` class

The `DomainDecomposition` class is presented in 4 steps: after its constructor (`__init__()` method), the code of its `assemble()` method is given. This method is used to add local contributions from a subdomain and its neighbors to a vector or matrix. Then, an `interface_dd()` method used to eliminate interior degrees of freedom is presented, and an example is given.

Class constructor (`__init__()` method). The class's constructor has three arguments:

- `ni`, the number of local unknowns (4 in our example).

- `neighbors`, a dictionnary that maps the MPI rank of neighbor subdomains to a list containing the local indices of shared unknowns with the said subdomain (for Ω_1 of MPI rank 0, `neighbors = {1: [2, 3], 2: [0, 3]}`).
- `comm`, a MPI communicator (defaults to `MPI.COMM_WORLD`).

```
class DomainDecomposition(object):

    def __init__(self, ni, neighbors, comm=MPI.COMM_WORLD):
        self.ni = ni
        self.neighbors = neighbors
        self.interface = np.unique(np.hstack(neighbors.values()))
        self.nG = len(self.interface)
        self.nI = ni - self.nG
        self.comm = comm
        self.n_subdomains = comm.Get_size()
        self.rank = comm.Get_rank()
        # Boolean partition of unity
        # for scalar products
        self.D = np.ones((ni, 1), dtype=np.int)
        for n in neighbors:
            if n < self.rank:
                self.D[neighbors[n], 0] = 0
        # Compute the global ordering
        # Number of unknowns the subdomain is responsible for
        ind_responsible = np.nonzero(self.D)[0]
        n_responsible = len(ind_responsible)
        n_per_subdomain = np.empty(self.n_subdomains, dtype=int)
        comm.Allgather(
            np.array(n_responsible),
            n_per_subdomain)
        # Total number of unknowns
        self.n = n_per_subdomain.sum()
        # Number of unknowns of smaller rank
        offset = n_per_subdomain[:self.rank].sum()
        # global numbering of the subdomain's unknowns
        self.global_indices = np.zeros((self.ni), dtype=int)
        self.global_indices[ind_responsible] = range(
            offset, offset+n_responsible)
        # Send global numbering of shared nodes to neighbors
        send = {n: self.global_indices[neighbors[n]]
                for n in neighbors if self.rank<n}
        n_recv = len([n for n in neighbors if n<self.rank])
        for n, recv in neighborSendRecv(
                send, n_recv=n_recv, is_ndarray=True, dtype=int):
```

```
40            self.global_indices[neighbors[n]] += recv.ravel()
```

Listing 26: `DomainDecomposition` (1/4).

The `assemble()` method. Using the same notations as in Chapter 2, if a vector b is a sum of local components $b = \sum_{j=1}^{N} \mathcal{R}_j^T b_j$, the restriction of b to subdomain Ω_i is $\mathcal{R}_i b = \mathcal{R}_i \sum_{j=1}^{N} \mathcal{R}_j^T b_j = \sum_{j=1}^{N} \mathcal{R}_i \mathcal{R}_j^T b_j$. Since $\mathcal{R}_i \mathcal{R}_j^T = 0$ if $j \neq i$ and j is not a neighbor of i, if each component b_i is only known in the process holding subdomain Ω_i, neighbor-to-neighbor communications are required to send and receive the $\mathcal{R}_i \mathcal{R}_j^T b_j$ components, in a process called *vector assembly*.

Similarly, if a matrix $\mathcal{A}$ is known as a sum of local components $\mathcal{A} = \sum_{j=1}^{N} \mathcal{R}_j^T \mathcal{A}_j \mathcal{R}_j$, computing the local matrix $\mathcal{A}_i^{(AS)} = \mathcal{R}_i \mathcal{A} \mathcal{R}_i^T$ of the AS preconditioner (Section 3.1) also requires a *matrix assembly*: $\mathcal{A}_i^{(AS)} = \mathcal{R}_i \sum_{j=1}^{N} \mathcal{R}_j^T \mathcal{A}_j \mathcal{R}_j \mathcal{R}_i^T = \sum_{j=1}^{N} \mathcal{R}_i \mathcal{R}_j^T \mathcal{A}_j \mathcal{R}_j \mathcal{R}_i^T$. The difference between the *vector* and *matrix assembly* is that in the *matrix assembly*, the restrictions are applied on both sides of the matrix.

When computing a matrix-vector product $y = \mathcal{A}x$ where x and y are known in each subdomain Ω_i through their local restrictions $x_i = \mathcal{R}_i x$ and $y_i = \mathcal{R}_i y$, the product is computed as $y_i = \mathcal{R}_i y = \mathcal{R}_i \mathcal{A} x = \mathcal{R}_i \sum_{j=1}^{N} \mathcal{R}_j^T \mathcal{A}_j \mathcal{R}_j x = \sum_{j=1}^{N} \mathcal{R}_i \mathcal{R}_j^T \mathcal{A}_j x_j$. Each process computes a local matrix-vector product $y'_j = \mathcal{A}_j x_j$. The resulting vector is *assembled* to compute $y_i = \sum_{j=1}^{N} \mathcal{R}_i \mathcal{R}_j^T y'_j$.

The $\mathcal{R}_i$ and $\mathcal{R}_j^T$ matrices represent global-to-local and local-to-global index transformations. However, the global ordering is not used in the *vector* and *matrix assembly* processes: the matrix product $\mathcal{R}_i \mathcal{R}_j^T$ represents a local (in Ω_j)-to-local (in Ω_i) transformation. This transformation is performed implicitly by requiring that the lists in the `neighbors` dictionaries from two neighboring subdomains are in the same order, as stated above. In our example, the interface between subdomains Ω_1 and Ω_2 is [2, 3] in global ordering. In local ordering, these two nodes are [2, 3] in Ω_1 and [1, 0] in Ω_2. Notice that in Ω_2, the nodes are not in ascending order: they are sorted so as to match the ordering of the corresponding nodes in the list in Ω_1.

The `assemble method` has 3 arguments:

- `v`, a vector or matrix to assemble,
- `dim`, 1 for *vector assembly*, 2 for *matrix assembly*,
- `debug`, a boolean flag passed to the `neighborSendRecv` function.

```
1     def assemble(self, v, dim=1, debug=False):
2         comm = self.comm
3         if dim==1:    # vector assembly
4             indices = self.neighbors
5         elif dim==2: # matrix assembly
6             indices = {n: np.ix_(i,i) if len(i)>1 else (i,i)
7                        for n, i in self.neighbors.items()}
```

```
        else:
            raise ValueError("dim should be 1 or 2")
        v_ = v.copy()
        send = {j: v[indices[j]] for j in indices}
        for j, recv in neighborSendRecv(send, debug=debug):
            if recv.shape != send[j].shape:
                recv = recv.reshape(send[j].shape)
            v_[indices[j]] += recv
        return v_
```

Listing 27: `DomainDecomposition` (2/4).

The `interface_dd()` method. Some DDM use an interface formulation where interior unknowns that belong to only one subdomain are eliminated. This elimination processes produces a Schur complement system on the interface unknowns. From a

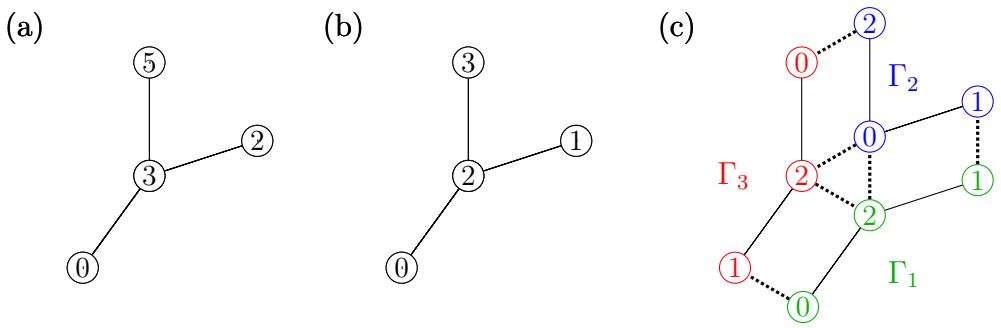


Figure 4.9: The interior unknowns can be eliminated from the domain presented in Figure 4.8 to keep only interface unknown that are shared by two subdomains or more (a). A global interface numbering is introduced (b). The global interface can be expressed as the union of the three local interfaces Γ_1, Γ_2 and Γ_3 in (c). An interface Domain Decomposition is a particular case of Domain Decomposition where all unknowns are assigned to more than one subdomain.

`DomainDecomposition` object, one can use the `interface_dd` method to compute a local ordering of the interface.

```
    def interface_dd(self):
        """return the DomainDecomposition of the interface matrix"""
        neighbors = {n: np.searchsorted(self.interface, i).tolist()
                     for n, i in self.neighbors.items()}
        return DomainDecomposition(len(self.interface), neighbors,
                                   self.comm)
```

Listing 28: `DomainDecomposition` (3/4).

Testing the `DomainDecomposition` class. A `DomainDecomposition` object `dd` representing the domain in Figure 4.8 is available by using the `test()` static method of the `DomainDecomposition` class

```
    @staticmethod
    def test():
        # see Figure 4.8
        comm = MPI.COMM_WORLD
        rank = comm.Get_rank()
        assert(comm.Get_size() == 3)
        if rank==0:   # Ω1
            neighbors = {1: [2, 3], # interface with Ω2
                         2: [0, 3]} # interface with Ω3
            ni = 4                  # number of local unknowns
        elif rank==1: # Ω2
            neighbors = {0: [1, 0], # interface with Ω1
                         2: [0, 3]} # interface with Ω3
            ni = 4                  # number of local unknowns
        elif rank==2: # Ω3
            neighbors = {0: [2, 3], # interface with Ω1
                         1: [3, 0]} # interface with Ω2
            ni = 4                  # number of local unknowns
        dd = DomainDecomposition(ni, neighbors)
        return dd
```

Listing 29: `DomainDecomposition` (4/4).

The corresponding interface `DomainDecomposition` object `ddG` representing the interface in Figure 4.9 is available using the `interface_dd()` method. For both domain decompositions and for each of the three subdomains, the global indices corresponding to the local unknown are given. Then, a boolean partition of unity `D` is presented: the value of `D` for an unknown is 1 if this unknown is in the interior of the current domain or if the current domain is of smallest rank among the subdomains that the unknown belongs to. If an unknown is shared with a subdomain of smaller rank, the value of `D` for the index corresponding to this unknown is 0. Assembling this partition of unity, one sums locally the contributions of all neighboring subdomains. Since all but one subdomain contribute 0 on each unknown and the last subdomain (of smallest rank) contributes 1, the resulting value is 1 (`assemble(D)`).

The effect of the assembly process is also illustrated through the computation of a `multiplicity` vector: a vector `x = np.ones(4) = [1, 1, 1, 1]` is assembled as `multiplicity = assemble(x)`. The result obtained by adding the contributions (1) coming from all subdomains, is the number of subdomain that each node is assigned to: an interior node (1, 4 and 6 in global ordering) has a multiplicity of 1, an interface node between two subdomains has a multiplicity of 2 (0, 2 and 5) and a crosspoint between 3 subdomains has a multiplicity of 3 (node 3 in global ordering). In the interface

`DomainDecomposition` object, all interior nodes of multiplicity 1 are eliminated and all remaining nodes have a multiplicity of 2 or more.

```
from ddmpy import *
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

dd = DomainDecomposition.test() # Figure 4.8
ddG = dd.interface_dd()         # Figure 4.9

for dd_ in [dd, ddG]:
    multiplicity = dd_.assemble(np.ones(dd_.ni, int))
    D_assembled = dd_.assemble(dd_.D)
    for i in range(3):
        comm.barrier()
        if rank == i:
            if rank == 0:
                print("dd\n" if dd_ is dd else "ddG\n")
            print("rank: {}\n"
                  "global indices: {}\n"
                  "D:              {}\n"
                  "assemble(D):    {}\n"
                  "multiplicity:   {}\n"
                  .format(rank,
                          dd_.global_indices,
                          dd_.D[:, 0],
                          D_assembled[:, 0],
                          multiplicity))
```

```
mpirun -np 3 python testDD.py
```

```
dd

rank: 0
global indices: [0 1 2 3]
D:              [1 1 1 1]
assemble(D):    [1 1 1 1]
multiplicity:   [2 1 2 3]

rank: 1
global indices: [3 2 4 5]
D:              [0 0 1 1]
assemble(D):    [1 1 1 1]
```

```
multiplicity:   [3 2 1 2]

rank: 2
global indices: [5 6 0 3]
D:              [0 1 0 0]
assemble(D):    [1 1 1 1]
multiplicity:   [2 1 2 3]

ddG

rank: 0
global indices: [0 1 2]
D:              [1 1 1]
assemble(D):    [1 1 1]
multiplicity:   [2 2 3]

rank: 1
global indices: [2 1 3]
D:              [0 0 1]
assemble(D):    [1 1 1]
multiplicity:   [3 2 2]

rank: 2
global indices: [3 0 2]
D:              [0 0 0]
assemble(D):    [1 1 1]
multiplicity:   [2 2 3]
```

4.3.6.2 The `DistVector` class

A global vector b in `ddmpy` can be stored through its local component $b_i = \mathcal{R}_i b$ in the subdomain Ω_i handled by the current MPI process. The values of the local components b_i and b_j on a node shared between Ω_i and Ω_j are the same: the distributed vector is said to be compatible on the interface. Some DDM that use for instance the N-Lagrange formulation (see Section 2.6.5) use distributed vectors that are not compatible: both sides of an interface can hold different values.

Class constructor. The `DistVector` class is used to handle both compatible and not compatible distributed vectors. Its constructor has 4 arguments:

- `bi`, the value of the vector on the nodes in Ω_i, in local ordering,
- `dd`, a `DomainDecomposition` object such that `bi.shape[0] = dd.ni`,
- `compatible`, a boolean used to define the compatibility of the vector on the interface. If the vector is compatible, interface components are stored redundantly on all the

subdomains that share an interface and a partition of unity (see Section 2.7.2) has to be used to compute the dot product in order not to count interface nodes several times,

- assemble, a boolean used to specify if the vector should be assembled during the constructor.

In practice, only the three following combinations of `compatible` and `assemble` are used:

- `compatible=True` and `assemble=False` when the local components of a compatible distributed vector are known. For instance, an initial guess whose value is `xi` can be created as `DistVector(xi, dd, compatible=True, assemble=False)`, or `DistVector(xi, dd)` using the arguments default values.

- `compatible=True` and `assemble=True` when the vector should be computed as the sum of known local contributions. For instance, when the linear system is the FEM discretization of a PDE, the right-hand side vector is known as $b = \sum_{j=1}^{N} \mathcal{R}_j^T b_j$. The local component of b is $\mathcal{R}_i b = \sum_{j=1}^{N} \mathcal{R}_i \mathcal{R}_j^T b_j$ and can be obtained by assembling the vector (see Listing 27). The corresponding `DistVector` object can be created as `DistVector(bi, dd, compatible=True, assemble=True)` or `DistVector(bi, dd, assemble=True)`.

- `compatible=False` and `assemble=False` when the local components of a vector are not expected to hold the same value on shared interface unknowns. This is the case for instance using an augmented or N-Lagrange formulation (see sections 2.6.1 and 2.6.5). A corresponding vector can be created as `DistVector(lambda_i, dd, compatible=False, assemble=False)` or, more simply, `DistVector(lambda_i, dd, compatible=False)`.

A DistVector object has `shape`, `ndim`, `__len__()`, `copy()` and `T` attributes and methods that match those of `numpy` arrays.

```
1  class DistVector(object):
2
3      def __init__(self, bi, dd, compatible=True,
4                   assemble=False):
5          if len(bi.shape) != 2:
6              bi = bi.reshape((dd.ni, -1))
7          if assemble:
8              self.local = dd.assemble(bi)
9          else:
10             self.local = bi
11         self.dd = dd
12         self.shape = dd.n, bi.shape[1]
13         self.ndim = len(self.shape)
```

```
14          self.transposed = False
15          self.compatible = compatible
16
17      def __len__(self):
18          return self.shape[0]
19
20      def copy(self):
21          return DistVector(self.local.copy(), self.dd,
22                            self.compatible)
23
24      @property
25      def T(self):
26          other = DistVector(self.local, self.dd, self.compatible)
27          other.transposed = not self.transposed
28          other.shape = self.shape[::-1]
29          return other
```

Listing 30: `DistVector` (1/3).

Arithmetic operations. Some arithmetic operations on `DistVector` are defined using the *special methods* `__add__()`, `__sub__()`, `__rmul__()`, `__mul__()`, `__div__()`, `__truediv__()` and `__matmul__()` that are used to implement the `+`, `-`, `*`, `/` and `@` operations.

```
1       @TimeIt()
2       def __add__(self, other): # self + other
3           with TimeIt("Local __add__"):
4               return DistVector(self.local + other.local,
5                                 self.dd, self.compatible)
6
7       @TimeIt()
8       def __sub__(self, other): # self - other
9           with TimeIt("Local __sub__"):
10              return DistVector(self.local - other.local,
11                                self.dd, self.compatible)
12
13      @TimeIt()
14      def __rmul__(self, scalar): # scalar * self
15          with TimeIt("Local __rmul__"):
16              return DistVector(scalar * self.local,
17                                self.dd, self.compatible)
18
19      @TimeIt()
20      def __mul__(self, scalar): # self * scalar
```

```
21          with TimeIt("Local __mul__"):
22              return DistVector(self.local * scalar,
23                                self.dd, self.compatible)
24
25      @TimeIt()
26      def __div__(self, scalar): # self / scalar
27          with TimeIt("Local __div__"):
28              return DistVector(self.local / scalar,
29                                self.dd, self.compatible)
30
31      def __truediv__(self, scalar): # self / scalar
32          return self.__div__(scalar)
33
34      @TimeIt()
35      def dot(self, other):
36          dd = self.dd
37          if self.transposed: # self.T @ other
38              # multiplication along the distributed dimension
39              assert(self.dd==other.dd)
40              if not isinstance(other, DistVector):
41                  return NotImplemented
42              with TimeIt("Local dot"):
43                  if self.compatible:
44                      # don't count redundant entries twice
45                      local_dot = self.local.T.dot(
46                          self.dd.D*other.local)
47                  else:
48                      local_dot = self.local.T.dot(other.local)
49              with TimeIt("MPI reduce"):
50                  global_dot = dd.comm.allreduce(local_dot)
51              return  global_dot
52          else: # self @ other
53              # multiplication along the global dimension
54              return DistVector(self.local.dot(other), dd,
55                                self.compatible)
56
57      def __matmul__(self, other): # self @ other
58          return self.dot(other)
```

Listing 31: `DistVector` (2/3).

The `dot` method between a transposed vector and a non-transposed vector computes a scalar product: the scalar products of local components are first computed, and the results are summed, using a MPI reduction.

A `DistVector` instance can be centralized on either one or all the MPI processes by calling its `centralize()` method, which has two arguments:

- `root`, the MPI rank of the process that will eventually hold the global vector. If `root` is `None`, all processes eventually hold the global vector.
- `loc2glob`, a local-to-global mapping to be used for computing the global ordering. If `None`, a default global ordering is used.

```
    def centralize(self, root=None, loc2glob=None):
        if loc2glob is None:
            loc2glob = self.dd.global_indices
        b_global = np.zeros((self.dd.n, self.shape[1]), self.local.dtype)
        a = self.dd.D * self.local
        b_global[loc2glob] = a
        if root is None:
            self.dd.comm.Allreduce(b_global.copy(), b_global)
        else:
            self.dd.comm.Reduce(b_global.copy(), b_global, root=root)
        return b_global
```

Listing 32: `DistVector` (3/3).

Testing the `DistVector` class. Two vectors `x` and `y` are defined with their local index as an input (see Figure 4.8). The vector `x` is defined as a non-compatible vector, and subdomains may have distinct values from each other. For instance, the central node of global index 3 is numbered 3 in Ω_1 and Ω_3 and 0 in Ω_2. The vector y is a compatible vector obtained by assembling (summing) these local indices. Its value on the same central node is therefore $3 + 0 + 3 = 6$. Operations on `DistVector` objects are performed so as to compute `rmul = 2*y`, `add = y + rmul` and `sub = rmul - y` (hence `add = 3*y` and `sub = y`).

```
from ddmpy import *

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

dd = DomainDecomposition.test()
x = DistVector(np.arange(4), dd, compatible=False)
y = DistVector(np.arange(4), dd, assemble=True)
rmul = 2*y
add = y + rmul
sub = rmul - y

for i in range(3):
    comm.barrier()
```

```
15      if rank == i:
16          print(("rank: {}\n"
17                 "global_indices:{}\n"
18                 "x:             {}\n"
19                 "y:             {}\n"
20                 "rmul:          {}\n"
21                 "add:           {}\n"
22                 "sub:           {}\n")
23          .format(rank, dd.global_indices,
24                  x.local[:, 0], y.local[:, 0],
25                  rmul.local[:, 0], add.local[:, 0],
26                  sub.local[:, 0]))
27
28
29  xx = x.T.dot(x)
30  yy = y.T.dot(y)
31  y_glob = y.centralize(root=0)
32  yy_g = y_glob.T.dot(y_glob)
33  if rank==0:
34      print("x.T @ x:           {}\n"
35            "y.T @ y:           {}\n"
36            "y_glob:            {}\n"
37            "y_glob.T @ y_glob: {}"
38            .format(xx, yy, y_glob[:, 0], yy_g))
```

```
1  mpirun -np 3 python testDistVector.py
```

```
rank: 0
global_indices:[0 1 2 3]
x:             [0 1 2 3]
y:             [2 1 3 6]
rmul:          [ 4  2  6 12]
add:           [ 6  3  9 18]
sub:           [2 1 3 6]

rank: 1
global_indices:[3 2 4 5]
x:             [0 1 2 3]
y:             [6 3 2 3]
rmul:          [12  6  4  6]
add:           [18  9  6  9]
sub:           [6 3 2 3]

rank: 2
```

```
global_indices:[5 6 0 3]
x:             [0 1 2 3]
y:             [3 1 2 6]
rmul:          [ 6  2  4 12]
add:           [ 9  3  6 18]
sub:           [3 1 2 6]

x.T @ x:           [[42]]
y.T @ y:           [[64]]
y_glob:            [2 1 3 6 2 3 1]
y_glob.T @ y_glob: [[64]]
```

4.3.6.3 The `DistMatrix` class

Class constructor. The `DistMatrix` class is used to store a matrix $\mathcal{A}$ stored as a sum of local components $\mathcal{A} = \sum_{i=1}^{N} \mathcal{R}_i^T \mathcal{A}_i \mathcal{R}_i$. The constructor takes two parameters:

- `Ai`, the local matrix, can be any object with a `dot` method (using the *duck-typing* mechanism presented in Section 4.3.2); for instance, `Ai` can be a dense matrix (`numpy.ndarray`) or a sparse matrix (`scipy.sparse.spmatrix`).
- `dd`, a `DomainDecomposition` object such that `Ai.shape = (dd.ni, dd.ni)`.

```
1 class DistMatrix(object):
2
3     def __init__(self, Ai, dd):
4         self.local = Ai
5         self.dd = dd
6         self.shape = dd.n, dd.n
7
8     def copy(self):
9         return DistMatrix(self.local.copy(), self.dd)
```

Listing 33: `DistMatrix` (1/4).

Distributed Matrix-vector product. The matrix vector product $u = \mathcal{A}b$ is implemented in two steps: first, $u_i' = \mathcal{A}_i b_i$ ($b_i = \mathcal{R}_i b$) is computed (line 6 of Listing 34). Then, an assembly operation gives $u_i = \sum_{j=1}^{N} \mathcal{R}_i \mathcal{R}_j^T u_j' = \mathcal{R}_i \sum_{j=1}^{N} \mathcal{R}_j^T \mathcal{A}_j \mathcal{R}_j b = \mathcal{R}_i \mathcal{A} b$ (line 7).

```
1     @TimeIt()
2     def dot(self, b, transpose=False):
3         assert(isinstance(b, DistVector))
4         assert(self.dd==b.dd)
```

```
        with TimeIt("Local dot"):
            if transpose:
                Abi = self.local.T.dot(b.local)             # A_i^T b_i
            else:
                Abi = self.local.dot(b.local)               # A_i b_i
        return DistVector(Abi, self.dd, assemble=True) # R_i A b

    def __matmul__(self, b):
        return self.dot(b)

    def __rmatmul__(self, b):
        return self.dot(b.T, transpose=True).T
```

Listing 34: `DistMatrix` (2/4).

A `DistMatrix` object can be centralized on one or all MPI processes by calling the `centralize` method, which has two arguments:

- `root`, the MPI rank of the process which will hold the global matrix. If `root` is `None`, all processes hold the global matrix.
- `loc2glob`, a local-to-global mapping to be used for computing the global ordering. If `None`, a default global ordering is used.

```
    def centralize(self, root=None, loc2glob=None):
        """return the global matrix on process root.
        If no root is provided, then all get the global matrix
        """
        if loc2glob is None:
            loc2glob = self.dd.global_indices
        A_global_i = ssp.lil_matrix((self.dd.n, self.dd.n))
        A_global_i[np.ix_(loc2glob, loc2glob)] = self.local
        if root is None:
            return self.dd.comm.allreduce(A_global_i)
        else:
            return self.dd.comm.reduce(A_global_i, root=root)
```

Listing 35: `DistMatrix` (3/4).

Testing the `DistMatrix` class. A `test()` static method in the `DistMatrix` class provides a SPD distributed matrix $\mathcal{K} = \sum_{i=1}^{3} \mathcal{R}_i^T \mathcal{K}_i \mathcal{R}_i$ such that $\mathcal{K}_i$ is the laplacian matrix of the graph of the local domain plus the identity matrix.

```
    @staticmethod
    def test():
```

```
        dd = DomainDecomposition.test()
        Ki = np.array(
            [[ 3, -1,  0, -1],
             [-1,  4, -1, -1],
             [ 0, -1,  3, -1],
             [-1, -1, -1,  4]], dtype=float)
        K = DistMatrix(Ki, dd)
        return K
```

Listing 36: `DistMatrix` (4/4).

The cg method in ddmpy (see sections 4.2.3 and 4.3.9.1) can handle distributed matrices and vectors and gives the same solution as the `scipy.sparse.linalg.cg` method applied on centralized matrices.

```
from ddmpy import *

K = DistMatrix.test()
dd = K.dd
b = K @ DistVector(np.ones((4, 1)), dd)

x, i = cg(K, b) # ddmpy distributed CG from Listing 46
if dd.rank == 0:
    print("Distributed CG (ddmpy)")
for i in range(3):
    dd.comm.barrier()
    if dd.rank == i:
        print(("rank:  {}\n"
               "index: {}\n"
               "x:     {}\n")
              .format(dd.rank, dd.global_indices,
                      x.local[:, 0]))

b_ = b.centralize()
K_ = K.centralize()
if dd.rank == 0:
    x_, i = sla.cg(K_, b_) # scipy sequential CG
    print(("Global CG (scipy) on "
           "centralized matrix and right-hand side\n"
           "index: {}\n"
           "x:     {}")
          .format(range(7), x_))
```

```
mpirun -np 3 python3 testDistMatrix.py
```

```
Distributed CG (ddmpy)
rank:  0
index: [0 1 2 3]
x:     [ 1.  1.  1.  1.]

rank:  1
index: [3 2 4 5]
x:     [ 1.  1.  1.  1.]

rank:  2
index: [5 6 0 3]
x:     [ 1.  1.  1.  1.]

Global CG (scipy) on centralized matrix and RHS
index: range(0, 7)
x:     [ 1.  1.  1.  1.  1.  1.  1.]
```

4.3.7 The Linear Operator interface

The `ddmpy` module is built upon the notion of *solver* as the central concept (see Section 1.1.4): a DD *solver* is itself built by combining several other *solvers*, for instance to compute a Schur complement or solve the local problem of an aS preconditioner (see sections 2.7.2 and 3.2). In `ddmpy`, several solvers are available as well as various ways to combine them to define new DD solvers. In this section, we focus on the choices made regarding their interface, *i.e.*, the way they can be used regardless of their particular implementation. All the solvers in `ddmpy` share the same interface so that one solver can seamlessly be replaced with another one in some portion of the code.

The most straightforward way to implement the operation of finding x such that $\mathcal{A}x = b$ would be to define a `solve` function

```
1 x = solve(A, b, parameters)
```

There are two limitations attached to this functional paradigm:

- subsequent solves with the same matrix `A` and multiple right-hand sides (rhs) `b1`, `b2`, `...` require the matrix and all parameters to be submitted again for each rhs: it is not possible to reuse some information from previous solves (such as a factorization of `A`).
- the linearity of the solve operation with respect to the rhs does not appear in this formulation. Using the operation `x` $\rightarrow$ `solve(A, x)` as a preconditioner for an iterative method (*e.g.*, using iterative refinement for this operation) requires to define a new function for this operation. Furthermore, implementing a preconditioned iterative method requires to accept a matrix or a function as a preconditioner.

Our approach is to write all solvers and preconditioners as objects with a matrix-like interface:

```
S = LinearOperator(A, parameters)
x = S @ b # or S.dot(b) in python2
```

or

```
S = LinearOperator(parameters)
S.setup(A, updated_parameters)
x = S @ b # or S.dot(b) in python2
```

In ddmpy, solving a distributed linear problem using, for instance, a substructuring approach in which the interior variables are eliminated using Pastix and the interface variables are solved using a Preconditioned CG (PCG) with an AS preconditioner that uses Mumps as an internal solver can be written as:

```
S = DistSchurSolver(A,
                    interface = [0, 1, 2, 3],
                    local_solver = Pastix,
                    interface_solver = ConjGrad(
                        M=AdditiveSchwarz(
                            local_solver=Mumps)))
x = S @ b # or S.dot(b) in python2
```

This implements the AS/$\mathcal{S}$ method Carvalho et al. (2001b) used in Section 3.5.5 and Chapter 5.

4.3.7.1 The abstract class LinearOperator

All solvers in ddmpy inherit from the LinearOperator base class. An instance of a child of the LinearOperator class should have:

- a setup() method.
- a solve() method.
- optionally, a defaults class attribute (dictionnary).

Class constructor. The constructor of the LinearOperator class mainly stores the parameters given as keyword arguments in a parameters instance attribute. Parameters that are not present in the constructor arguments are taken from the defaults dictionnary.

If a matrix A is given in the constructor, the setup() method is called.

```
class LinearOperator(object):

```

```
3       defaults = {}
4
5       def __init__(self, A=None, **kwargs):
6           """ Constructor of the solver
7
8           Store the keyword arguments as parameters for the solver,
9           performs all the analysis steps that are possible without
10          having the matrix A
11          and performs the setup if the matrix A is provided
12          """
13          if hasattr(self, "defaults"):
14              self.parameters = OrderedDict(
15                  sorted(self.defaults.items(),
16                         key=lambda x:str.lower(x[0])))
17              self.parameters.update(kwargs)
18          else:
19              self.parameters = OrderedDict(kwargs.items())
20          self.setup_performed = False
21          if A is not None:
22              self.setup(A)
```

Listing 37: `LinearOperator` (1/5).

The `setup()` method. The `setup()` method of the `LinearOperator` abstract class merely stores a pointer to the matrix `A` and updates the `parameters` dictionary using any keyword arguments given to the function. If a parameter is a class, it is replaced by an instance of this class. This enables to write `SchurSolver(local_solver=Pastix)` instead of `SchurSolver(local_solver=Pastix())`.

Some solvers perform some operations on the matrix that do not depend on the value of the right-hand side. For instance, a direct solver can factorize the matrix without any knowledge of the right-hand side. These steps should be performed in the `setup()` method of the subclass.

```
1       # abstract method
2       def setup(self, A, **kwargs):
3           """ Setups the solver to work on a particular matrix A
4
5           Store the keyword arguments as parameters for the solver
6           and performs all computational steps that are possible
7           without having the RHS.
8           """
9           self.A = A
10          self.shape = A.shape
11          self.parameters.update(kwargs)
```

```
12          # some default parameters are solver types instead of
13          # instance and a type v has to be replaced by an
14          # instance of v
15          for k, v in self.parameters.items():
16              if type(v)==type:
17                  instance = v.__new__(v)
18                  instance.__init__()
19                  self.parameters[k] = instance
20          self.setup_performed = True
```

Listing 38: `LinearOperator` (2/5).

The `solve()` method. The steps that directly depend on the rhs should be implemented in the `solve()` method. For compatibility with `numpy` and the `@` operator, this method is also accessible through a `dot()` method and a `__matmul__()` method.

```
1       # abstract method
2       def solve(self, b):
3           """Performs a solve with b as a right-hand side vector.
4           A subclass may allow multiple rhs to be specified as a
5           matrix.
6           """
7           pass
8
9       def dot(self, b):
10          if self.setup_performed:
11              return self.solve(b)
12          else:
13              raise RuntimeError("The operator was not setup before"
14                                 " performing the solve")
15
16      def __matmul__(self, b): # self @ b
17          return self.dot(b)
18
19      def matvec(self, b): # scipy.sparse.linalg.aslinearoperator()
20          return self.dot(b)
```

Listing 39: `LinearOperator` (3/5).

A string representation of a `LinearOperator` `s` is accessible through its `__str__()` special method. `print(s)` prints the name of the class of the solver, and the value of all its parameters. Parameters that are themselves `LinearOperator` instances are recursively converted to a string representation.

```
1       def __str__(self, level=0):
2           s = [self.__class__.__name__]
```

```
        for key, value in self.parameters.items():
            k = str(key)
            if isinstance(value, LinearOperator):
                v = value.__str__(level+1)
            else:
                v = str(value)
            s.append("  "*(1+level) + k + " : " + v)
        return "\n".join(s)
```

Listing 40: `LinearOperator` (4/5).

For easier use, a parameter `p` of a `LinearOperator` instance `s` is available as `s.p` in addition to the classical `s.parameters[p]`.

```
    def __getattr__(self, item):
        if item != "parameters" and item in self.parameters:
            return self.parameters[item]
        elif item in self.__dict__:
            return self.__dict__[item]
        else:
            raise AttributeError("No {} in {}."
                                 .format(item, self))
```

Listing 41: `LinearOperator` (5/5).

4.3.7.2 Testing the `LinearOperator` class

To test the `LinearOperator` class, a very simple `ScalarMultiply` subclass is defined by:

```
from ddmpy import *

class ScalarMultiply(LinearOperator):
    defaults = {"dummy_parameter": None,
                "alpha": 0}
    @TimeIt()
    def setup(self, A, **kwargs):
        LinearOperator.setup(self, A, **kwargs)
    @TimeIt()
    def solve(self, b):
        return self.alpha*b
```

This `ScalarMultiply` operator, which is only given as a very simple example of subclass of `LinearOperator`, takes a vector b in argument (`b`) and returns $x = \alpha b$, where α is a parameter (`alpha`) with a default value $\alpha = 0$.

Then, a matrix `A` (whose values have thus no mathematical effect in the particular case of this dummy operator) and a vector `b` are created by

```
A = np.arange(9).reshape((3, 3))
b = np.ones((3, 1))
```

eventually an instance is created with default parameters

```
s = ScalarMultiply()
print(s)
```

```
ScalarMultiply
  alpha : 0
  dummy_parameter : None
```

Since no matrix has been provided yet to `s`, its `dot()` method cannot be properly invoked

```
s.dot(b)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ddmpy.py", line 446, in dot
    raise RuntimeError("The operator was not setup before"
RuntimeError: The operator was not setup before performing the solve
```

However, one can call the `setup()` method to provide a matrix (although its values have no effects in the particular case of this dummy solver) before calling the `dot()` method

```
s.setup(A)
print(s.dot(b))
```

```
[[ 0.]
 [ 0.]
 [ 0.]]
```

A vector $x = \alpha b$ zero vector is obtained since the $\alpha = 0$ default parameter was used. It is possible to override this default value in the setup method to instead compute $x = 42b$

```
s.setup(A, alpha=42)
print(s.dot(b))
```

```
[[ 42.]
 [ 42.]
 [ 42.]]
```

Thanks to the `@TimeIt` decorators above the method definitions in the class, the time spent in the methods is automatically added to the instance parameters

```
print(s)
```

```
ScalarMultiply
  alpha : 42
  dummy_parameter : None
  t_ScalarMultiply_setup : 0.000132322311401
  t_ScalarMultiply_solve : 7.39097595215e-05
```

It is also possible to supply the matrix and parameters at initialization

```
s = ScalarMultiply(A, dummy_parameter="toto")
print(s)
```

```
ScalarMultiply
  alpha : 0
  dummy_parameter : toto
  t_ScalarMultiply_setup : 1.81198120117e-05
```

4.3.8 Direct linear solvers

Direct solvers use a variation of Gaussian elimination (Section 1.1.4) to compute a factorization of $\mathcal{A}$, *i.e.*, a decomposition as a product of matrices. Then, the factorization is used to solve the linear problem. Usually, the factorization step (`setup()` in `ddmpy`) is far most expensive than the solve step (`solve()`).

4.3.8.1 Factorizing matrices using Scipy

A simple example of a `LinearOperator` subclass is the `ScipyDirectSolver` class, which solves the linear problem $\mathcal{A}x = b$ using direct solvers available in `scipy` (Anderson et al., 1999; Davis, 2008; Demmel et al., 1999). $\mathcal{A}$ should be a simple (not distributed) dense or sparse matrix (`numpy.ndarray` or `scipy.sparse.spmatrix`). A suitable solver in `scipy` is chosen to factorize the matrix, depending on the type of `A` and the `symmetry` parameter given to the `ScipyDirectSolver` constructor or `setup()` method.

```
class ScipyDirectSolver(LinearOperator):
    """Factorize a matrix using scipy

    Choose an appropriate Scipy solver according to the
    format (dense/sparse) and symmetry of A. Optional
    parameters are:
    - symmetry=True/False. Default: False
```

```
    """

    defaults = {"symmetry": False}

    @TimeIt()
    def setup(self, A, **kwargs):
        LinearOperator.setup(self, A, **kwargs)
        if ssp.issparse(A):
            self.solve_ = sla.factorized(A)
        else:
            if self.symmetry in {2, "SPD"}:
                self.LLt = la.cho_factor(A)
                self.solve_ = lambda b:la.cho_solve(
                    self.LLt, b)
            else:
                self.LU = la.lu_factor(A)
                self.solve_ = lambda b: la.lu_solve(
                    self.LU, b)

    @TimeIt()
    def solve(self, b):
        return self.solve_(b)
```

Listing 42: `ScipyDirectSolver`.

4.3.8.2 Computing a pseudoinverse

The `ScipyDirectSolver` class raises an exception when trying to factorize a singular matrix; in that case, one should use a `Pinv` solver that computes a dense pseudoinverse of the matrix using the `scipy.linalg.pinv()` function.

```
class Pinv(LinearOperator):

    @TimeIt()
    def setup(self, A, **kwargs):
        LinearOperator.setup(self, A)
        if ssp.issparse(A):
            self.A_pinv = la.pinv(A.A)
        else:
            self.A_pinv = la.pinv(A)

    @TimeIt()
    def solve(self, b):
        return self.A_pinv.dot(b)
```

Listing 43: `Pinv` (pseudoinverse).

Computing a dense pseudoinverse should be avoided as it takes a lot more time and memory than factorizing the matrix, especially when it is large and sparse. In that case, one should use one of the `Pastix` and `Mumps` classes presented below.

4.3.8.3 The Mumps sparse direct solver

A good alternative to the `ScipyDirectSolver` and `Pinv` classes for direct solves (sections 4.3.8.1 and 4.3.8.2) for factorizing sparse matrices is the `Mumps` solver (Amestoy et al., 2001). For the purpose of this thesis, we have developed an extensive `python` support for the `Mumps` solver. This wrapper is available on `https://gitlab.inria.fr/gmarait/PYMUMPS`. It can factorize `scipy.sparse.spmatrix` matrices in a sequential or distributed (in global ordering) fashion, as well as `DistMatrix` matrices in local ordering. Note that as explained in Section 4.3.3, the `Mumps` solver is optional for `ddmpy`.

The `ddmpy` toolbox needs only a subset of this pymumps extensive `Mumps` support. We therefore do not further document it in this thesis, and we only report the `ddmpy Mumps` class that is built on top of `pymumps`, exposing the four following parameters:

- a `verbose` boolean.
- a `symmetry` boolean or integer. False/0 means not symmetric, True/1 means symmetric and 2 means SPD. (Note that the native Fortran Mumps API uses different values for the `id.sym` variable).
- a `comm` MPI communicator if the matrix is distributed in global ordering.
- a `ordering` string that can be "auto", "Scotch" or "Metis" to choose the graph partitioner used by the solver.

```
class Mumps(LinearOperator):
    """Factorize a matrix using mumps

    Three modes:
    - sequential: comm is None and A is a Scipy Sparse Matrix
    - distributed in global ordering: comm is not None
      and A is a Scipy Sparse Matrix
    - distributed in local ordering: A is a DistMatrix
      the communicator used is A.dd.comm, comm is not used


    Optional parameters are:
    - verbose=True/False. Default: False
    - symmetry=0 (General), 1 (Symmetric), 2 (SPD). Default: 0.
      If symmetry>0, only the lower triangular part of A is used.
```

```
    - comm, optional. Default: None
    """

    defaults = {"verbose": False,
                "symmetry": False,
                "comm"   : None,
                "ordering": "auto"}

    @TimeIt()
    def setup(self, A, **kwargs):
        LinearOperator.setup(self, A, **kwargs)
        # We split the function in two part to be able
        # to easily add the Schur code in between
        self.init()
        self.factorize()

    def init(self):
        self.driver = pymumps.Mumps('D')
        id = self.driver.id
        A = self.A

        # Set id.par, id.comm and id.sym before init
        id.par = 1 # Process 0 takes part in the facto

        if ssp.issparse(A):
            if self.comm is None:
                # sequential
                self.driver.ICNTL[18] = 0 # centralized entry
                self.comm = MPI.COMM_SELF
            else:
                # distributed with global ordering
                self.driver.ICNTL[18] = 3 # distributed entry
            self.A_internal = A
        elif isinstance(A, DistMatrix):
            # Distributed matrix in local ordering
            self.driver.ICNTL[18] = 3 # distributed entry
            # internally, we switch to global ordering
            dd = A.dd
            self.A_internal = ssp.coo_matrix(
                dd.Ri.T.dot(
                    ssp.csc_matrix(A.local).dot(
                        dd.Ri)))
            self.comm = A.dd.comm
        id.comm_fortran = self.comm.py2f()

```

```
61          if self.ordering == "auto":
62              self.driver.ICNTL[7] = 7
63          elif self.ordering == "Scotch":
64              self.driver.ICNTL[7] = 3
65          elif self.ordering == "Metis":
66              self.driver.ICNTL[7] = 5
67
68          id.sym = self.symmetry
69          if self.symmetry>0:
70              id.sym = 3 - id.sym # 0 Ge, 1 SPD, 2 Sym
71              self.A_internal = ssp.tril(self.A_internal)
72
73          self.driver.initialize()
74
75          if self.verbose:
76              id.icntl[0:4] = [6, 0, 6, 3]
77              self.driver.ICNTL[11] = 2 # compute main stats
78          else:
79              id.icntl[0:4] = [0, 0, 0, 0]
80
81          self.driver.ICNTL[24] = 1 # null pivot detection
82
83          self.driver.set_A(self.A_internal)
84
85          self.is_forward = False
86
87      def factorize(self):
88          with TimeIt("MumpsDriver Analysis"):
89              self.driver.drive(1) # Analysis
90          with TimeIt("MumpsDriver Facto"):
91              self.driver.drive(2) # Facto
92
93      @TimeIt()
94      def solve(self, b):
95          A = self.A
96          if ssp.issparse(A) and self.comm is not None:
97              b_ = self.comm.reduce(b, root=0)
98          elif isinstance(A, DistMatrix):
99              b_ = b.centralize(root=0)
100         else:
101             b_ = b
102         self.driver.set_RHS(b_)
103         self.driver.drive(3)
104         x_ = self.driver.get_solution()
105         x_.shape = b.shape
```

```
106         x = x_
107         if self.comm is not None:
108             self.comm.Bcast(x_, root=0)
109         if isinstance(A, DistMatrix):
110             x = DistVector(b.dd.Ri.dot(x_), b.dd)
111         return x
```

Listing 44: `Mumps` (1/2).

Functions related to computing a Schur complement matrix with Mumps are presented in Section 4.3.10.1 (Listing 54).

4.3.8.4 The Pastix sparse direct solver

The `Pastix` class uses the `pypastix` module available with recent releases of Pastix[4]. It can factorize `scipy.sparse.spmatrix` matrices. Current versions do not handle distributed matrices, although they can exploit shared-memory parallelism internally. The `Pastix` class has 6 parameters:

- a `verbose` boolean or integer level (0/False, 1/True or 2).
- a `symmetry` boolean or integer. False/0 means not symmetric, True/1 means symmetric and 2 means SPD.
- a `refine` boolean to toggle iterative refinement.
- a `threads` number to choose the number of threads used by Pastix; `threads = "auto"` for automatic selection.
- a `check` boolean to toggle a verification of the solution (needs `x0`)
- a `x0` vector containing the solution.

```
1  class Pastix(LinearOperator):
2      """Factorize a matrix using pastix
3      """
4
5      defaults = {"verbose": False,
6                  "symmetry": False,
7                  "refine": True,
8                  "threads": 1,
9                  "check": False,
10                 "x0": None}
11
12     @TimeIt()
13     def setup(self, A, **kwargs):
```

[4] `https://gitlab.inria.fr/solverstack/pastix`

```
14          LinearOperator.setup(self, A, **kwargs)
15          self.init(A)
16          pypastix.task_analyze(self.pastix_data, self.spmA)
17          pypastix.task_numfact(self.pastix_data, self.spmA)
18
19      def init(self, A):
20          """ Register the options in iparm and dparm, and setup A"""
21          iparm, dparm = pypastix.initParam()
22          self.iparm, self.dparm = iparm, dparm
23          # Verbose
24          d = {0: pypastix.verbose.Not,
25               1: pypastix.verbose.Yes,
26               2: pypastix.verbose.No}
27          iparm[pypastix.iparm.verbose] = d[self.verbose]
28          # Threads
29          if self.threads=="auto":
30              iparm[pypastix.iparm.thread_nbr] = -1
31          else:
32              iparm[pypastix.iparm.thread_nbr] = self.threads
33          # Symmetry
34          d = {0: pypastix.factotype.LU,
35               1: pypastix.factotype.LDLT,
36               2: pypastix.factotype.LLT,
37               "SPD": pypastix.factotype.LLT}
38          self.factotype = d[self.symmetry]
39          self.iparm[pypastix.iparm.factorization] = self.factotype
40          # init
41          self.pastix_data = pypastix.init(iparm, dparm)
42          self.spmA = pypastix.spm(A)
43          if self.verbose:
44              self.spmA.printInfo()
45
46      @TimeIt()
47      def solve(self, b):
48          x = b.copy()
49          pypastix.task_solve(self.pastix_data, x)
50          if self.refine:
51              pypastix.task_refine(self.pastix_data, b, x)
52          if self.check and self.x0 is not None:
53              self.spmA.checkAxb(self.x0, b, x)
54          return x
```

Listing 45: `Pastix` (1/2).

Functions related to computing a Schur complement matrix with Pastix are presented in Section 4.3.10.1 (Listing 55).

4.3.8.5 Test of the direct solvers

The direct solvers presented above can be tested

```
from ddmpy import *
A_Gen = np.array(
    [[1./(1+i+2.*j) for i in range(3)]
     for j in range(3)])
A_SPD = A_Gen.T.dot(A_Gen)

solvers = (ScipyDirectSolver(),
           Pinv())
for s in solvers:
    for symmetry in False, True:
        A = A_SPD if symmetry else A_Gen
        b = A.dot(np.ones((3, 1)))
        s.setup(A, symmetry=symmetry)
        x = s.dot(b)
        print(s)
        print(x)
```

```
ScipyDirectSolver
  symmetry : False
  t_ScipyDirectSolver_setup : 8.10623168945e-05
  t_ScipyDirectSolver_solve : 3.69548797607e-05
[[ 1.]
 [ 1.]
 [ 1.]]
ScipyDirectSolver
  symmetry : True
  t_ScipyDirectSolver_setup : 0.000131130218506
  t_ScipyDirectSolver_solve : 6.69956207275e-05
[[ 1.]
 [ 1.]
 [ 1.]]
Pinv
  t_Pinv_setup : 0.000162839889526
  t_Pinv_solve : 5.00679016113e-06
[[ 1.]
 [ 1.]
 [ 1.]]
Pinv
  t_Pinv_setup : 0.000295877456665
  t_Pinv_solve : 1.00135803223e-05
[[ 1.]
```

```
 [ 1.]
 [ 1.]]
```

4.3.9 Iterative linear solvers

Contrary to direct solvers, iterative solvers find the solution of $\mathcal{A}x = b$ by building a sequence of approximate solutions that converge towards the exact solution. Currently, two iterative solvers are available in `ddmpy`: a CG solver for SPD matrices and a Generalized Conjugate Residual (GCR) algorithm (Eisenstat et al., 1983) for general (non SPD) matrices. A Generalized Minimal Residual (GMRES) solver (Saad and Schultz, 1986), and multipreconditioned solvers (Bridson and Greif, 2006; Greif et al., 2011; Spillane, 2016) are under development.

4.3.9.1 Conjugate gradient

The `cg()` function. The `cg()` function that we introduced in Listing 13 (Section 4.2.3) was in fact a reduced version of the `cg()` function provided in `ddmpy` that we present below in Listing 46. This `cg()` function in `ddmpy` is available as a replacement for the `scipy.sparse.linalg.cg()` function.

```
def cg(A, b, x0=None, tol=1e-5, maxiter=None, xtype=None,
       M=None, callback=None, ritz=False, save_x=False,
       debug=False, true_res=False):
    """Solves the linear problem Ax=b using the
    Conjugate Gradient algorithm. Interface compatible
     with scipy.sparse.linalg.cg

    Parameters
    ----------
    A : matrix-like object
        A is the linear operator on which we want to perform
        the solve operation. The only requirement on A is to
        provide a method dot(self, x) where x is a
        vector-like object.
    b : vector-like object
        b is the right-hand side of the system to solve. The
        only requirement on b is to provide the following
        methods: __len__(self) (not necessary if maxiter is
        provided), copy(self), __add__(self, w),
        __sub__(self, w), __rmul__(self, a), dot(self, w)
        where w is a vector-like and a is a scalar
    x0 : vector-like object
        starting guess for the solution, optional
    tol : float
        Relative tolerance to achieve, optional, default: 1e-5
```

```
26      maxiter : integer
27          Maximum number of iterations, optional, default: len(b)
28      xtype :
29          not used (compatibility with scipy.sparse.linalg.cg)
30      M : matrix-like object
31          Preconditioner for A, optional
32      callback : function
33          After each iteration, callback(x) is called, where x is
34          the current solution, optional
35      ritz : boolean
36          Store the dot products in order to compute the Ritz
37          values later, optional
38      save_x : boolean
39          Store the value of x at each iteration, optional
40      debug : boolean
41          print debug info, optional
42      true_res : boolean
43          recompute the residual at each iteration, optional
44
45      """
46      bb = b.T.dot(b)                     # ||b||^2 = b^T b
47      maxiter = A.shape[0] if maxiter is None else maxiter
48
49      with TimeIt("Instrumentation"):
50          global _cg_n_iter
51          _cg_n_iter=0
52          if ritz:
53              global _cg_omega, _cg_gamma
54              _cg_omega = np.zeros(maxiter)
55              _cg_gamma = np.zeros(maxiter)
56
57      # Initialization
58      x = 0 * b if x0 is None else x0
59      r = b - A.dot(x)                    # r = b - Ax
60      rr = r.T.dot(r)                     # ||r||^2 = r^T r
61      if rr / bb <= tol * tol:            # ||r||/||b|| <= eps
62          return x, 0
63      z = r if M is None else M.dot(r) # z = Mr
64      p = z.copy()                        # p = z
65      rz = r.T.dot(z)                     # r^T z
66
67      with TimeIt("Instrumentation"):
68          if save_x:
69              global _cg_x
70              _cg_x = [x]
```

```
71
72      for i in range(maxiter):
73          Ap = A.dot(p)                       # Ap
74          alpha = (rz / (p.T.dot(Ap)))[0,0]  # α = r^T z / p^T Ap
75          x += alpha * p                      # x = x + αp
76          if true_res:
77              r = b - A.dot(x)                # r = b − Ax
78          else:
79              r -= alpha * Ap                 # r = r − α Ap
80
81          with TimeIt("Instrumentation"):
82              if ritz:
83                  if i>0:
84                      _cg_gamma[i-1] = beta
85                  _cg_omega[i] = alpha
86              _cg_n_iter += 1
87              if callback:
88                  callback(x)
89              if save_x:
90                  _cg_x.append(x.copy())
91
92          rr = r.T.dot(r)                     # ||r||^2 = r^T r
93
94          if debug:
95              print("Iteration: {}, "
96                    "||r||_2/||b||_2 = {}"
97                    .format(i, np.sqrt(rr/bb)[0, 0]))
98
99          if rr / bb <= tol * tol:            # ||r|| / ||b|| ≤ ε
100             return x, 0
101
102         z = r if M is None else M.dot(r) # z = Mr
103         rz, rzold = r.T.dot(z), rz
104         beta = (rz / rzold)[0,0]            # β = r_i^T z_i / r_{i-1}^T z_{i-1}
105         p = z + beta * p                    # p = z + βp
106     return x, i
```

Listing 46: `cg()` standalone function.

The `ConjGrad` solver. This standalone `cg()` function is also available through a `ConjGrad` solver class with the following parameters:

- `x0`, an initial guess for the solution.
- `tol`, the stopping criterion (relative backward error).

- `maxiter`, the maximum number of iterations.
- `M`, a preconditioner.
- `callback`, a function called at each iteration with the current iterate solution as an argument.
- `ritz`, a boolean to enable the computation of the eigenvalues of the Hessenberg matrix (Ritz values of `A`) from coefficients computed during the iterations (see the `ritz_values()` method in Listing 48).
- `save_x`, a boolean to toggle the conservation of the intermediate solution at each iteration in an attribute `self.x_`.
- `setup_M`, a boolean to toggle the call of `M.setup()`.
- `debug`, to toggle the printing of debug info.
- `true_res`, to toggle the computation of the true residual `r = b - A @ x` at each iteration instead of using an iterative formula.

```
1  class ConjGrad(LinearOperator):
2      """Solve Ax=b using the CG algorithm
3
4      Optional parameters are:
5      - x0, initial guess for the solution, default=0*b
6      - tol, relative tolerance to achieve, default=1e-5
7      - maxiter, maximum number of iterations, default=len(b)
8      - M, preconditioner for A
9      - callback, after each iteration, callback(x) is called,
10       where x is the current solution
11     - ritz=True/False, whether to compute the ritz values
12       (approximate eigenvalues of A), default=False
13     - save_x=True/False, whether to store x at each
14       iteration, default=False
15     - setup_M=True/False, whether to try and call
16       M.setup(A) during the setup phase
17     - debug=True/False, whether to print debug info
18     - true_res=True/False, whether to recompute the true
19       residual instead of a recurrence formula
20     """
21
22     defaults = {"x0": None,
23                 "tol": 1e-5,
24                 "maxiter": None,
25                 "M": None,
26                 "callback": None,
```

```
                "ritz": False,
                "save_x": False,
                "setup_M": True,
                "debug": False,
                "true_res": False}

    @TimeIt()
    def setup(self, A, **kwargs):
        LinearOperator.setup(self, A, **kwargs)
        # We setup the preconditioner if possible and
        # asked for by the user
        if self.setup_M:
            if hasattr(self.M, "setup"):
                self.M.setup(A)

    @TimeIt()
    def solve(self, b):
        # For deflation, the preconditioner first
        # orthogonalize x0, through its self.M.x0 method
        x0_f = getattr(self.M, "x0", None)
        if x0_f is not None:
            r0 = b if self.x0 is None else (
                b - self.A.dot(self.x0))
            self.x0 = x0_f(r0)
        # reshape b
        if b.ndim==1:
            b = b.reshape((-1, 1))
        # Call the standalone cg function
        x, self.i = cg(self.A, b, self.x0, self.tol, self.maxiter,
                       None, self.M, self.callback, self.ritz,
                       self.save_x, self.debug)

        self.n_iter = _cg_n_iter
        self.parameters["n_iter"] = self.n_iter
        if self.ritz:
            self.omega = _cg_omega
            self.gamma = _cg_gamma
        if self.save_x:
            self.x_ = _cg_x
        return x
```

Listing 47: `ConjGrad` (1/2).

Computing the eigenvalues of the tridiagonal matrix. If the `ritz` parameter is set to `True`, it is possible, after the solve, to compute the eigenvalues of the tridiagonal

Lanczos matrix $\mathcal{T}$ of the CG algorithm (Burkitt and Irving, 1988). These eigenvalues are Ritz values of the preconditioned matrix $\mathcal{MA}$. The condition number ($\kappa(\mathcal{T}) = \lambda_{\max}/\lambda_{\min}$) gives an approximation of the condition number of the preconditioned matrix $\kappa(\mathcal{MA})$.

```
    def ritz_values(self):
        """Compute the ritz values of the Hessenberg matrix.

        Call this function after a solve has been performed
        with self.ritz==True
        """
        if self.n_iter>1:
            alpha = np.zeros(self.n_iter)
            alpha[0] = 1/self.omega[0]
            beta = np.zeros(self.n_iter-1)
            for i in range(self.n_iter-1):
                alpha[i+1] = (1/self.omega[i+1]
                              + self.gamma[i]/self.omega[i])
                beta[i] = (np.sqrt(max(self.gamma[i], 0))
                           / self.omega[i])
                T = (np.diag(alpha)
                     + np.diag(beta, 1)
                     + np.diag(beta, -1))
            lambda_ = la.eigvalsh(T)
        else:
            lambda_ = np.array([1])
        return lambda_
```

Listing 48: `ConjGrad` (2/2).

Testing the `ConjGrad` class. The `ConjGrad` class can be used in sequential

```
b = A_SPD.dot(np.ones((3, 1)))
s = ConjGrad(A_SPD, ritz=True, tol=1e-7)
x = s.dot(b)
print(s)
```

```
ConjGrad
  callback : None
  debug : False
  M : None
  maxiter : None
  ritz : True
  save_x : False
```

```
  setup_M : True
  tol : 1e-07
  true_res : False
  x0 : None
  t_ConjGrad_setup : 3.69548797607e-05
  n_iter : 3
  t_ConjGrad_solve : 0.000254154205322
array([  6.96932698e-06,   1.56060904e-02,   1.64729510e+00])
```

```
s.ritz_values()
```

```
array([  6.96932698e-06,   1.56060904e-02,   1.64729510e+00])
```

or in parallel as in Section 4.3.6.3.

4.3.9.2 Generalized conjugate residual

The GCR solver is an iterative solver suited for non-symmetric matrices (Eisenstat et al., 1983). It is a descent method that builds a $\mathcal{A}^T\mathcal{A}$-orthogonal ($\mathcal{M}^T\mathcal{A}^T$) set of search directions. Right preconditioning is used such that the norm of the residual used in the stopping criterion does not depend on the choice of a preconditioner $\mathcal{M}$.

As for the CG solver, a standalone `gcr()` and a `GCR` class are available.

The `gcr()` function.

```
def gcr(A, b, x0=None, tol=1e-5, maxiter=None, M=None):

    bb = b.T.dot(b)                    # ||b||^2 = b^T b
    maxiter = A.shape[0] if maxiter is None else maxiter

    with TimeIt("Instrumentation"):
        global _gcr_n_iter
        _gcr_n_iter=0

    # Initialization
    x = 0 * b if x0 is None else x0
    r = b - A.dot(x)

    rr = r.T.dot(r)
    if rr / bb <= tol * tol:
        return x, 0

    MP = []
    AMP = []
```

```
    for i in range(maxiter):
        Mr = M.dot(r) if M is not None else r.copy()
        MP.append(Mr)
        AMP.append(A.dot(MP[i]))
        # orthonormalization
        for j in range(i):              # (AMp_i)^T AMp_j = 0
            alpha = np.asscalar(AMP[i].T.dot(AMP[j]))
            MP[i] -= alpha * MP[j]
            AMP[i] -= alpha * AMP[j]
        normAMPi = np.asscalar(np.sqrt(AMP[i].T.dot(AMP[i])))
        MP[i] /= normAMPi
        AMP[i] /= normAMPi              # ||AMp_i||_2 = 1
        # update x and r
        rAMPi = np.asscalar(r.T.dot(AMP[i]))
        x += rAMPi * MP[i]              # x_{i+1} = x_i + (r_i^T AMp_i) Mp_i
        r -= rAMPi * AMP[i]             # r_{i+1} = r_i - (r_i^T AMp_i) AMp_i

        with TimeIt("Instrumentation"):
            _gcr_n_iter += 1

        rr = r.T.dot(r)
        if rr / bb <= tol * tol:
            return x, 0

    return x, i
```

Listing 49: `gcr` standalone function.

The `GCR` solver.

```
class GCR(LinearOperator):

    defaults = {"x0": None,
                "tol": 1e-5,
                "maxiter": None,
                "M": None,
                "setup_M": True}
    @TimeIt()
    def setup(self, A, **kwargs):
        LinearOperator.setup(self, A, **kwargs)
        # We setup the preconditioner if possible and
        # asked for by the user
        if self.setup_M:
            if hasattr(self.M, "setup"):
```

```
                self.M.setup(A)

    @TimeIt()
    def solve(self, b):
        # For deflation, the preconditioner first
        # orthogonalize x0, through its self.M.x0 method
        x0_f = getattr(self.M, "x0", None)
        if x0_f is not None:
            r0 = b if self.x0 is None else (
                b - self.A.dot(self.x0))
            self.x0 = x0_f(r0)
        # reshape b
        if b.ndim==1:
            b = b.reshape((-1, 1))
        # Call the standalone gcr function
        x, self.i = gcr(self.A, b, self.x0, self.tol,
                        self.maxiter, self.M)

        self.n_iter = _gcr_n_iter
        self.parameters["n_iter"] = self.n_iter
        return x
```

Listing 50: GCR

Testing the GCR class.

```
from ddmpy import *

K = DistMatrix.test()
b = K @ DistVector(np.ones((4, 1)), K.dd)
dd = K.dd
x = (GCR(K) @ b).centralize(root=0)
if dd.rank == 0:
    print(x[:,0])
```

```
mpirun -np 3 python3 testGCR.py
```

```
[ 1.  1.  1.  1.  1.  1.  1.]
```

4.3.10 Hybrid linear solvers

4.3.10.1 Schur solver

From a linear system $\mathcal{K}u = f$ of size (n, n) and a partition of the indices $\{0, 1, \ldots, n-1\} = \mathcal{I} \cup \Gamma$, one can reorder the rows and columns (symmetric permutation) such that the system

becomes

$$\begin{pmatrix} \mathcal{K}_{\mathcal{I}\mathcal{I}} & \mathcal{K}_{\mathcal{I}\Gamma} \\ \mathcal{K}_{\Gamma\mathcal{I}} & \mathcal{K}_{\Gamma\Gamma} \end{pmatrix} \begin{pmatrix} u_{\mathcal{I}} \\ u_{\Gamma} \end{pmatrix} \begin{pmatrix} f_{\mathcal{I}} \\ f_{\Gamma} \end{pmatrix}.$$

If $\mathcal{K}_{\mathcal{I}\mathcal{I}}$ is not singular, this system is equivalent to

$$\begin{cases} \mathcal{K}_{\mathcal{I}\mathcal{I}} u_{\mathcal{I}} = f_{\mathcal{I}} - \mathcal{K}_{\mathcal{I}\Gamma} u_{\Gamma}, \\ \mathcal{S} u_{\Gamma} = \widetilde{f}_{\Gamma}, \end{cases}$$

where $\mathcal{S} = \mathcal{K}_{\Gamma\Gamma} - \mathcal{K}_{\Gamma\mathcal{I}} \mathcal{K}_{\mathcal{I}\mathcal{I}}^{-1} \mathcal{K}_{\mathcal{I}\Gamma}$ and $\widetilde{f}_{\Gamma} = f_{\Gamma} - \mathcal{K}_{\Gamma\mathcal{I}} \mathcal{K}_{\mathcal{I}\mathcal{I}}^{-1} f_{\mathcal{I}}$ (see Section 2.4).

The solution of the linear system $\mathcal{K}u = f$ can therefore be performed in four steps using two different solvers: one solver to handle $\mathcal{K}_{\mathcal{I}\mathcal{I}}$ (lines 17-51 of Listing 51) and another solver to solve the Schur complement system (lines 52-54 of Listing 51). The four steps are:

$$\text{factorize } \mathcal{K}_{\mathcal{I}\mathcal{I}} \text{ and compute } \mathcal{S} = \mathcal{K}_{\Gamma\Gamma} - \mathcal{K}_{\Gamma\mathcal{I}} \mathcal{K}_{\mathcal{I}\mathcal{I}}^{-1} \mathcal{K}_{\mathcal{I}\Gamma}, \tag{4.1}$$

$$\text{compute } \widetilde{f}_{\Gamma} = f_{\Gamma} - \mathcal{K}_{\Gamma\mathcal{I}} \mathcal{K}_{\mathcal{I}\mathcal{I}}^{-1} f_{\mathcal{I}}, \tag{4.2}$$

$$\text{solve } \mathcal{S} u_{\Gamma} = \widetilde{f}_{\Gamma}, \tag{4.3}$$

$$\text{solve } \mathcal{K}_{\mathcal{I}\mathcal{I}} u_{\mathcal{I}} = f_{\mathcal{I}} - \mathcal{K}_{\mathcal{I}\Gamma} u_{\Gamma}. \tag{4.4}$$

Computing a Schur complement matrix with any solver The `SchurSolver` class in `ddmpy` solves the linear problem $\mathcal{K}u = f$ using this method. It has four parameters:

- a `local_solver` used on $\mathcal{K}_{\mathcal{I}\mathcal{I}}$.
- an `interface_solver` used on $\mathcal{S}$.
- an `interface` list that contains the indices in Γ.
- a `symmetry` boolean.

The Schur complement matrix $\mathcal{S}$ is available as a `S` attribute of the solver after the `setup()` method has been called.

Some solvers such as `Mumps` and `Pastix` in sections 4.3.8.3 and 4.3.8.4 have optimized functions to compute the Schur complement system by performing a partial factorization. The `SchurSolver` class tries using the `local_solver.schur()` if it exists to compute $\mathcal{S}$. If `local_solver` does not provide such a method (for instance, if `local_solver` is a `ScipyDirectSolverr` instance), the Schur complement matrix is computed using Equation (4.1) (lines 22-51 of Listing 51).

```
1 class SchurSolver(LinearOperator):
2     """ Solve a system using a Schur complement matrix
3
4     Use local_solver to eliminate all variables in A that are
5     not in interface and compute the Schur complement matrix S. Then, use
```

```
6       interface_solver to solve S.
7       """
8
9       defaults = {"interface": None,
10                  "local_solver": ScipyDirectSolver,
11                  "interface_solver": GCR,
12                  "symmetry": False}
13
14      @TimeIt()
15      def setup(self, A, **kwargs):
16          LinearOperator.setup(self, A, **kwargs)
17          try:
18              self.S = self.local_solver.schur(self.A,
19                                               self.interface,
20                                               symmetry=self.symmetry)
21          except AttributeError:
22              interface = self.interface
23              n = self.A.shape[0]
24              nG = len(interface)
25              nI = n - nG
26              interior = np.setdiff1d(range(n), interface)
27              RI = ssp.csc_matrix((np.ones_like(interior),
28                                   (range(nI), interior)),
29                                  shape=(nI, n))     # R_I
30              RG = ssp.csc_matrix((np.ones_like(interface),
31                                   (range(nG), interface)),
32                                  shape=(nG, n))     # R_Γ
33              self.RI, self.RG = RI, RG
34              if isinstance(A, np.ndarray):
35                  A = ssp.csc_matrix(A)
36              with TimeIt("Schur_AII"):
37                  AII = RI.dot(A.dot(RI.T))          # K_II
38              self.local_solver.setup(AII,           # K_II^-1
39                                      symmetry=self.symmetry)
40              with TimeIt("Schur_AIG"):
41                  AIG = (RI.dot(A.dot(RG.T))).A      # K_IΓ
42              if not self.symmetry:
43                  with TimeIt("Schur_AGI"):
44                      AGI = (RG.dot(A.dot(RI.T))).A # K_ΓI
45              with TimeIt("Schur_AGG"):
46                  self.S = (RG.dot(A.dot(RG.T))).A  # S = K_ΓΓ
47              with TimeIt("Schur_AII^{-1}AIG"):     #   -K_ΓI K_II^-1 K_IΓ
48                  if self.symmetry:
49                      self.S -= AIG.T.dot(self.local_solver.dot(AIG))
50                  else:
```

```
51                      self.S -= AGI.dot(self.local_solver.dot(AIG))
52          if self.interface_solver is not None:
53              self.interface_solver.setup(self.S,
54                                          symmetry=self.symmetry)
```

Listing 51: `SchurSolver` (1/3).

To compute $\widetilde{f}_\Gamma$ from f and u from u_Γ (see equations (4.2) and (4.4)), two methods are used: `b2f()` and `y2x()`, respectively. In the following code, the variables `A`, `b`, `x`, `y` and `f` are used to represent $\mathcal{K}$, f, u, u_Γ and $\widetilde{f}_\Gamma$, respectively. For simplification a vector

$$\begin{pmatrix} v_\mathcal{I} \\ v_\Gamma \end{pmatrix} = \left(v_\mathcal{I}^T \; v_\Gamma^T\right)^T$$

is noted $[v_\mathcal{I}, v_\Gamma]^T$ in the code comments.

```
1       @TimeIt()
2       def b2f(self, b): # f̃_Γ = f_Γ − K_ΓI K_II^{-1} f_I
3           try:
4               f = self.local_solver.b2f(b)
5           except AttributeError:
6               RI, RG = self.RI, self.RG
7               bI = RI.dot(b)                          # b_I
8               bI_I = self.local_solver.dot(bI) # K_II^{-1} f_I
9               bI_I_ = RI.T.dot(bI_I)                  # [K_II^{-1} f_I, 0]^T
10              f_ = self.A.dot(bI_I_)                  # [f_I, K_ΓI K_II^{-1} f_I]^T
11              f = RG.dot(b - f_)                      # f_Γ − K_ΓI K_II^{-1} f_I
12          return f
13
14      @TimeIt()
15      def y2x(self, y, b): # u_I = K_II^{-1}(f_I − K_IΓ u_Γ)
16          # y = x_Γ
17          if len(b.shape)==1:
18              b = b.reshape((-1,1))
19          try:
20              x = self.local_solver.y2x(y, b)
21          except AttributeError:
22              RI, RG = self.RI, self.RG
23              tmp = RG.T.dot(y)           # [0, u_Γ]^T
24              tmp = b - self.A.dot(tmp)   # [f_I − K_IΓ u_Γ, f_Γ − K_ΓΓ u_Γ]^T
25              tmp = RI.dot(tmp)           # f_I − K_IΓ u_Γ
26              tmp = self.local_solver.dot(tmp) # K_II^{-1}(f_I − K_IΓ u_Γ)
27              tmp = RI.T.dot(tmp)         # [K_II^{-1}(f_I − K_IΓ u_Γ), 0]^T
28              x = RG.T.dot(y) + tmp       # [K_II^{-1}(f_I − K_IΓ u_Γ), u_Γ]^T
29          return x
```

Listing 52: `SchurSolver` (2/3).

If an iterative solver is used to solve the interface system (Equation (4.3)) up to a tolerance `interface_solver.tol` computed as a bound on the backward error $\frac{||f-\mathcal{K}u||}{||f||}$, `tol` is modified to take into account the fact that the normalization changes from $||\widetilde{f}_\Gamma||$ to $||f||$. This way, the tolerance is computed on the backward error of the full system $\mathcal{K}u = f$ instead of the interface system $\mathcal{S}u_\Gamma = \widetilde{f}_\Gamma$ provided to the interface solver.

```
    @TimeIt()
    def solve(self, b): # solve Ku = f
        f = self.b2f(b) # f̃_Γ = f_Γ − K_ΓI K_II^{-1} f_I
        param = self.interface_solver.parameters
        if "tol" in param:
            # ||r||/||f̃_Γ|| = ||r||/||f|| ||f||/||f̃_Γ||
            bb = b.T.dot(b) # ||f||^2
            ff = f.T.dot(f) # ||f̃_Γ||^2
            ratio = np.sqrt(bb/ff)[0, 0] # ||f||/||f̃_Γ||
            param["global_tol"] = param["tol"]
            param["tol"] *= ratio
        y = self.interface_solver.dot(f) # solve Su_Γ = f̃_Γ
        x = self.y2x(y, b) # u_I = K_II^{-1} f_I − K_IΓ u_Γ
        return x # [u_I, u_Γ]^T
```

Listing 53: `SchurSolver` (3/3).

Computing a Schur complement matrix with Mumps. The `Mumps` solver provides functions to compute a Schur complement matrix.

```
    @TimeIt()
    def schur(self, A, interface, **kwargs):
        """ Perform a partial factorization and compute the Schur complement matrix S
        LinearOperator.setup(self, A, **kwargs)
        self.init()
        self.driver.set_schur_listvar(interface)
        self.factorize()
        self.S = self.driver.get_schur()
        self.interface = interface
        return self.S

    @TimeIt()
    def b2f(self, b):
```

```
        """ Compute the reduced RHS f from the complete RHS b """
        self.driver.set_RHS(b)
        f = self.driver.schur_forward()
        self.is_forward=True
        f.shape = (len(f), 1)
        return f

    @TimeIt()
    def y2x(self, y, b):
        """ Compute the complete solution x from the Schur complement solution y """
        if not self.is_forward:
            self.b2f(b)

        x = self.driver.schur_backward(y)
        x.shape = b.shape
        self.is_forward = False
        return x

    def __del__(self):
        try:
            self.driver.finalize()
        except (TypeError, AttributeError):
            pass
```

Listing 54: `Mumps` (2/2).

Computing a Schur complement matrix with Pastix. The `Pastix` solver provides functions to compute a Schur complement matrix.

```
    @TimeIt()
    def schur(self, A, interface, **kwargs):
        """Perform a partial factorization and compute
        the Schur complement matrix S
        """
        LinearOperator.setup(self, A, **kwargs)
        self.init(A)
        self.iparm[pypastix.iparm.schur_solv_mode] = (
                pypastix.solv_mode.Interface)
        schur_list = np.asarray(interface, pypastix.pastix_int)
        self.schur_list = schur_list + self.spmA.findBase()
        pypastix.setSchurUnknownList(self.pastix_data,
                                     self.schur_list)
        pypastix.task_analyze(self.pastix_data, self.spmA)
        pypastix.task_numfact(self.pastix_data, self.spmA)
```

```
16          nschur = len(schur_list)
17          self.nschur = nschur
18          self.S = np.zeros((nschur, nschur),
19                            order='F', dtype=A.dtype)
20          pypastix.getSchur(self.pastix_data, self.S)
21          return self.S
22
23      @TimeIt()
24      def b2f(self, b):
25          """Compute the reduced RHS f from the
26          complete RHS b
27          """
28          x = b.copy()
29          pypastix.subtask_applyorder(
30              self.pastix_data, pypastix.dir.Forward, x)
31          if self.factotype == pypastix.factotype.LLT:
32              pypastix.subtask_trsm(
33                  self.pastix_data, pypastix.side.Left,
34                  pypastix.uplo.Lower, pypastix.trans.NoTrans,
35                  pypastix.diag.NonUnit, x)
36          else:
37              pypastix.subtask_trsm(
38                  self.pastix_data, pypastix.side.Left,
39                  pypastix.uplo.Lower, pypastix.trans.NoTrans,
40                  pypastix.diag.Unit, x)
41          if self.factotype == pypastix.factotype.LDLT:
42              pypastix.subtask_diag(self.pastix_data, x)
43          self.x = x
44          f = x[-self.nschur:]
45          return f
46
47      @TimeIt()
48      def y2x(self, y, b):
49          """ Compute the complete solution x
50          from the Schur complement solution y
51          """
52          x = self.x.copy()
53          x[-self.nschur:] = y
54          if self.factotype == pypastix.factotype.LDLT:
55              pypastix.subtask_trsm(
56                  self.pastix_data, pypastix.side.Left,
57                  pypastix.uplo.Lower, pypastix.trans.Trans,
58                  pypastix.diag.Unit, x)
59          elif self.factotype == pypastix.factotype.LLT:
60              pypastix.subtask_trsm(
```

```
                self.pastix_data, pypastix.side.Left,
                pypastix.uplo.Lower, pypastix.trans.Trans,
                pypastix.diag.NonUnit, x)
        else: # LU
            pypastix.subtask_trsm(
                self.pastix_data, pypastix.side.Left,
                pypastix.uplo.Upper, pypastix.trans.NoTrans,
                pypastix.diag.NonUnit, x)
        pypastix.subtask_applyorder(
            self.pastix_data, pypastix.dir.Backward, x)
        if self.check and self.x0 is not None:
            self.spmA.checkAxb(self.x0, b, x)
        return x
```

Listing 55: `Pastix` (2/2).

Testing the `SchurSolver` class. The `SchurSolver` class is tested on a general problem

```
from ddmpy import *
n = 5

A = ssp.spdiags([d*np.ones(n) for d in 3, -2, -1], [0, 1, -1], n, n)
b = A.dot(np.ones((n, 1)))

s = SchurSolver(A,
                interface=[0, 1],
                symmetry=False)
s.S
s.dot(b)
```

```
array([[ 3.        , -2.        ],
       [-1.        ,  2.06666667]])
array([[ 1.],
       [ 1.],
       [ 1.],
       [ 1.],
       [ 1.]])
```

and a SPD linear system

```
A = ssp.spdiags([d*np.ones(n) for d in 2, -1, -1], [0, 1, -1], n, n)
b = A.dot(np.ones((n, 1)))
```

```
3
4 s = SchurSolver(A,
5                 interface=[0, 1],
6                 symmetry=True,
7                 interface_solver=ConjGrad)
8 s.S
9 s.dot(b)
```

```
array([[ 2.  , -1.  ],
       [-1.  ,  1.25]])
array([[ 1.],
       [ 1.],
       [ 1.],
       [ 1.],
       [ 1.]])
```

One can check solver information by printing the solver

```
1 print(s)
```

```
SchurSolver
  interface : [0, 1]
  interface_solver : ConjGrad
    callback : None
    debug : False
    M : None
    maxiter : None
    ritz : False
    save_x : False
    setup_M : True
    tol : 1.37198868114e-05
    true_res : False
    x0 : None
    symmetry : True
    t_ConjGrad_setup : 4.6968460083e-05
    global_tol : 1e-05
    n_iter : 2
    t_ConjGrad_solve : 0.000164985656738
  local_solver : ScipyDirectSolver
    symmetry : True
    t_ScipyDirectSolver_setup : 0.000147104263306
    t_ScipyDirectSolver_solve : 0.000120878219604
  symmetry : True
```

```
t_SchurSolver_setup : 0.00435090065002
t_SchurSolver_b2f : 0.000667095184326
t_SchurSolver_y2x : 0.000555038452148
t_SchurSolver_solve : 0.00150990486145
```

4.3.10.2 Distributed Schur solver

A distributed subclass of the `SchurSolver` class can be used to solve a distributed linear system using a primal interface formulation (see Section 2.5.2). The only differences with the `SchurSolver` class are the following:

- there is no need for an `interface` parameter: the `interface` is deduced from the `dd` attribute of the `DistMatrix` object representing the distributed matrix $\mathcal{A} = \sum_{i=1}^{N} \mathcal{R}_{\Omega_i}^T \mathcal{A}_i \mathcal{R}_{\Omega_i}$.
- the `interface_solver` parameter must accept a `DistMatrix` object: a `ConjGrad` or a `Mumps` solver can be used as an `interface_solver`, whereas a `ScipyDirectSolver` or `Pastix` solver can not.

```
class DistSchurSolver(SchurSolver):

    defaults = {"local_solver": ScipyDirectSolver,
                "interface_solver": GCR,
                "symmetry": False}

    @TimeIt()
    def setup(self, A, **kwargs):
        LinearOperator.setup(self, A, **kwargs)
        self.interface = A.dd.interface
        self.local_schur = SchurSolver(A.local,
                                       interface=self.interface,
                                       local_solver=self.local_solver,
                                       interface_solver=None,
                                       symmetry=self.symmetry)
        self.S = DistMatrix(self.local_schur.S, A.dd.interface_dd())
        if self.interface_solver is not None:
            self.interface_solver.setup(self.S)

    @TimeIt()
    def b2f(self, b):
        bi = b.dd.D * b.local
        fi = self.local_schur.b2f(bi)
        f = DistVector(fi, self.S.dd, assemble=True)
        return f

    @TimeIt()
```

```
28      def y2x(self, y, b):
29          xi = self.local_schur.y2x(y.local, b.local)
30          x = DistVector(xi, b.dd)
31          return x
```

Listing 56: `DistSchurSolver`

Testing the `DistSchurSolver` class. The `DistSchurSolver` is tested on the distributed matrix `DistMatrix.test()`.

```
from ddmpy import *

K = DistMatrix.test()
b = K @ DistVector(np.ones((4, 1)), K.dd)
dd = K.dd
s = DistSchurSolver(K)
x = (s @ b).centralize(root=0)
if dd.rank == 0:
    print(x[:,0])
```

```
mpirun -np 3 python3 testDistSchurSolver.py
```

```
[ 1.  1.  1.  1.  1.  1.  1.]
```

4.3.10.3 N-Lagrange formulation

The N-Lagrange formulation introduced in Section 2.6.5 is implemented in `ddmpy`. This formulation is used on the interface system, and as such, it can be seen as a reformulation of the third step (Equation (4.3)) of the Schur solver. The system $\mathcal{S}u_\Gamma = \widetilde{f}_\Gamma$ is solved in four steps

$$\text{factorize } \widehat{\mathcal{S}}_i = \mathcal{S}_i + \mathcal{T}_i, \tag{4.5}$$

$$\text{compute } RHS_i = \sum_{j\in\mathcal{N}(i)} \left(\mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_j}^T\mathcal{T}_j + \mathcal{T}_i W_{ij}\mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_j}^T\right)\widehat{\mathcal{S}}_j^{-1}\widetilde{f}_{\Gamma_j}^{(j)}, \tag{4.6}$$

$$\text{solve } \lambda_i + \sum_{j\in\mathcal{N}(i)} \left[\mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_j}^T - \left(\mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_j}^T\mathcal{T}_j + \mathcal{T}_i W_{ij}\mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_j}^T\right)\widehat{\mathcal{S}}_j^{-1}\right]\lambda_j = RHS_i, \tag{4.7}$$

$$\text{solve } u_{\Gamma_i} = \widehat{\mathcal{S}}_i^{-1}(\widetilde{f}_{\Gamma_i}^{(i)} + \lambda_i). \tag{4.8}$$

The other steps in equations (4.1), (4.2) and (4.4) are unchanged.

The `NLagrangeSolver` solver has 5 parameters:

- `Ti`, a scalar to add on the interface.

- `local_solver`, a solver used to compute the Schur complement matrix $\mathcal{S}_i$.
- `interface_solver`, a solver used to solve Equation 4.7.
- `Sih_solver`, a solver used to factorize the local matrix $\mathcal{S}$.
- `symmetry`, the symmetry of the matrix.

```python
class NLagrangeSolver(DistSchurSolver):
    defaults = {"Ti": 1,
                "local_solver": ScipyDirectSolver,
                "interface_solver": GCR,
                "Sih_solver": ScipyDirectSolver,
                "symmetry": False}

    @TimeIt()
    def setup(self, A, **kwargs):
        LinearOperator.setup(self, A, **kwargs)
        # Compute the Schur
        interface_solver = self.parameters["interface_solver"]
        self.parameters["interface_solver"] = None
        DistSchurSolver.setup(self, A, **kwargs)
        self.parameters["interface_solver"] = interface_solver
        # Factorize Sih
        self.Sih_solver.setup(self.S.local
                              + self.Ti*ssp.eye(A.dd.nG))
        # compute W
        dd = self.S.dd
        self.W = {}
        W_ = np.zeros((dd.ni, 1))
        for n, i in dd.neighbors.items():
            self.W[n] = 1 - W_[i]
            W_[i] = 1
        self.operator = self.NLOperator(self)
        self.interface_solver.setup(self.operator)

    def communicate(self, ui, mui):                # u_i, μ_i
        dd = self.S.dd
        res = np.zeros_like(ui)
        ui_mui = np.hstack((ui, mui))              # [u_i, μ_i]
        send = {n: ui_mui[dd.neighbors[n], :]
                for n in dd.neighbors}
        for j, uj_muj in neighborSendRecv(send):
            uj_muj = uj_muj.reshape((-1, 2))       # R_Γi R_Γj^T [u_j, μ_j]
            uj = uj_muj[:,0:1]                     # R_Γi R_Γj^T u_j
```

```
38              muj = uj_muj[:,1:2]                     # R_Γi R_Γj^T μ_j
39              i = dd.neighbors[j]
40              res[i] += (muj                          # R_Γi R_Γj^T μ_j
41                         + self.Ti * self.W[j] * uj)# T_i W_ij R_Γi R_Γj^T u_j
42          return res                # Σ_{j∈N(i)} R_Γi R_Γj^T μ_j + T_i W_ij R_Γi R_Γj^T u_j
43
44      # inner class representing the linear operator
45      class NLOperator(object):
46
47          def __init__(self, NLSolver):
48              self.NLSolver = NLSolver
49              ni = np.array(NLSolver.S.dd.ni)
50              n = ni.copy()
51              self.NLSolver.A.dd.comm.Allreduce(ni, n)
52              self.shape = (np.asscalar(n), np.asscalar(n))
53
54          @TimeIt()
55          def dot(self, l):
56              li = l.local
57              SihI = self.NLSolver.Sih_solver
58              ui = - SihI.dot(li)
59              mui = li + self.NLSolver.Ti*ui
60              li_ = li + self.NLSolver.communicate(ui, mui)
61              return DistVector(li_, l.dd, compatible=False)
62
63      @TimeIt()
64      def solve(self, b):
65          with TimeIt("rhs"):
66              bi = b.dd.D * b.local
67              fi = self.local_schur.b2f(bi) # f̃_Γi^(i)
68              SihI = self.Sih_solver # Ŝ_i^{-1}
69              ui = SihI.dot(fi)      # u_i = Ŝ_i^{-1} f̃_Γi^(i)
70              mui = self.Ti*ui       # μ_i = T_i u_i
71              # Σ R_Γi R_Γj^T μ_j + T_i W_ij R_Γi R_Γj^T u_j
72              rhs = DistVector(self.communicate(ui, mui),
73                               self.A.dd, compatible=False)
74          li = self.interface_solver.solve(rhs).local # λ_i
75          yi = SihI.dot(fi + li)     # u_Γi = Ŝ_i^{-1}(f̃_Γi^(i) + λ_i)
76          xi = self.local_schur.y2x(yi, b.local) # [u_Ii, u_Γi]^T
77          x = DistVector(xi, b.dd)
78          return x
```

Listing 57: `NLagrangeSolver`.

Testing the `NLagrangeSolver` class. The `NLagrangeSolver` class is tested on the same distributed matrix

```
from ddmpy import *

K = DistMatrix.test()
dd = K.dd
b = K.dot(DistVector(np.ones((4, 1)), dd))
s = NLagrangeSolver(K)
x = s.dot(b).centralize(root=0)
if dd.rank == 0:
    print(x[:,0])
    print(s)
```

```
mpirun -np 3 python testNLagrangeSolver.py
```

```
[ 1.  1.  1.  1.  1.  1.  1.]
NLagrangeSolver
  interface_solver : GCR
    M : None
    maxiter : None
    setup_M : True
    tol : 1e-05
    x0 : None
    t_GCR_setup : 3.79085540771e-05
    n_iter : 9
    t_GCR_solve : 0.0139119625092
  local_solver : ScipyDirectSolver
    symmetry : False
    t_ScipyDirectSolver_setup : 0.000228881835938
    t_ScipyDirectSolver_solve : 3.93390655518e-05
  Sih_solver : ScipyDirectSolver
    symmetry : False
    t_ScipyDirectSolver_setup : 9.29832458496e-05
    t_ScipyDirectSolver_solve : 0.00149488449097
  symmetry : False
  Ti : 1
  t_NLagrangeSolver_setup : 0.0117557048798
  t_NLagrangeSolver_solve : 0.0156950950623
```

4.3.11 Centralizing a distributed problem

A distributed linear system $\mathcal{A}x = b$ can be solved using a local (*i.e.*, not distributed) solver such as `ScipyDirectSolver` or `Pastix` through the `CentralizedSolver` class:

the `DistMatrix` $\mathcal{A}$ and `DistVector` b are assembled on one or all processes using the `centralize()` methods of these objects. The `local_solver` can then be used to compute the global solution `x_global` and broadcast it to other processes if needed. The `CentralizedSolver` has two parameters:

- `local_solver`, a `Solver` used locally to solve the centralized problem.
- `root`, the MPI rank of the process that will hold the `local_solver` instance and execute `local_solver.setup(A)` and `local_solver.solve(b)`. If `None`, all processes run a different instance of `local_solver` and perform these operations redundantly.

```
class CentralizedSolver(LinearOperator):

    defaults = {"local_solver": ScipyDirectSolver,
                "root": None}

    @TimeIt()
    def setup(self, A, **kwargs):
        LinearOperator.setup(self, A, **kwargs)
        with TimeIt("centralize_A"):
            self.A_global = A.centralize(self.root)
        if self.root is None or self.root==A.dd.rank:
            self.local_solver.setup(self.A_global)
            self.A_global_inv = self.local_solver

    @TimeIt()
    def solve(self, b):
        with TimeIt("centralize_b"):
            b_global = b.centralize(self.root)
        if self.root is None:
            x_global = self.A_global_inv.dot(b_global)
        else:
            if self.root==b.dd.rank:
                x_global = self.A_global_inv.dot(b_global)
            else:
                x_global = np.empty(b_global.shape,
                                    dtype=b.local.dtype)
            with TimeIt("distribute_x"):
                b.dd.comm.Bcast(x_global, root=self.root)
        xi = x_global[b.dd.global_indices, :]
        return DistVector(xi, b.dd)
```

Listing 58: `CentralizedSolver`.

Testing the `CentralizedSolver` class. The `CentralizedSolver` class is used to solve the same distributed matrix using the (sequential) `ScipyDirectSolver` solver on process 0. the `TimeIt` output shows that the `ScipyDirectSolver` `setup()` and `solve()` methods are only called on process 0.

```
from ddmpy import *

K = DistMatrix.test()
dd = K.dd
b = K.dot(DistVector(np.ones((4, 1)), dd))
TimeIt.reset()
s = CentralizedSolver(K, local_solver=ScipyDirectSolver, root=0)
x = s.dot(b)
for i in range(3):
    dd.comm.Barrier()
    if dd.rank == i:
        print("rank: {}\n{}\n".format(dd.rank, TimeIt()))
```

```
mpirun -np 3 python testCentralizedSolver.py
```

```
rank: 0
CentralizedSolver setup     | 0 | 0.0001321 | 0.0018079 | 0.0019400
!  ScipyDirectSolver setup | 1 | 0.0014982 | 0.0004280 | 0.0019262
CentralizedSolver solve     | 0 | 0.0019500 | 0.0001130 | 0.0020630
!  centralize_b             | 1 | 0.0019531 | 0.0000379 | 0.0019910
!  ScipyDirectSolver solve | 1 | 0.0020001 | 0.0000319 | 0.0020320
!  distribute_x             | 1 | 0.0020392 | 0.0000119 | 0.0020511

rank: 1
CentralizedSolver setup     | 0 | 0.0001781 | 0.0006111 | 0.0007892
CentralizedSolver solve     | 0 | 0.0008051 | 0.0012598 | 0.0020649
!  centralize_b             | 1 | 0.0008101 | 0.0000429 | 0.0008531
!  distribute_x             | 1 | 0.0008662 | 0.0011818 | 0.0020480

rank: 2
CentralizedSolver setup     | 0 | 0.0001822 | 0.0006020 | 0.0007842
CentralizedSolver solve     | 0 | 0.0008042 | 0.0012648 | 0.0020690
!  centralize_b             | 1 | 0.0008092 | 0.0000479 | 0.0008571
!  distribute_x             | 1 | 0.0008702 | 0.0011818 | 0.0020521
```

4.3.12 Domain decomposition preconditioners

A preconditioner $\mathcal{M} = \mathcal{M}_L \mathcal{M}_R$ can be used to accelerate an iterative solver by replacing a linear system $\mathcal{A}x = b$ by the equivalent system $\mathcal{M}_L \mathcal{A} \mathcal{M}_R y = \mathcal{M}_L b$, where $x = \mathcal{M}_R y$.

A good preconditioner for $\mathcal{A}x = b$ should be close to $\mathcal{A}^{-1}$, in some sense, and cheap to compute and apply. In `ddmpy`, the one-level and two-level aS preconditioners introduced in Section 2.7 and analyzed in more details in Chapter 3 are implemented.

4.3.12.1 One-level aS preconditioners

Abstract Schwarz (aS) preconditioner. Since preconditioners are linear operators built from a matrix, they are implemented as subclasses of the `LinearOperator` class presented in Section 4.3.7. One-level aS preconditioners, introduced in Section 2.7.2, are implemented using an `AbstractSchwarz` class; they can be written as

$$\mathcal{M}_{aS} = \sum_{i=1}^{N} \mathcal{R}_i^T \widehat{\mathcal{A}}_i^{\dagger} \mathcal{R}_i,$$

and are very similar to distributed matrices $\mathcal{A} = \sum_{i=1}^{N} \mathcal{R}_i^T \mathcal{A}_i \mathcal{R}_i$. The `AbstractSchwarz` class is therefore also a subclass of the `DistMatrix` class. The local matrix $\widehat{\mathcal{A}}_i$ in the definition of $\mathcal{M}_{aS}$ is built from $\mathcal{A}_i$ in several optional steps:

- an *assembly* step $\mathcal{A}_i \to \widehat{\mathcal{A}}_i = \mathcal{R}_i \mathcal{A} \mathcal{R}_i^T = \sum_{j \in \mathcal{N}(i)} \mathcal{R}_i \mathcal{R}_j^T \mathcal{A}_j \mathcal{R}_j \mathcal{R}_i^T$.
- an *addition* step $\widehat{\mathcal{A}}_i \to \widehat{\mathcal{A}}_i + \mathcal{T}_i$.
- a *partition of unity* step $\widehat{\mathcal{A}}_i \to D_i^{-1} \widehat{\mathcal{A}}_i D_i^{-1}$ where D_i is a partition of unity such that $\sum_{i=1}^{N} \mathcal{R}_i^T D_i \mathcal{R}_i = I$ (I is the identity matrix). D_i can be computed using the multiplicity (each subdomain has the same weight) or the local matrix (the weight for an unknown is proportional to the corresponding value on the diagonal of $\mathcal{A}_i$), or be a boolean weight (only one subdomain is 1, its neighbors are 0).

Then, $\widehat{\mathcal{A}}_i$ is factorized.

The `AbstractSchwarz` class has the following parameters:

- `assemble`, a boolean to toggle the assembly operation.
- `Ti`, a matrix $\mathcal{T}_i$ to add on the interface. Can be a scalar or an interface vector; in that case, it represents the diagonal of a diagonal matrix. If `None`, the addition step is not performed.
- `Di`, a method to compute the partition of unity D_i. If `None`, no partition of unity is used. If `"multiplicity"`, each subdomain has a weight of 1. If `"matrix"`, the diagonal of $\mathcal{A}_i$ is used as a weight. If `"boolean"`, the `DomainDecomposition.D` boolean partition of unity is used.
- `local_solver` the solver used to factorize $\widehat{\mathcal{A}}_i$.

```
1 class AbstractSchwarz(LinearOperator, DistMatrix):
2
3     defaults = {"assemble": True,
```

```
4                 "Ti": None,
5                 "Di": None,
6                 "local_solver": ScipyDirectSolver}
7
8     @TimeIt()
9     def setup(self, A, **kwargs):
10        LinearOperator.setup(self, A, **kwargs)
11
12        # Assembly step for additive Schwarz
13        if self.assemble:
14            self.Aih = A.dd.assemble(A.local, dim=2)
15        else:
16            self.Aih = A.local
17        # Add T_i on the interface for Robin
18        if self.Ti is not None:
19            if np.isscalar(self.Ti):
20                Ti = self.Ti * ssp.eye(A.dd.nG)
21            elif self.Ti.shape[1] == 1:
22                Ti = ssp.diags(self.Ti)
23            else:
24                Ti = self.Ti
25            if A.dd.nI:
26                RG = ssp.csc_matrix(
27                    (np.ones_like(A.dd.interface),
28                     (range(A.dd.nG), A.dd.interface)),
29                    shape=(A.dd.nG, A.dd.ni))  # R_Γ
30                Ti = RG.T.dot(Ti).dot(RG)
31            self.Aih += Ti
32        # Partition of unity for Neumann Neumann
33        if self.Di is not None:
34            self.D = self.partition_of_unity(A, self.Di)
35            D_inv = ssp.diags(1./self.D.clip(1e-12))
36            self.Aih = D_inv.dot(D_inv.dot(self.Aih.T).T)
37            # dense.dot(sparse) => (sparse.T.dot(dense.T)).T
38        # factorize Â_i
39        if self.local_solver is not None:
40            self.local_solver.setup(self.Aih)
41            DistMatrix.__init__(self, self.local_solver, A.dd)
42
43    @TimeIt()
44    def solve(self, b):
45        return DistMatrix.dot(self, b)
46
47    @staticmethod
48    def partition_of_unity(A, Di="matrix"):
```

```
        if Di == "multiplicity":
            mult = np.ones(A.dd.ni, dtype=np.int)
            for i in A.dd.neighbors.values():
                mult[i] += 1
            D = 1./mult
        elif Di == "matrix":
            diagA = A.local.diagonal()
            diagA_ = A.dd.assemble(diagA)
            D = diagA/diagA_
        elif Di == "boolean":
            D = A.dd.D
        else:
            raise TypeError((
                "Parameter Di in AbstractSchwarz"
                ".partition_of_unity should be 'multiplicity',"
                "'matrix' or 'boolean'. It is '{}'.").format(Di))
        return D
```

Listing 59: `AbstractSchwarz`.

Additive Schwarz (AS) preconditioner. The AS preconditioner

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

is a particular case of aS preconditioner where only the *assembly* step is performed.

```
class AdditiveSchwarz(AbstractSchwarz):

    defaults = AbstractSchwarz.defaults.copy()
    defaults.update({"assemble": True,
                     "Ti": None,
                     "Di": None})
```

Listing 60: `AdditiveSchwarz`.

Neumann-Neumann (NN) preconditioner. The NN preconditioner

$$\mathcal{M}_{NN} = \sum_{i=1}^{N} \mathcal{R}_i^T D_i \mathcal{A}_i^{\dagger} D_i \mathcal{R}_i,$$

has only the *partition of unity* step.

```python
class NeumannNeumann(AbstractSchwarz):

    defaults = AbstractSchwarz.defaults.copy()
    defaults.update({"assemble": False,
                     "Ti": None,
                     "Di": "matrix"})
```

Listing 61: NeumannNeumann.

Robin-Robin (RR) preconditioner. The Robin-Robin preconditioner

$$\mathcal{M}_{RR} = \sum_{i=1}^{N} \mathcal{R}_i^T (\mathcal{A}_i + \mathcal{T}_i)^{-1} \mathcal{R}_i,$$

has only the *addition* step.

```python
class RobinRobin(AbstractSchwarz):

    defaults = AbstractSchwarz.defaults.copy()
    defaults.update({"assemble": False,
                     "Ti": 1,
                     "Di": None})
```

Listing 62: RobinRobin

Testing the aS preconditioners. These three aS preconditioners (AS, NN and RR) are tested in a primal formulation ($\mathcal{A} = \mathcal{K}$) and on a primal formulation on the interface Γ ($\mathcal{A} = \mathcal{S}$).

```python
from ddmpy import *

K = DistMatrix.test()
dd = K.dd
b = K.dot(DistVector(np.ones((4, 1)), dd))
for mat in "K", "S":
    for MaS in AdditiveSchwarz, NeumannNeumann, RobinRobin:
        iterative_solver = ConjGrad(M=MaS)
        if mat == "K":
            solver = iterative_solver
        else:
            solver = DistSchurSolver(
                interface_solver=iterative_solver)
        solver.setup(K)
```

```
        x = solver.dot(b)
        if dd.rank == 0:
            print("{} / {}: {} iterations".format(
                MaS.__name__, mat,
                iterative_solver.n_iter))
```

```
mpirun -np 3 python testaS.py
```

```
AdditiveSchwarz / K: 5 iterations
NeumannNeumann / K: 3 iterations
RobinRobin / K: 4 iterations
AdditiveSchwarz / S: 3 iterations
NeumannNeumann / S: 2 iterations
RobinRobin / S: 3 iterations
```

Due to the very small size of the linear problem and the poor choice of a transmission condition in the RR preconditioner used in this example, this test should not be considered as a representative comparison of these three preconditioners, but merely as a demonstration of how they can be used.

4.3.12.2 Two-level aS preconditioners

These one-level aS preconditioners can be combined, either additively or using deflation, with a coarse solve to build a two-level aS preconditioner (see Section 2.7.3).

Coarse Solve. The `CoarseSolve` class implements the operator

$$\mathcal{M}_0 = V_0 \left(V_0^T \mathcal{A} V_0\right)^{-1} V_0^T$$

where the global coarse space V_0 is defined as a concatenation of local coarse spaces

$$V_0 = \begin{pmatrix} \mathcal{R}_1^T V_0^1 & \mathcal{R}_2^T V_0^2 & \cdots & \mathcal{R}_N^T V_0^N \end{pmatrix}.$$

The coarse matrix $\mathcal{A}_0 = V_0^T \mathcal{A} V_0$ is computed in parallel

$$\mathcal{A}_0 = V_0^T \mathcal{A} V_0 = V_0^T \left(\sum_{i=1}^{N} \mathcal{R}_i^T \mathcal{A}_i \mathcal{R}_i \right) V_0 = \sum_{i=1}^{N} \bar{V}_0^{iT} \mathcal{A}_i \bar{V}_0^i,$$
$$\text{where } \bar{V}_0^i = \mathcal{R}_i V_0 = \begin{pmatrix} \mathcal{R}_i \mathcal{R}_1^T V_0^1 & \mathcal{R}_i \mathcal{R}_2^T V_0^2 & \cdots & \mathcal{R}_i \mathcal{R}_N^T V_0^N \end{pmatrix}.$$

$\bar{V}_0^i$ can be built by assembling one subdomain's local coarse space (V_0^i) with the restriction of the coarse space of its neighbors to the common interface ($\mathcal{R}_i \mathcal{R}_j^T V_0^j$). Since $\mathcal{R}_i \mathcal{R}_j^T$ is zero if $\Gamma_i \cap \Gamma_j = \emptyset$, only neighbor-to-neighbor communications are needed to compute $\bar{V}_0^i$.

Each column in V_0 corresponds to a *coarse unknown*. The coarse matrix $\mathcal{A}_0$ is defined as a sum of local coarse matrices $\bar{V}_0^{iT} \mathcal{A}_i \bar{V}_0^i$. The local coarse matrix of subdomain Ω_i only

has nonzero elements on coarse unknowns from Ω_i or its neighbors. As a result, $\mathcal{A}_0$ can be implemented as a `DistMatrix` built from $\mathcal{A}$, but on another `DomainDecomposition` object `dd0`.

In the `setup()` method, first, a default coarse space is built as a partition-of-unity coarse space Sarkis (2003) (lines 8 to 13). Then, neighbor-to-neighbor communications are performed using the `neighborSendRecv()` function presented in Section 4.3.4 to compute $\bar{V}_0^i$ (`Wi`) from V_0^i (`Wi_`) and define the coarse domain decomposition `dd0` (lines 14 to 45). Then, the coarse matrix $\mathcal{A}_0$ is computed and factorized (lines 46 to 51).

The parameters of the `CoarseSolve` linear operator are:

- `Wi`, a basis of the local coarse space. If `None` (default), a partition-of-unity coarse space is chosen
- `global_solver`, the distributed solver used to factorize the coarse matrix $\mathcal{A}_0$. It can be for instance:
  - `Mumps` for a distributed factorization (see Section 5.4.2 for a presentation of this strategy implemented in the framework of the MaPHyS solver in Chapter 5)
  - `CentralizedSolver(local_solver=...`, root=0)= for a centralized factorization on one MPI process (see Section 5.4.3 for a generalization of this method using a sub-communicator of possibly more than one process for handling the coarse problem); any sparse solver can be used as a local solver.
  - `CentralizedSolver(local_solver=...`, root=None)= for a redundant factorization of the coarse matrix on each MPI process (see Section 5.4.5 for a generalization of this method)

The `solve()` method can be used to compute either $\mathcal{M}_0\ b$, or $\mathcal{P}_0\ b = \mathcal{M}_0\mathcal{A}\ b$ if the `project` argument is set to `True`.

```
1  class CoarseSolve(LinearOperator):
2
3      defaults = {"Wi": None,
4                  "global_solver": CentralizedSolver}
5
6      @TimeIt()
7      def setup(self, A, **kwargs):
8          LinearOperator.setup(self, A, **kwargs)
9          dd = A.dd
10         if self.Wi is None:
11             self.Wi = AbstractSchwarz.partition_of_unity(
12                 A, "matrix").reshape((-1, 1))
13         Wi = self.Wi # V_0^i
14         with TimeIt("MPI assemble"):
15             send = {j: (Wi.shape[1], # number of coarse vectors
16                         Wi[dd.neighbors[j], :], # coarse vectors
```

```
                        list(dd.neighbors)) # neighbors
                    for j in dd.neighbors}
            recv = dict(neighborSendRecv(send))
        # We add the subdomain to the dictionary
        recv[dd.rank] = (Wi.shape[1], # number of coarse vectors
                         Wi, # coarse vectors
                         list(dd.neighbors)) # neighbors
        # Now, for each subdomain in recv, we add the corresponding
        # vectors to Wi_ and we update neighbors0
        neighbors0 = dict()
        ni0 = sum(m[0] for m in recv.values()) # number of coarse unknowns
        Wi_ = np.zeros((A.dd.ni, ni0)) # V_0^i
        counter = 0
        for j in sorted(recv):
            n_j, Wj, neighbors_j = recv[j]
            if n_j == 0:
                continue
            if j == dd.rank:
                indices = slice(None)
            else:
                indices = dd.neighbors[j]
            Wi_[indices, counter:counter+n_j] = Wj
            for k in np.append(neighbors_j, j):
                if k != dd.rank:
                    neighbors0.setdefault(k, []).extend(
                        range(counter, counter+n_j))
            counter += n_j
        self.dd0 = DomainDecomposition(ni0, neighbors0, dd.comm)
        self.Wi_ = Wi_ # V_0^i
        with TimeIt("coarse computeA0"):
            self.AiWi_ = A.local.dot(Wi_)        # A_i V_0^i
            A0i = Wi_.T.dot(self.AiWi_)          # V_0^{iT} A_i V_0^i
            self.A0 = DistMatrix(A0i, self.dd0) # A_0 = V_0^T A V_0
        self.global_solver.setup(self.A0)
        self.A0_inv = self.global_solver         # A_0^{-1}

    @TimeIt()
    def solve(self, b, project=False):
        dd = b.dd
        dd0 = self.dd0
        Wi_ = self.Wi_ # V_0^i
        if project:    # P_0 b = M_0 A b
            b0i = self.AiWi_.T.dot(b.local)
        else:          # M_0 b
            b0i = Wi_.T.dot(dd.D*b.local)
```

```
62          b0 = DistVector(b0i, dd0, assemble=True)
63          x0 = self.A0_inv.dot(b0)
64          x = Wi_.dot(x0.local)
65          return DistVector(x, dd)
66
67      @TimeIt()
68      def project(self, b): # P_0 b = M_0 A b
69          return self.solve(b, project=True)
70
71      def __add__(self, other):
72          return AdditivePcd(B=other, C=self)
73
74      def __radd__(self, other):
75          return self + other
```

Listing 63: `CoarseSolve`.

Testing the `CoarseSolve` class. A coarse solve should give the $\mathcal{A}$-orthogonal projection of the solution onto the coarse space. In particular, if the solution is in the coarse space, a call to `CoarseSolve(K).dot(b)` should return the exact solution. If not specified, the local coarse space is computed from a partition of unity, and the constant vector is in the coarse space.

```
from ddmpy import *

K = DistMatrix.test()
dd = K.dd
b = K @ DistVector(np.ones((4, 1)), dd)

# x = M_0 A (1,...,1)^T = P_0 (1,...,1)^T = (1,...,1)^T
x = (CoarseSolve(K) @ b).centralize(root=0)
if dd.rank==0:
    print("x: {}".format(x[:, 0]))

u = DistVector(np.random.rand(4).reshape(4, 1),
               dd, assemble=True)     # u
u0 = CoarseSolve(K).project(u)        # u_0 = P_0 u
uL = u - u0                           # u_⊥ = u - u_0
scal = u0.T @ K @ uL                  # u_0^T A u_⊥ = 0
if dd.rank==0:
    print("u0.T @ K @ uL: {}".format(np.asscalar(scal)))
```

```
mpirun -np 3 python3 testCoarse.py
```

```
x: [ 1.  1.  1.  1.  1.  1.  1.]
u0.T @ K @ uL: 1.1657341758564144e-15
```

Additive two-level preconditioner. A two-level preconditioner can be obtained by adding a coarse solve to a one-level preconditioner $\mathcal{M}_2 = \mathcal{M}_0 + \mathcal{M}_1$ (see sections 2.7.3 and 3.4)

```python
class AdditivePcd(LinearOperator):

    defaults = {"B": AdditiveSchwarz,
                "C": CoarseSolve}

    @TimeIt()
    def setup(self, A, **kwargs):
        LinearOperator.setup(self, A, **kwargs)
        if not getattr(self.B, "setup_performed", True):
            self.B.setup(A)
        if not getattr(self.C, "setup_performed", True):
            self.C.setup(A)

    @TimeIt()
    def solve(self, b):
        return self.B.dot(b) + self.C.dot(b)
```

Listing 64: `AdditivePcd`.

Deflated preconditioner. Instead of adding the coarse correction, it is possible to use projection techniques to project the residual on the orthogonal of the coarse space at each iteration, defining $\mathcal{M}_D = \mathcal{M}_0 + (I - \mathcal{P}_0)\mathcal{M}_1(I - \mathcal{P}_0^T)$ where $\mathcal{P}_0 = \mathcal{M}_0\mathcal{A}$ is the $\mathcal{A}$-orthogonal projection onto V_0 (see sections 2.7.3 and 3.2). In practice, after an appropriate initial guess has been computed, the application of the preconditioner can be performed as $\mathcal{M}_D = (I - \mathcal{M}_0\mathcal{A})\mathcal{M}_1$.

```python
class DeflatedPcd(LinearOperator):

    defaults = {"M0": CoarseSolve,
                "M1": None}

    @TimeIt()
    def setup(self, A, **kwargs):
        LinearOperator.setup(self, A, **kwargs)
        if not getattr(self.M0, "setup_performed", True):
            self.M0.setup(A)
```

```
        if not getattr(self.M1, "setup_performed", True):
            self.M1.setup(A)

    @TimeIt()
    def solve(self, b):
        y = b if self.M1 is None else self.M1.dot(b) # M1 b
        return y - self.M0.project(y)                # (I - P0)M1 b

    @TimeIt()
    def x0(self, b): # Compute an appropriate initial guess x0 = M0 b
        return self.M0.dot(b)
```

Listing 65: `DeflatedPcd`

Testing the two-level preconditioners. The two-level preconditioners (additive or deflated) are tested alongside their one-level counterpart. Due to the size of the matrix, the number of iterations performed until convergence for this example is not at all representative of the capabilities of the tested methods.

```
from ddmpy import *

K = DistMatrix.test()
dd = K.dd
b = K @ DistVector(np.random.rand(4).reshape((4, 1)),
                   dd, assemble=True)

for mat in "K", "S":
    for MaS1 in AdditiveSchwarz, NeumannNeumann, RobinRobin:
        for MaS2, suffix in ((MaS1, "1"),
                             (MaS1() + CoarseSolve(), "2"),
                             (DeflatedPcd(M1=MaS1), "D")):
            iterative_solver = ConjGrad(M=MaS2)
            if mat == "K":
                solver = iterative_solver
            else:
                solver = DistSchurSolver(
                    interface_solver=iterative_solver)
            solver.setup(K)
            x = solver @ b
            if dd.rank == 0:
                print("{}_{} / {}: {} iterations".format(
                    MaS1.__name__, suffix, mat,
                    iterative_solver.n_iter))
```

```
1 mpirun -np 3 python3 test2level.py
```

```
AdditiveSchwarz_1 / K: 7 iterations
AdditiveSchwarz_2 / K: 7 iterations
AdditiveSchwarz_D / K: 4 iterations
NeumannNeumann_1 / K: 4 iterations
NeumannNeumann_2 / K: 5 iterations
NeumannNeumann_D / K: 3 iterations
RobinRobin_1 / K: 5 iterations
RobinRobin_2 / K: 7 iterations
RobinRobin_D / K: 7 iterations
AdditiveSchwarz_1 / S: 4 iterations
AdditiveSchwarz_2 / S: 4 iterations
AdditiveSchwarz_D / S: 1 iterations
NeumannNeumann_1 / S: 4 iterations
NeumannNeumann_2 / S: 4 iterations
NeumannNeumann_D / S: 1 iterations
RobinRobin_1 / S: 4 iterations
RobinRobin_2 / S: 4 iterations
RobinRobin_D / S: 4 iterations
```

4.3.12.3 GenEO coarse space for aS preconditioners

If $\mathcal{A}$ is SPD and $\mathcal{A}_i$ is SPSD, the coarse space can be built following the GenEO method introduced in Chapter 3 for our algebraic context. The coarse space is computed by solving generalized eigenproblems in each subdomain.

Solving Generalized Eigenproblems in python. In `scipy`, two methods are available to solve generalized eigenproblems $\mathcal{A}u = \lambda\mathcal{B}u$:

- the `scipy.linalg.eigh()` function for dense matrices, that uses a `Lapack` implementation Anderson et al. (1999).
- the `scipy.sparse.linalg.eigsh()` function for sparse matrices, that relies on the `ARPACK` solver Lehoucq et al. (1998).

The dense version computes all the eigenpairs. It is therefore quite costly. The sparse version computes a selection of eigenpairs using an iterative method. It is much faster but is not as robust: it does not converge for some matrices. As a result, the `ddmpy` package provides a `eigen()` function that calls the appropriate version depending on the type of the input matrices (lines 2-12 and 13-22 of Listing 66 for dense and sparse matrices, respectively). If the sparse version does not converge, it falls back to the dense version (lines 20-22). Its parameters are:

- `A` and `B`, representing the matrices $\mathcal{A}$ and $\mathcal{B}$.

- `B_I`, representing $\mathcal{B}^{-1}$ (optional).
- `n_v`, the number of eigenpairs to compute.
- `dense`, whether to force the dense version.
- `local_solver`, a `LinearOperator` to be used to factorize $\mathcal{B}$.

```python
def eigen(A, B, B_I=None, n_v=None, dense=True, local_solver=None):
    if dense or n_v is None:
        with TimeIt("eigen_dense"):
            if ssp.issparse(A) and ssp.issparse(B):
                w, v = la.eigh(A.A, B.A)
            else:
                w, v = la.eigh(A, B)
            if n_v is None:
                n_v = A.shape[0]
            else:
                n_v = min(A.shape[0], n_v)
            w, v = w[:n_v], v[:, :n_v]
    else:
        try:
            with TimeIt("eigen_sparse"):
                if B_I is None and local_solver is not None:
                    B_I = local_solver
                    B_I.setup(B)
                w, v = sla.eigsh(A, n_v, which='SM', M=B, Minv=B_I)
        except (sla.ArpackNoConvergence, sla.ArpackError) as err:
            print(err, '=> dense computation')
            w, v = eigen(A, B, B_I, n_v, dense=True)
    return w, v
```

Listing 66: eigen()= function for solving generalized eigenproblems.

Computing the GenEO coarse space. The GenEO coarse space associated with an aS preconditioner can be built using the `genEO_space()` function. It has the following arguments:

- `M1`, a one-level aS preconditioner.
- `alpha` and `beta`, representing the α and β parameters in the definition of the coarse space in Section 3.3.
- `n_v`, a maximum number of coarse vectors per subdomain.
- `local_solver`, a `LinearOperator` used to factorize local matrices.

```
def genEO_space(M1, alpha=10, beta=10, n_v=None):
    dense = not ssp.issparse(M1.A.local)
    if isinstance(M1, NeumannNeumann):
        NN = M1
        w1, v1 = np.zeros((0)), np.zeros((M1.Aih.shape[0], 0))
    else:
        NN = NeumannNeumann(M1.A, local_solver=None)
        w1, v1 = eigen(A=NN.Aih, B=M1.Aih, n_v=n_v, dense=dense)

    if alpha > 0:
        w1 = w1[w1*alpha < 1]
        v1 = v1[:, :w1.shape[0]]

    if isinstance(M1, AdditiveSchwarz):
        AS = M1
        w2, v2 = np.zeros((0)), np.zeros((M1.Aih.shape[0], 0))
        beta = alpha
    else:
        AS = AdditiveSchwarz(M1.A, local_solver=None)
        w2, v2 = eigen(A=M1.Aih, B=AS.Aih, n_v=n_v, dense=dense)
    if beta > 0:
        w2 = w2[w2*beta < 1]
        v2 = v2[:, :w2.shape[0]]

    w, v = np.hstack((w1, w2)), np.hstack((v1, v2))
    x = np.argsort(w)

    if n_v is not None and x.shape[0] > n_v:
        x = x[:n_v]

    return v[:, x]
```

Listing 67: `genEO_space()` function for computing the GenEO coarse space.

If the one-level aS preconditioner `M1` used in the method is AS or NN, one of the two eigenproblems from Section 3.3 becomes trivial and only the other eigenproblem needs to be solved.

The GenEO preconditioner. The basis of the GenEO coarse space returned by the `genEO_space()` function can be used to build a two-level aS preconditioner, using the `GenEOPcd` class. It has the following parameters:

- `M1`, the one-level aS preconditioner to use.
- `local_solver`, the solver to use for factorizing local matrices.

- `global_solver`, the solver to use for factorizing the (global) coarse problem.
- `deflated`, to toggle the use of a deflated coarse correction instead of an additive coarse correction.
- `alpha`, `beta` and `n_v`, the GenEO parameters used to build the coarse space.

```python
class GenEOPcd(LinearOperator):

    defaults = {"M1": AdditiveSchwarz,
                "local_solver": ScipyDirectSolver,
                "global_solver": CentralizedSolver,
                "deflated": True,
                "alpha": 10,
                "beta": 10,
                "n_v": 5}

    @TimeIt()
    def setup(self, A, **kwargs):
        LinearOperator.setup(self, A, **kwargs)
        if not getattr(self.M1, "setup_performed", True):
            self.M1.setup(A)

        with TimeIt("GenEO eigen") as t:
            self.Wi = genEO_space(self.M1, self.alpha, self.beta,
                                  self.n_v)
        A.dd.comm.barrier()
        self.M0 = CoarseSolve(A, Wi=self.Wi,
                              global_solver=self.global_solver)
        self.M0.parameters["Wi"] = "GenEO"
        self.parameters["t_GenEO_eigen"] = t.duration
        self.parameters["M0"] = self.M0
        if self.deflated:
            self.pcd = DeflatedPcd(A, M0=self.M0, M1=self.M1)
            self.x0 = self.pcd.x0
        else:
            self.pcd = AdditivePcd(A, B=self.M0, C=self.M1)

    @TimeIt()
    def solve(self, b):
        return self.pcd.solve(b)
```

Listing 68: `GenEOPcd`.

Testing the GenEO preconditioner The GenEO preconditioner is tested on the same distributed matrix as the preconditioners in the previous sections, both in a full primal formulation ($\mathcal{A} = \mathcal{K}$) and in a primal formulation on the interface ($\mathcal{A} = \mathcal{S}$). A GenEO coarse space is built for the AS, NN and RR, with a size of 1, 2 or 3 coarse unknowns per subdomain. The coarse correction is applied using deflation for all preconditioners, and additively for AS only.

```
from ddmpy import *

K = DistMatrix.test()
dd = K.dd
b = K @ DistVector(np.random.rand(4).reshape((4, 1)),
                   dd, assemble=True)

for mat in "K", "S":
    for MaS1, deflated in ((AdditiveSchwarz, False),
                           (AdditiveSchwarz, True),
                           (NeumannNeumann, True),
                           (RobinRobin, True)):
        for n_v in 1, 2, 3:
            M = GenEOPcd(M1=MaS1,
                         local_solver=ScipyDirectSolver,
                         global_solver=CentralizedSolver(
                             local_solver=Pinv,
                             root=0),
                         deflated=deflated,
                         n_v=n_v,
                         alpha=0,
                         beta=0)
            iterative_solver = ConjGrad(M=M)
            if mat == "K":
                solver = iterative_solver
            else:
                solver = DistSchurSolver(
                    interface_solver=iterative_solver)
            solver.setup(K)
            x = solver @ b
            if dd.rank == 0:
                print("GenEO({}, {}, {}) / {}: {} iterations".format(
                    MaS1.__name__, n_v,
                    "D" if deflated else "+", mat,
                    iterative_solver.n_iter))
```

```
mpirun -np 3 python3 testGenEO.py
```

```
GenEO(AdditiveSchwarz, 1, +) / K: 5 iterations
GenEO(AdditiveSchwarz, 2, +) / K: 6 iterations
GenEO(AdditiveSchwarz, 3, +) / K: 6 iterations
GenEO(AdditiveSchwarz, 1, D) / K: 4 iterations
GenEO(AdditiveSchwarz, 2, D) / K: 1 iterations
GenEO(AdditiveSchwarz, 3, D) / K: 0 iterations
GenEO(NeumannNeumann, 1, D) / K: 3 iterations
GenEO(NeumannNeumann, 2, D) / K: 1 iterations
GenEO(NeumannNeumann, 3, D) / K: 0 iterations
GenEO(RobinRobin, 1, D) / K: 3 iterations
GenEO(RobinRobin, 2, D) / K: 2 iterations
GenEO(RobinRobin, 3, D) / K: 0 iterations
GenEO(AdditiveSchwarz, 1, +) / S: 4 iterations
GenEO(AdditiveSchwarz, 2, +) / S: 4 iterations
GenEO(AdditiveSchwarz, 3, +) / S: 4 iterations
GenEO(AdditiveSchwarz, 1, D) / S: 2 iterations
GenEO(AdditiveSchwarz, 2, D) / S: 0 iterations
GenEO(AdditiveSchwarz, 3, D) / S: 0 iterations
GenEO(NeumannNeumann, 1, D) / S: 2 iterations
GenEO(NeumannNeumann, 2, D) / S: 0 iterations
GenEO(NeumannNeumann, 3, D) / S: 0 iterations
GenEO(RobinRobin, 1, D) / S: 1 iterations
GenEO(RobinRobin, 2, D) / S: 0 iterations
GenEO(RobinRobin, 3, D) / S: 0 iterations
```

4.4 Experimental study

4.4.1 Experimental setup

Similarly to the results obtained in the previous chapter (See Section 3.5.1), a weak scalability study is performed on problems similar to what was presented in (Spillane et al., 2014a). In the previous chapter, which essentially aimed at studying the numerical behavior of the proposed methods, the considered problems were of moderate size (at the exception of Section 3.5.5 which gave an insight on the parallel behavior). In the present chapter, as well as in the following chapter, the focus is essentially on the software design and parallel scalability of the methods and code; we therefore consider larger problems. They correspond to the *bâton* test case presented in more details in sections 3.5.1 and 5.5.2. A Darcy equation in a heterogeneous stratified medium of size $N \times 1 \times 1$ is solved using a DD of N subdomains. Each subdomain is a $1 \times 1 \times 1$ cube discretized with 30 Q1 elements in each direction. The subdomain matrices are generated using the `genfem` python module[5] developed for the purpose of this thesis. All the code needed to reproduce the experiments (including the batch job script) and generate the figures in the present section is available in the `.org` source of this document.

[5] `https://gitlab.inria.fr/solverstack/genfem`

On page 177 and 178, several DD preconditioners are compared. In the first figure, CG is applied on the matrix $\mathcal{K}$, whereas in the second figure, CG is applied on the Schur complement matrix $\mathcal{S}$.

Both figures have a grid layout with 3×8 plots: the rows correspond to various degrees of heterogeneity (K=1, K=1,000 and K=1,000,000) in the conductivity of the layers in the stratified medium (see Section 5.5.2). With $K = 1$, the problem is a homogeneous Poisson problem; increasing the heterogeneity parameter K increases the condition number of the matrix $\mathcal{K}$ and makes the linear system harder to solve.

The 8 columns correspond to the following preconditioners:

0 no preconditioner.

0_D deflation on a partition-of-unity coarse space (no additional local preconditioning).

AS_1 one-level AS preconditioner $\mathcal{M}_{AS}$ (no coarse correction).

AS_2 two-level AS preconditioner with an additive coarse correction with a partition-of-unity coarse space $\mathcal{M}_{AS,2}$.

AS_D deflated AS preconditioner with a partition-of-unity coarse space $\mathcal{M}_{AS,D}$.

AS_{GD3} deflated AS preconditioner $\mathcal{M}_{AS,D}$ with an adaptive (GenEO) coarse space ($n_v = 3$ vectors per subdomain).

NN_D deflated NN preconditioner with a partition-of-unity coarse space $\mathcal{M}_{NN,D}$ (when applied on the Schur complement matrix $\mathcal{S}$, the $NN_D/\mathcal{S}$ method is the BDD method (Mandel, 1993)).

NN_{GD3} deflated NN preconditioner $\mathcal{M}_{NN,D}$ with an adaptive (GenEO) coarse space ($n_v = 3$ vectors per subdomain).

Note that these notations for the solvers are specific to this section.

The 16 solvers (8 preconditioners applied on $\mathcal{K}$ or on $\mathcal{S}$) compared in this study are all built using the `ddmpy` module presented in Section 4.3.

Each individual plot represents the time needed to solve the bâton problem of size $(30N + 1) \times 30 \times 30$ on N CPU cores (the global domain is decomposed into N subdomains). Each node on the computing platform has 24 cores, and experiments were performed for N between 3 and 3,072. Each subdomain is handled by 1 MPI process on 1 CPU core. The stopping criterion of the conjugate gradient is chosen such that the normwise backward-error of the complete system (including interior unknowns) is under the prescribed tolerance $\frac{\|\mathcal{K}u-f\|_{\mathcal{K}}}{\|f\|_{\mathcal{K}}} \leq 10^{-5}$.

All dense computations are performed using the Intel MKL library 2017 through scipy and the `ScipyDirectSolver` class presented in Section 4.3.8.1, whereas sparse matrix factorizations use the Pastix 6.0.0 solver through the wrapper presented in Section 4.3.8.4.

The total time to solution is divided into 6 solver steps:

Schur Factorization **(blue)** if the PCG solver is applied on $\mathcal{S}$, the interior block $\mathcal{K}_{\mathcal{I}_i\mathcal{I}_i}$ is factorized in each subdomain to compute the local Schur complement matrix $\mathcal{S}_i$.

Local Pcd Setup **(dark green)** if a local preconditioner (AS or NN) is used, the local preconditioner matrix $\widehat{\mathcal{A}}_i$ is computed and factorized.

Coarse Eigen Solve **(green)** a generalized eigenproblem is solved for the adaptive coarse space.

Coarse Pcd Setup (light green) the coarse matrix is assembled and factorized redundantly on each CPU core.

Direct Solve (orange) the reduced right-hand side $\widetilde{f_\Gamma}$ is computed from the global right-hand side f and the interior solution $u_\mathcal{I}$ from the interface solution $u\Gamma$ if the PCG is applied on $\mathcal{S}$

Iterative Solve (red) PCG iterations.

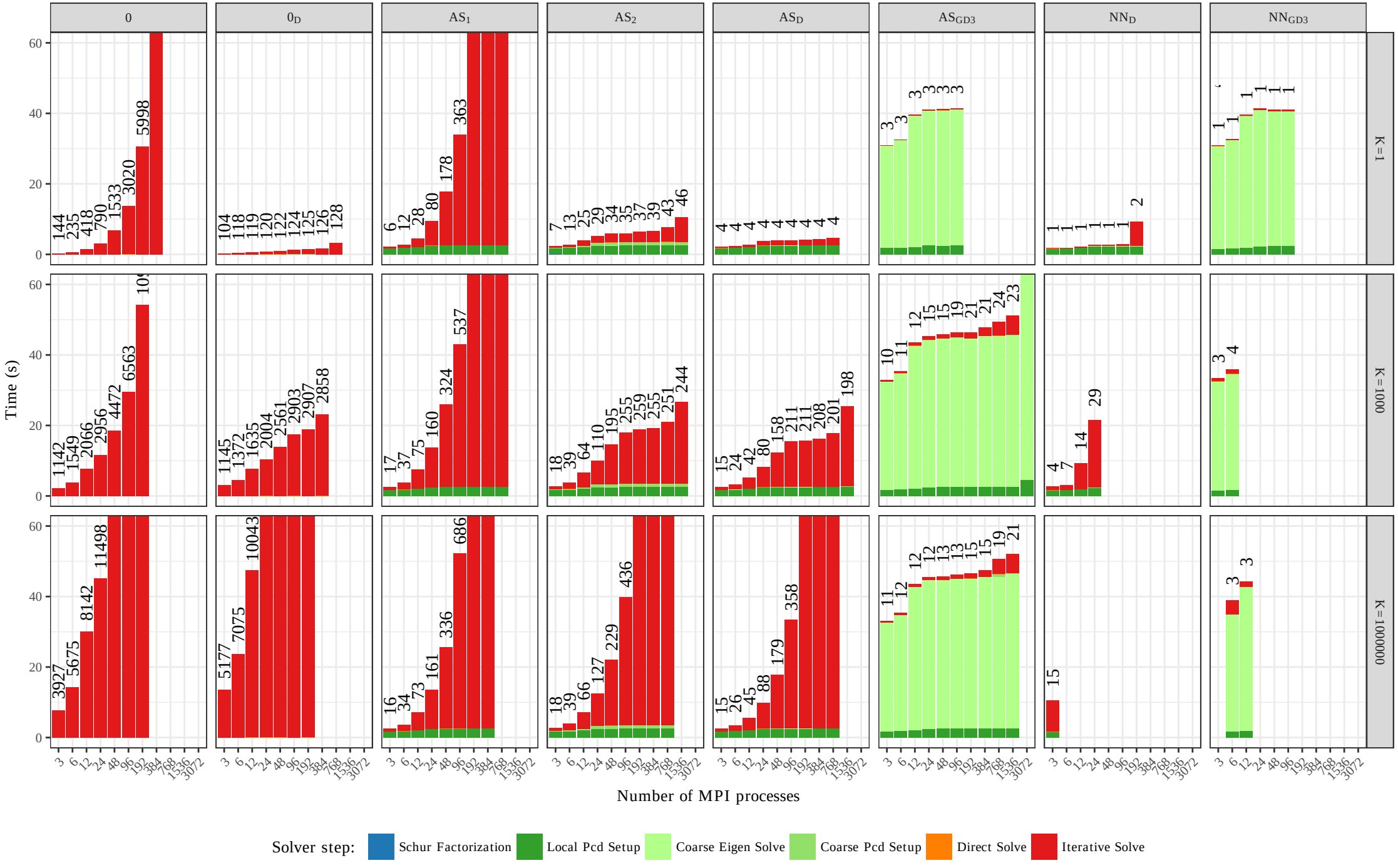
The number of iteration needed to reach convergence is indicated on top of each bar plot.

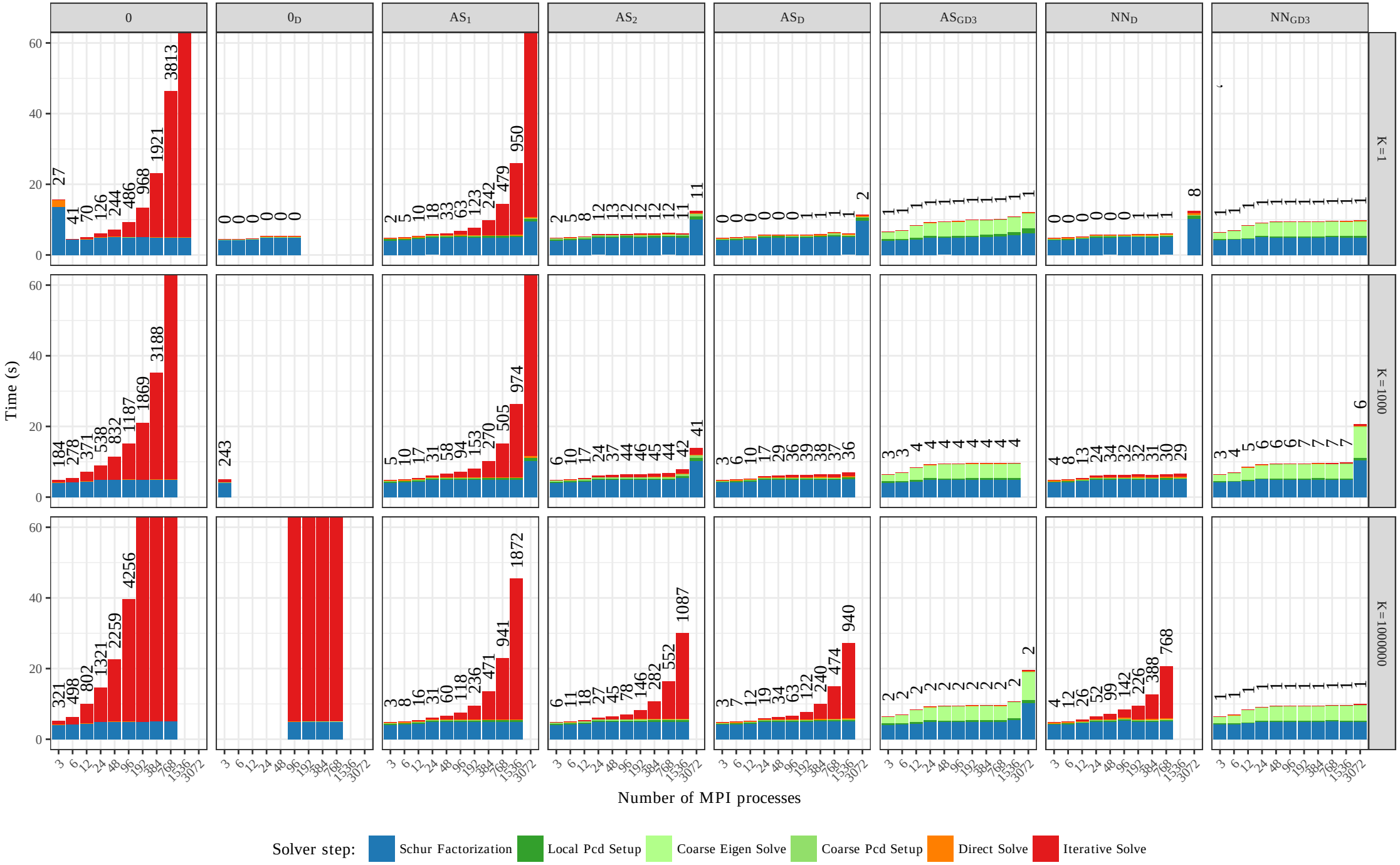
4.4.2 Numerical convergence

According to Chapter 3, (Corollary 2 and Theorem 2, as used in Section 3.5.3) the condition number of the methods presented here is bounded by $\kappa(\mathcal{MA}) \leq \mathcal{O}(\lambda_{n_v+1}^{-1})$ where λ_{n_v+1} is the smallest eigenvalue corresponding to an eigenvector not included in the coarse space. The matrix $\mathcal{A}_i^{(NN)}$ of the Neumann local problem in the generalized eigenproblem is singular, and $\lambda_1 = 0$. Without CSC ($n_v = 0$), the theoretical bound is infinite, meaning that the condition number is not bounded; this is in agreement with our experimental results where the number of iterations of the 0 and AS_1 methods increase proportionally to N.

Using a partition-of-unity coarse space (methods 0_D, AS_2, AS_D and NN_D) is equivalent to setting $n_v = 1$: the kernel of $\mathcal{A}_i^{(NN)}$ is the coarse space, and the bound becomes $\kappa(\mathcal{MA}) \leq \mathcal{O}(\lambda_2^{-1})$. Since the matrices in all the subdomains except the first and last one are the same, this bound is independent of N. However, it depends on the local matrix $\mathcal{A}_i$ (and therefore on the heterogeneity factor K), and on the local preconditioner. In our experiments, the number of iterations of these methods is indeed bounded independently of N (but the bound varies with the chosen method and on K) for $K = 1$ and $K = 1,000$ (for each method, the number of iterations is larger if $K = 1,000$ than if $K = 1$). For $K = 1,000,000$, the number of iterations seems to increase proportionally to N for these methods; this indicates that the bound on the number of iterations resulting from the bound on the condition number is too large to be effective in practice: when thousands of CG iterations are performed, especially when using deflation techniques, round-off errors accumulate and prevent the algorithm from converging due to loss of orthogonality (see, for instance, (Saad et al., 2000) for a description of this phenomenon and some remediation strategies that have not been implemented in `ddmpy`). This explains the corresponding missing bar plots in the figures.

The solution proposed in Chapter 3 is to include in the coarse space the eigenvectors corresponding to these very small eigenvalues. Choosing $n_v = 3$ eigenvectors (consistently


Step by step comparison of the solvers (on $\boldsymbol{K}$)
0
0_D
AS_1
AS_2
AS_D
AS_{GD3}
NN_D
NN_{GD3}
K = 1
K = 1000
K = 1000000
Time (s)
Number of MPI processes
Solver step:
Schur Factorization
Local Pcd Setup
Coarse Eigen Solve
Coarse Pcd Solve
Direct Solve
Iterative Solve
Weak scalability: constant subdomain size.


Step by step comparison of the solvers (on $\boldsymbol{S}$)
0
0_D
AS_1
AS_2
AS_D
AS_{GD3}
NN_D
NN_{GD3}
K = 1
K = 1000
K = 1000000
Time (s)
Number of MPI processes
Solver step:
Schur Factorization
Local Pcd Setup
Coarse Eigen Solve
Coarse Pcd Setup
Direct Solve
Iterative Solve
Weak scalability: constant subdomain size.

with the number of high-conductivity layers in the subdomains) is enough: the number of iterations for the AS_{GD3} and NN_GD3 methods are bounded by 24. If these methods are applied on $\mathcal{S}$, the number of iterations is bounded by 7.

4.4.3 Performance analysis

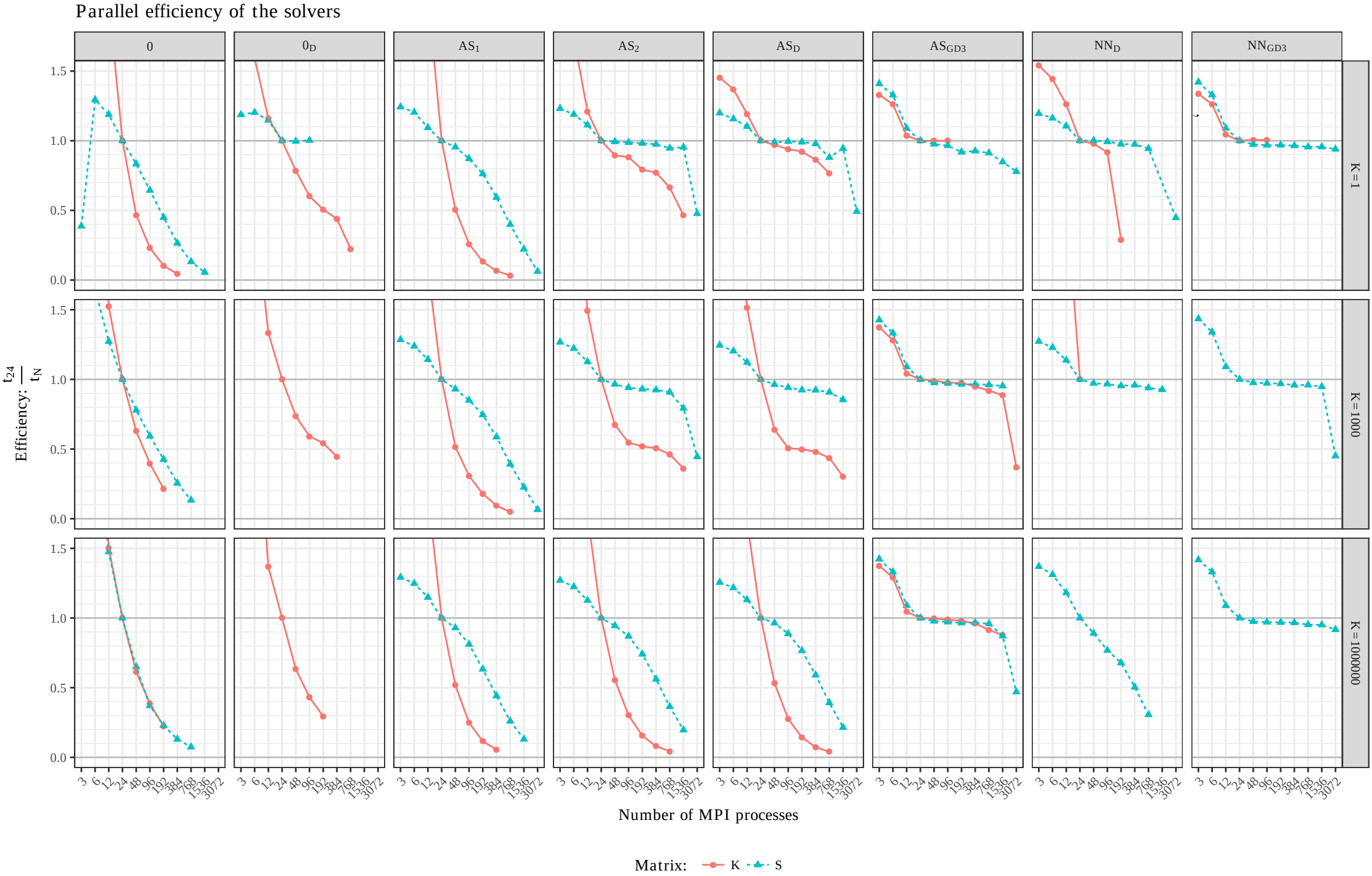
The data presented on pages 177 and 178 represent a set of 480 different configurations: 8 preconditioners applied on $\mathcal{K}$ and $\mathcal{S}$, for three different values of K and 10 values of N. For each of these configurations, the time needed to compute the Schur complement matrix and the local preconditioner, solve the eigenproblem and setup the coarse preconditioner, and perform the direct and iterative parts of the solve are displayed (the displayed times are the maxima among all MPI processes).

These experiments compare three different ways of hybridizing an iterative solver by introducing a direct solver: one can use a local (one-level aS) preconditioner (in dark green), a coarse global preconditioner (in medium green), or a Schur complement approach (in blue). In light green, the coarse space for the coarse preconditioner is computed using an eigensolver. Each of these (optional) operations add to the setup time of the solver, and can additionally increase the cost of each iteration. Our experimental results show that, for the most difficult problems ($N \gg 1$ and $K \gg 1$), the overhead of these additional operations is compensated by the reduction in the number of iterations discussed in the previous sections.

Our results show that in our homogeneous test case ($K = 1$), the best method up to $N = 768$ subdomains is a simple deflated CG (on $\mathcal{K}$) without a local preconditioner. In the other test cases, it is always beneficial to eliminate interior unknowns first using a Schur complement approach: replacing $\mathcal{K}$ by $\mathcal{S}$ in the iterative solver has several advantages. First, the linear system to be solved ($\mathcal{S}u_\Gamma = \widetilde{f}_\Gamma$ instead of $\mathcal{K}u = f$) is smaller and better conditioned, hence a reduction in the number of iterations. Furthermore, in addition to being of smaller size, the local interface matrices $\mathcal{S}_i$ are dense whereas the original local matrices $\mathcal{K}_i$ are sparse. As such, optimized solvers as provided by the Intel MKL library can be used for computing the local preconditioner and solving the eigenproblems. For instance, the cost of computing the GenEO coarse space drops from 40 s. to 4 s. when using a Schur complement approach. Since the overhead of factorizing the interior matrix $\mathcal{K}_{\mathcal{I}_i\mathcal{I}_i}$ and computing the local Schur complement matrix $\mathcal{S}_i$ is only 5 s. (thanks to using the optimized sparse direct solver Pastix that implements this operation), it is always faster to use a Schur complement approach when computing a spectral coarse space for this *bâton* test case.

A comparison of the NN and AS local preconditioners show that, on $\mathcal{K}$, the AS preconditioner is much more robust than NN, as the singularity of the local preconditioner matrix $\mathcal{K}_i^{(NN)}$ is not well handled by the Pastix solver. On $\mathcal{S}$, both preconditioners show a quite similar behavior. Both $AS_{GD3}/\mathcal{S}$ and $NN_{GD3}/\mathcal{S}$ methods are extremely scalable and very robust up to $N = 1,536$ or $3,072$ subdomains.

The parallel efficiency of the methods discussed above is presented in the figure page 180. We define the parallel weak efficiency as $\eta = \frac{t_{24}}{t_N}$: the basis for comparison is the total time to solution on one node, each node being composed of 24 CPU cores to each of which is associated an MPI process handling a subdomain.


Parallel efficiency of the solvers
0
0_D
AS_1
AS_2
AS_D
AS_{GD3}
NN_D
NN_{GD3}
K = 1
K = 1000
K = 1000000
Efficiency: $\frac{t_{24}}{t_N}$
0.0
0.5
1.0
1.5
3
6
12
24
48
96
192
384
768
1536
3072
Number of MPI processes
Matrix: K S
Weak scalability: constant subdomain size.

The adaptive CSC implemented in the AS_{GD3} and NN_{GD3} methods scales up to $N = 1,536$ subdomains with a very high parallel efficiency ($\geq 84\%$). At $N = 3,072$ subdomains, some solvers failed to converge ($AS_{GD3}/\mathcal{S}$ for $K = 1,000$ and $AS_{GD3}/\mathcal{K}$ for $K = 1,000$, for numerical reasons related to the deflation technique (Saad et al., 2000; Giraud et al., 2006). Furthermore, the parallel efficiency of some solvers drop suddenly to less than 50%, as for instance the $AS_{GD3}/\mathcal{S}$ solver for $K = 1,000$. To analyze where the bottleneck is, we look in the next section at a log of the `TimeIt` events in this run.

4.4.4 A posteriori analysis using the `TimeIt` object

The `TimeIt` *decorator* and *context manager*, introduced in Section 4.3.5, is used to track the time at which different functions and other code blocks in `ddmpy` are entered and exited. This information can be exploited after a problem has been solved to identify where the time was spent. In the experiments presented in this chapter, the first 2,000 TimeIt events of 10 processes (at most) were saved after each run for analysis. These limitations (only 2,000 events, and only 10 processes) were introduced in order not to overload the file system, as would be the case if 3,072 processes were to write simultaneously a few tens of thousands of lines each in a file.

Even with this limitation, the information of 10 processes is still too much to represent in a pleasant way: the processes are not synchronized and their respective events overlap with each others. For instance, the processes that handle subdomains on the boundary have a smaller interface than the other ones and fewer neighbors; they often perform some operations much faster than the other subdomains. For any event (let us take, for instance, the computation of the Schur complement using Pastix), we have the time at which each of the 10 processes entered and exited the event. Instead of displaying all this information, we only display the minimum, median and maximum of the starting and ending time of the event, as in figures 4.10 and 4.11 for the $NN_{GD3}/\mathcal{S}$ method on 1,536 and 3,072 cores, respectively, for $K = 1,000$.

The TimeIt events form a hierarchy, since a timed function can be called inside another one. At the lowest level (0), one has the main `DistSchurSolver` `setup()` and `solve()` methods. Each of them can call other timed functions or blocks that are of level 1, which can themselves call other functions of higher levels.

The comparison of the two `TimeIt` traces in figures 4.10 and 4.11 shows that the drop in efficiency is due to operations that take place during the setup of the `DistSchurSolver` object, between the setup of the local `SchurSolver` object and the setup of the iterative solver `ConjGrad`, and to operations that take place in the setup of the `GenEOPcd` preconditioner, after the eigenproblem is solved. Looking at the code in Listing 56, we can see that this corresponds to the instruction

```
1 self.S = DistMatrix(self.local_schur.S, A.dd.interface_dd())
```

In order to improve the scalability of the solvers in `ddmpy`, one should therefore focus on the operations taking place in this line, such as the `interface_dd()` method of the `DomainDecomposition` class presented in Listing 28. We did not pursue this time consuming optimization any further.

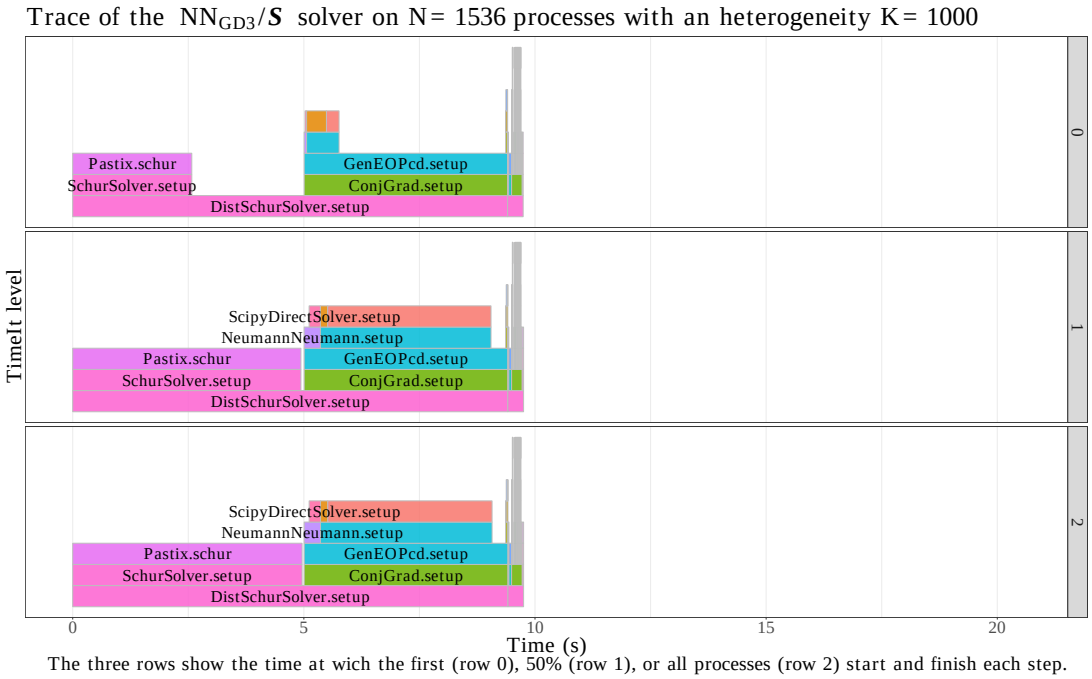


Figure 4.10: A trace of the $NN_{GD3}/\mathcal{S}$ solver on 1,536 processes.

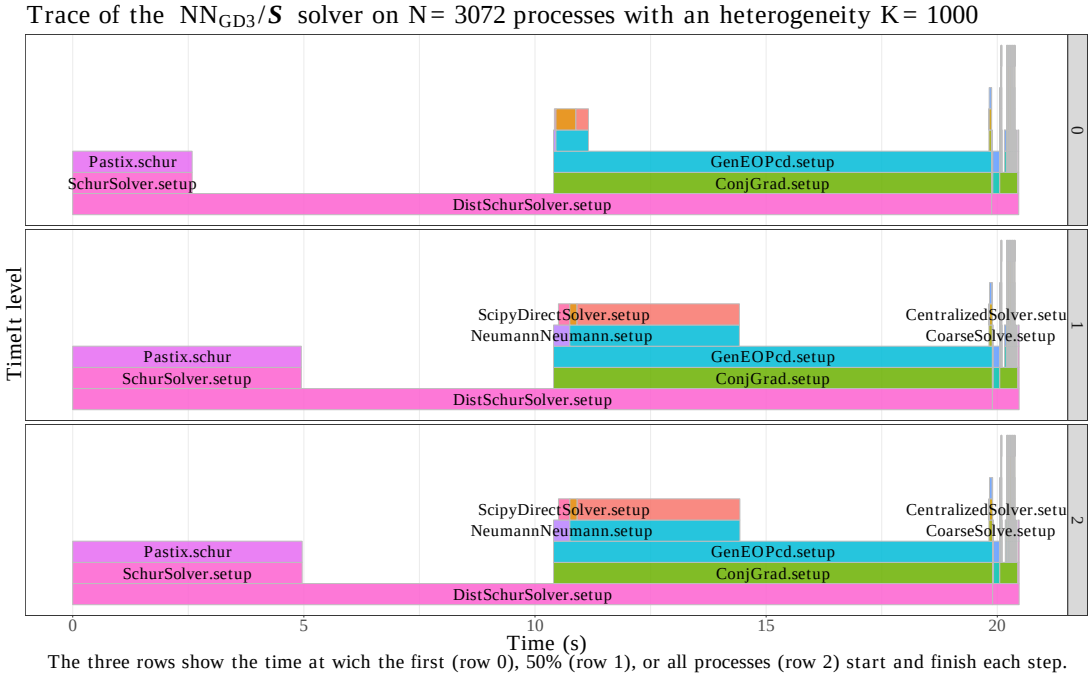


Figure 4.11: A trace of the $NN_{GD3}/\mathcal{S}$ solver on 3,072 processes.

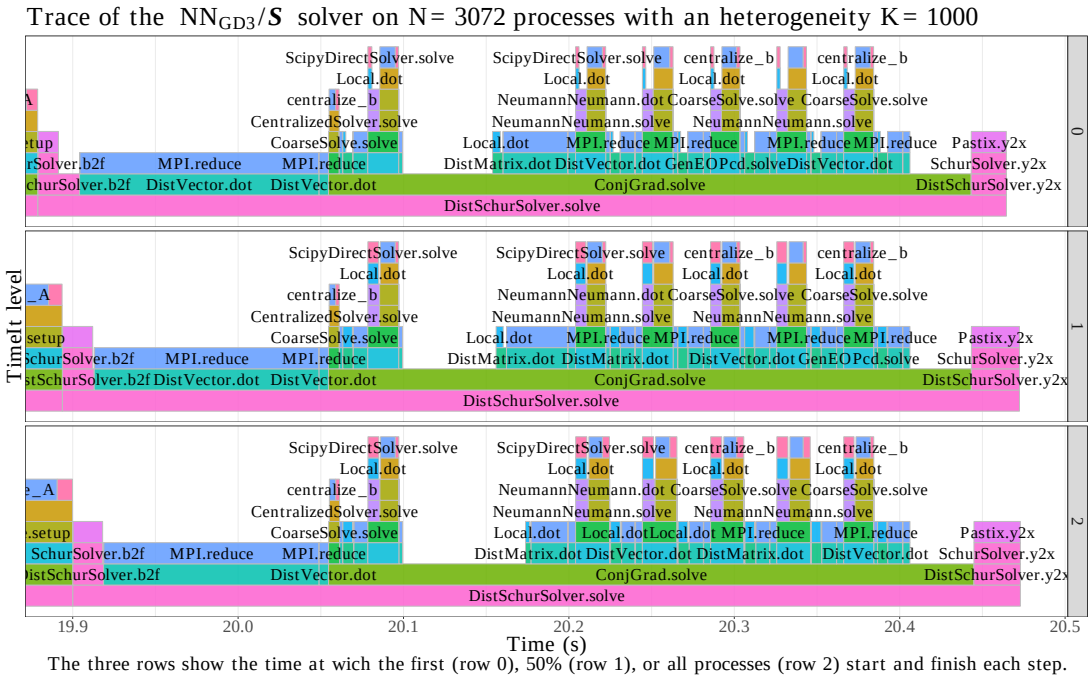


Figure 4.12: A zoom on the Solve step of Figure 4.11. The 6 CG iterations can be identified.

The solve step (6 s.) is much faster than the setup step (20 s.), and is condensed on the right of Figure 4.11. A zoom on this part is provided in Figure 4.12: the `DistSchurSolver.solve()` method (level 0) starts by computing the reduced right-hand side using the `DistSchurSolver.b2f()` method (level 1), the local `SchurSolver.b2f()` (level 2) and the `Pastix.b2f()` (level 3) methods. Then, the norm of the global right-hand side and of the reduced right-hand side are computed using two `DistVector.dot()` distributed scalar products. At this point, the interface system is solved using the `ConjGrad` solver. One can identify the CG initialization (first application of the preconditioner around 20.05 s.), the 5 subsequent complete iterations between 20.2 and 20.4 s. and the incomplete 6th iteration (no application of the preconditioner) before the `y2x()` method is called to compute the interior solution $u_{\mathcal{I}}$ from the interface solution u_Γ given by the CG.

One can zoom further on an individual iteration (Figure 4.13), and identify the steps of the CG algorithm (Listing 46): after the matrix-vector product (`DistMatrix.dot()`), two scalar products are computed (`DistVector.dot()`) separated by a (very fast) vector addition and subtraction. Then, the preconditioner is applied (`GenEOPcd.solve()`) and a third scalar product is performed. The two components of the two-level $\mathcal{M}_{NN,D}$ preconditioner can be identified: a local `NeumannNeumann` solve (using Scipy), followed by a coarse solve. Since the coarse problem is duplicated on all processes, one only need to centralize the coarse right-hand side (`centralize_b`) before computing the coarse correction (`ScipyDirectSolver.solve()`).

The `TimeIt` class is a very powerful tool for analyzing the performance of distributed applications in python. The ability to profile selected functions or code blocks using a

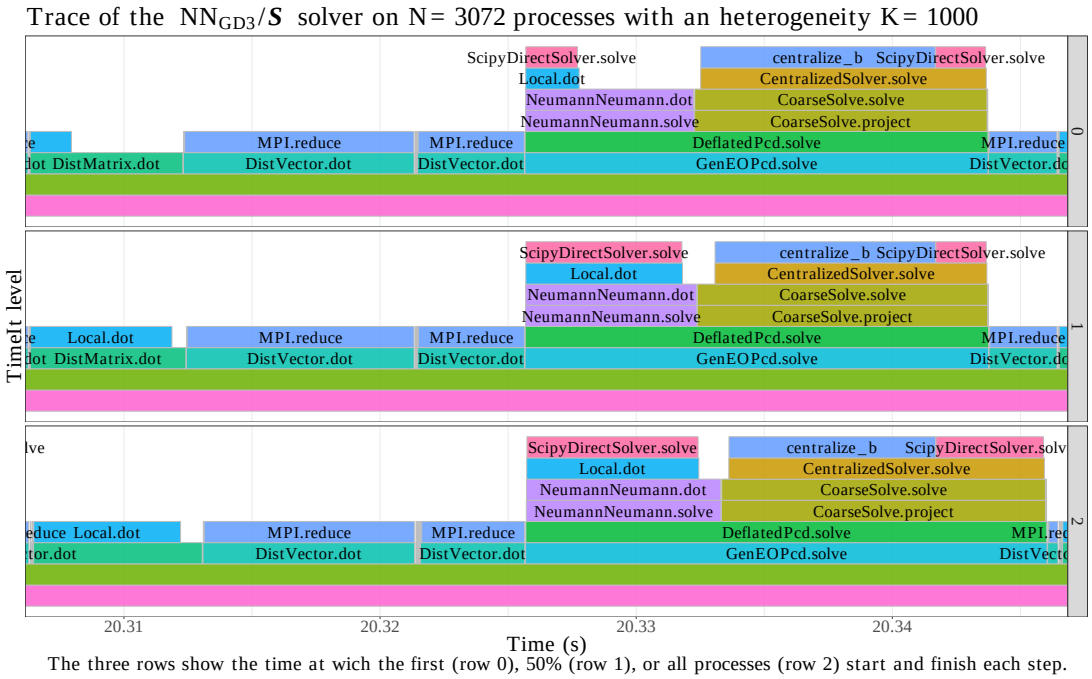


Figure 4.13: A zoom on one CG iteration of Figure 4.12.

decorator or a context manager makes it very easy to track the state of an application at the appropriate level, without a significant loss in performance in our case.

4.5 Limitations of the python language

Developing the `ddmpy` module, we encountered some difficulties related to the choice of the python language. First, it is not possible to run several threads concurrently in python. It is however possible to run external multithreaded code (such as Pastix for instance). This is due to a Global Interpretor Lock (GIL) that prevents threads from accessing python objects simultaneously. This limitation was not critical in our case since we decided to rely on a distributed paradigm.

The second difficulty we ran into was the lack of scalability of the import system in python. For each `import` instruction, python checks several different locations in the file system until it finds and loads the module. When many processes import the same modules simultaneously on a shared file system, they block each other and saturate the system. As a result, the standard python distributions cannot be used for large-scale distributed computing due to unpredictable and slow loading time. In some of our experiments, even when using only 2 nodes (48 CPU cores), the loading time of python was more than 10 times longer than the time needed to generate and solve the linear problem. Various solutions have been proposed to remedy this problem (Feng and Hand, 2016; Feng, 2018; Frings et al., 2013); we tested the spindle executable (LeGendre and Frings, 2018) but were not able to install it successfully on the targeted machine. We decided to use a modified python interpreter (Enkovaara and Louhivuori, 2017) that remove the import congestion by rewriting the `stdio.c` file so that only one process does the import and

broadcasts the module to the other processes using MPI. Since this `scalable-python` interpreter is based on `python2.7.13`, we had to rewrite `ddmpy` in python2 as it was originally written in python3. We were then able to perform the parallel experiments presented in the present chapter.

The third and last limitation of python for `ddmpy` is the fact that it is very cumbersome to use a solver written in python in a code written in another language. As a result, although it could be quite as performant as a compiled library, the `ddmpy` module cannot be readily used from a compiled application. This led us to initiate the development of a new library in C++ to keep the advantages of using a high-level language like python, while being easier to interface with compiled solvers and applications, as discussed in Section 6.3.

4.6 Conclusion

In this chapter, we presented a new domain decomposition (DD) toolbox in python. In Section 4.2, we demonstrated the relevance of python as a language for numerically prototyping HPC DD solvers: the overhead of using python instead of a compiled language such as C is on the order of 10^{-6} s. per program instruction in our experiments. It is therefore possible to write a competitive DD solver in python by using wrappers for calling third party HPC implementations for all computation-intensive operations. For instance, BLAS functions that operate on data arrays can be called through the `numpy` and `scipy` python modules. Using this approach, one single python instruction calls a function that perform many individual operations, and the cost of the overhead becomes negligible if the size of the data is large enough so that the time spent performing these individual operations is orders of magnitude larger than 10^{-6}.

A DD toolbox in python named `ddmpy` was implemented following this principle and presented in Section 4.3 using a literate programming approach. Wrappers for DD operations not included in `scipy`, such as computing a Schur complement matrix or performing a distributed conjugate gradient, are included in `ddmpy`. DD methods introduced in Chapter 2, such as a N-Lagrange solver or abstract Schwarz preconditioners for primal formulations, are available in `ddmpy`, as well as the two-levels methods introduced in Chapter 3.

The scalability of solvers built using the `ddmpy` toolbox was then assessed in Section 4.4: a linear system of 88.5M unknowns was solved in less than 10 s. on 128 computing nodes with 24 CPU cores each (3,072 cores in total). The two-level solvers that include the coarse space correction introduced in Chapter 3 showed a parallel efficiency of 84% on 1,536 cores. These experiments are fully reproducible as the code needed to launch the experiments, analyze the results and plot the figures is available in the source of this document.

We then shared in Section 4.5 some drawbacks we experienced due to choosing the python language for implementing the `ddmpy` toolbox. The lack of scalability of the import system of python on distributed computers forced us to install a modified interpreter and backport our code from python3 to python2 to run the parallel experiments presented in Section 4.4. Furthermore, the choice of python makes it inconvenient for compiled

applications to use `ddmpy`. This led us to implement one of the robust two-level methods in the MaPHyS HPC hybrid solver, in Fortran 90. Several parallelization strategies for handling the coarse space correction were implemented and are discussed in Chapter 5. A new implementation in modern C++ is underway (see Section 6.3 in the perspectives).

Chapter 5

Parallel design of coarse space correction for hybrid solvers

5.1 Introduction

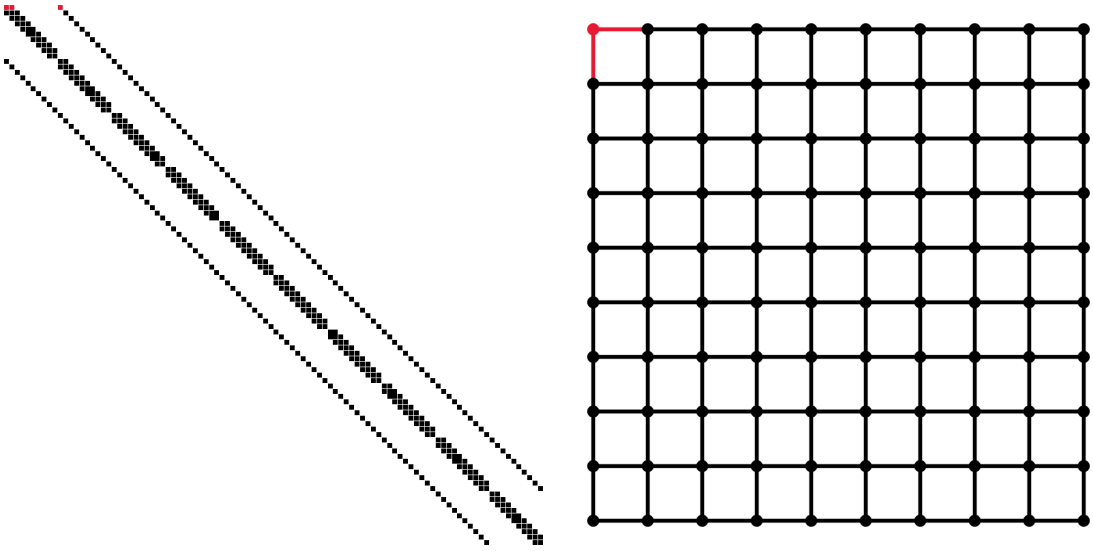
After a general presentation of domain decomposition methods (DDM) in Chapter 2, the numerical convergence of abstract Schwarz (aS) methods was studied in Chapter 3 and various DDM were implemented in the `ddmpy` toolbox presented in Chapter 4, including two-level aS methods. For those two-level aS methods, two strategies were devised for handling the coarse space correction (CSC) in a parallel context: either handling it with a fully distributed mode or by centralizing it. The goal of the current chapter is to further study the parallelization of the CSC. For that, we consider a fully-featured sparse hybrid solver, MaPHyS. Contrary to `ddmpy` which has been designed to have the flexibility of instantiating various DDM, MaPHyS implements one particular DDM, the Additive Schwarz on the Schur (AS/S) method. While MaPHyS was a one-level AS/S methods only, we extend it with a CSC mechanism and propose five strategies to handle the CSC in a parallel context.

The chapter is organized as follows. First, the baseline (corresponding to the state of the code before the present thesis) one-level MaPHyS solver is described in Section 5.2. A CSC extension, as introduced in Chapter 3, and various parallelization strategies for its implementation are presented in sections 5.3 and 5.4. The resulting code has been employed to give a flavor of the potential of two-level hybrid solvers in Chapter 3 (see Section 3.5.5). More comprehensive numerical experiments and a comparison of the strategies on different test cases are presented in Section 5.5.

5.2 The baseline MaPHyS sparse hybrid solver

5.2.1 The additive Schwarz on the Schur (AS/S) method

The MaPHyS solver is based on a primal formulation on the interface (Section 2.6.3) with an AS preconditioner (Section 3.4). This method, referred to as *Additive Schwarz on the Schur* or AS/S Carvalho et al. (2001b); Giraud et al. (2008), has already been introduced



(a) Example of a matrix $\mathcal{K}$ (b) Adjacency graph associated with the matrix

Figure 5.1: Example of a matrix and its adjacency graph.

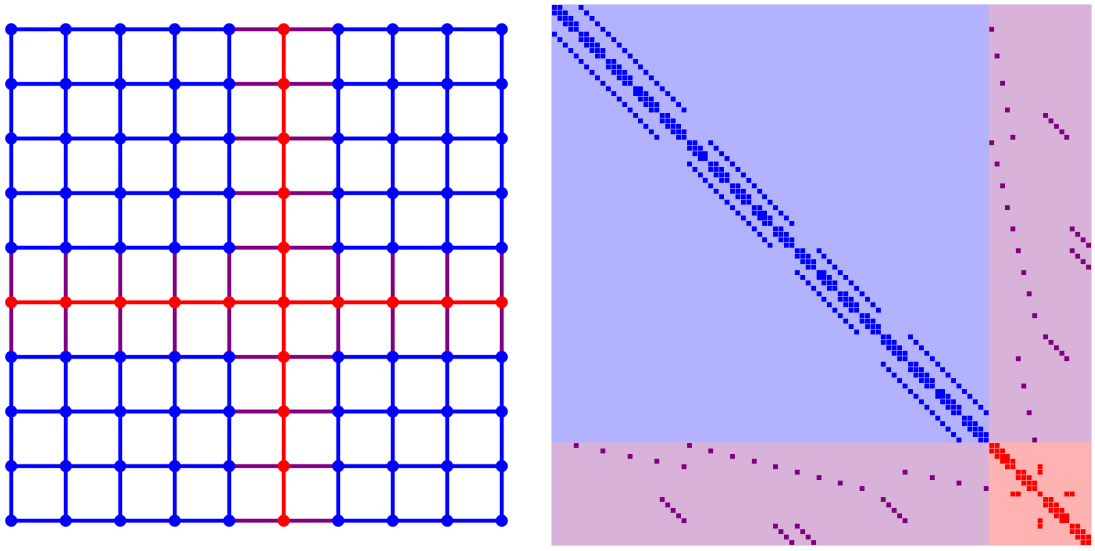
in previous chapters. In the present section, we present it again with a special focus on the parallel implementation.

As in previous chapters, we aim at solving a sparse linear system of the form $\mathcal{K}u = f$, where $\mathcal{K}$ is a large, sparse, symmetric positive definite (SPD) matrix. We note $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$ the adjacency graph associated with $\mathcal{K}$. In this graph, each vertex is associated with a row or column of the matrix $\mathcal{K}$ and it exists an edge between the vertices i and j if the entry $k_{i,j}$ is non zero. Figure 5.1 gives an example of matrix $\mathcal{K}$ (on the left) with its adjacency graph (on the right) where the red vertex corresponds to the first unknown of the linear system.

The governing idea behind the Schur complement approach is to split the unknowns into two categories: interior and interface vertices. We assume that the vertices of the graph $\mathcal{G}$ are partitioned into N disconnected subgraphs $\mathcal{I}_1, \ldots, \mathcal{I}_N$ separated by the global vertex separator Γ. We also decompose the vertex separator Γ into non-disjoint subsets Γ_i, where Γ_i is the set of vertices in Γ that disconnects $\mathcal{I}_i$ from other interior sets. Notice that this decomposition is not a partition as $\Gamma_i \cap \Gamma_j \neq \emptyset$ when the set of vertices in this intersection defines the separator of $\mathcal{I}_i$ and $\mathcal{I}_j$. By analogy with classical DDM in a finite element framework, $\Omega_i = \mathcal{I}_i \cup \Gamma_i$ is referred to as a subdomain with internal unknowns $\mathcal{I}_i$ and interface unknowns Γ_i. If we denote $\mathcal{I} = \cup\, \mathcal{I}_i$ and order vertices in $\mathcal{I}$ first, we obtain the following block reordered linear system

$$\begin{pmatrix} \mathcal{K}_{\mathcal{I}\mathcal{I}} & \mathcal{K}_{\mathcal{I}\Gamma} \\ \mathcal{K}_{\Gamma\mathcal{I}} & \mathcal{K}_{\Gamma\Gamma} \end{pmatrix} \begin{pmatrix} u_{\mathcal{I}} \\ u_{\Gamma} \end{pmatrix} = \begin{pmatrix} f_{\mathcal{I}} \\ f_{\Gamma} \end{pmatrix} \tag{5.1}$$

where u_Γ contains all unknowns associated with the separator and $u_\mathcal{I}$ contains the unknowns associated with the interiors. Figure 5.2 gives an example of graph/matrix problem partitioning into 4 parts, with interior vertices/unknowns in blue and interface vertices/unknowns in red.



(a) Partitioned adjacency graph associated with the matrix

(b) Reordered matrix $\mathcal{K}$ (interior unknowns are in blue and interface unknowns are in red)

Figure 5.2: Example of graph partitioning into 4 parts with its corresponding reordered matrix.

Eliminating $u_{\mathcal{I}}$ from the second block row (with a direct method in our case, see below) of Equation (5.1) leads to the reduced system

$$\mathcal{S} u_\Gamma = \widetilde{f}_\Gamma \tag{5.2}$$

where

$$\mathcal{S} = \mathcal{K}_{\Gamma\Gamma} - \mathcal{K}_{\Gamma\mathcal{I}} \mathcal{K}_{\mathcal{I}\mathcal{I}}^{-1} \mathcal{K}_{\mathcal{I}\Gamma} \quad \text{and} \quad \widetilde{f}_\Gamma = f_\Gamma - \mathcal{K}_{\Gamma\mathcal{I}} \mathcal{K}_{\mathcal{I}\mathcal{I}}^{-1} f_{\mathcal{I}}. \tag{5.3}$$

The matrix $\mathcal{S}$ is referred to as the *Schur complement matrix* (see sections 2.4 and 2.5.2) and inherits the symmetric positive definite property of $\mathcal{K}$. This reformulation leads to a general strategy for solving (5.1). A Preconditioned Conjugate Gradient (PCG) can be implemented to solve the reduced system (5.2). Once u_Γ has been computed the interior variables $u_{\mathcal{I}}$ can be computed with one additional solve for the interior unknowns via

$$u_{\mathcal{I}} = \mathcal{K}_{\mathcal{I}\mathcal{I}}^{-1} \left(f_{\mathcal{I}} - \mathcal{K}_{\mathcal{I}\Gamma} u_\Gamma \right). \tag{5.4}$$

Because a direct solver is considered for this last step one can notice that

$$\| \mathcal{K} u - f \| \approx \| \mathcal{S} u_\Gamma - \widetilde{f}_\Gamma \|;$$

we use therefore the following normwise backward error stopping criterion for the PCG iterations

$$\frac{\| \mathcal{S} u_\Gamma - \widetilde{f}_\Gamma \|}{\| f \|} \leq \varepsilon.$$

While the Schur complement system is significantly smaller and better conditioned than the original matrix $\mathcal{K}$, it is important to consider further preconditioning to accelerate

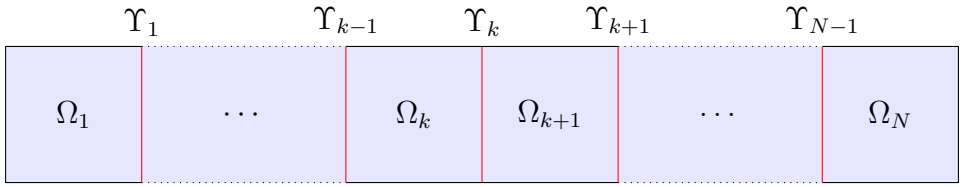


Figure 5.3: Planar graph partitioned into horizontal strips (1D decomposition). The interface between any two neighbors is $\Upsilon_k = \Omega_k \bigcap \Omega_{k+1}$.

the convergence of CG. The AS/S preconditioner in MaPHyS is built as follows. First, we define $\bar{\mathcal{S}}_i = \mathcal{R}_{\Gamma_i} \mathcal{S} \mathcal{R}_{\Gamma_i}^T$, where $\mathcal{R}_{\Gamma_i} : \Gamma \to \Gamma_i$ is the canonical point-wise restriction which maps full vectors defined on Γ into vectors defined on Γ_i. $\bar{\mathcal{S}}_i$ corresponds to the restriction of the Schur complement to the interface Γ_i of each subdomain. If $\mathcal{I}_i$ is a fully connected subgraph of $\mathcal{G}$, and if for each γ in Γ_i, there is an edge (γ, v) in $\mathcal{G}$ with v in $\mathcal{I}_i$, then the matrix $\bar{\mathcal{S}}_i$ is dense.

With these notations the algebraic Additive Schwarz preconditioner on the Schur complement system (AS/S) given by Equation (5.1) reads

$$\mathcal{M}_{AS/\mathcal{S}} = \sum_{i=1}^{N} \mathcal{R}_{\Gamma_i}^T \bar{\mathcal{S}}_i^{-1} \mathcal{R}_{\Gamma_i}. \tag{5.5}$$

We notice that this preconditioner has a form similar to the Neumann-Neumann preconditioner (Bourgat et al., 1989; De Roeck and Le Tallec, 1991), but in the SPD case $\mathcal{M}_{AS/\mathcal{S}}$ is always fully defined and SPD (as $\mathcal{S}$ is SPD (Carvalho et al., 2001b)); which is not always the case for Neumann-Neumann.

If we considered a planar graph partitioned into horizontal strips (1D decomposition) with $\Upsilon_k = \Omega_k \bigcap \Omega_{k+1}$ as depicted on Figure 5.3, the resulting Schur complement matrix has a block tridiagonal structure as depicted in Equation (5.6),

$$\mathcal{S} = \begin{pmatrix} \ddots & & & & \\ & \mathcal{S}_{k-1,k-1} & \mathcal{S}_{k-1,k} & & \\ & \mathcal{S}_{k,k-1} & \mathcal{S}_{k,k} & \mathcal{S}_{k,k+1} & \\ & & \mathcal{S}_{k+1,k} & \mathcal{S}_{k+1,k+1} & \\ & & & & \ddots \end{pmatrix}. \tag{5.6}$$

For that particular structure of $\mathcal{S}$, the submatrices in boxes correspond to $\bar{\mathcal{S}}_k$ and $\bar{\mathcal{S}}_{k+1}$ that are the restriction of the Schur complement $\mathcal{S}$ to the interface of Ω_k and Ω_{k+1}. Such diagonal blocks, which overlap with one another, 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 (e.g., (Cai and Sarkis, 1999)) for the solution of general sparse linear systems.

5.2.2 Parallelization strategy for distributed memory architectures

MaPHyS is based on an algebraic domain decomposition idea whose primary motivation is to naturally exploit a coarse grained parallelism between the computation performed on each sub-problem of the decomposition using MPI.

Based on the decomposition of $\mathcal{G}$ we can define a decomposition of the matrix $\mathcal{K}$ where each sub-matrix is associated with a subdomain and is allocated to one MPI process. The local interiors are disjoint and form a partition of the interior $\mathcal{I} = \sqcup \mathcal{I}_i$. Consequently the matrix $\mathcal{K}_{\mathcal{I}\mathcal{I}}$ associated with the interior unknowns has a block diagonal structure; each diagonal block $\mathcal{K}_{\mathcal{I}_i\mathcal{I}_i}$ corresponds to the set of internal unknowns of Ω_i.

Two subdomains Ω_i and Ω_j are defined as neighbors if their interfaces intercept with each other, that is $\Gamma_i \bigcap \Gamma_j \neq \emptyset$. The non disjoint union of the subdomain boundaries form the overall interface $\Gamma = \cup \Gamma_i$. This implies that a special attention has to be paid for the partitioning of $\mathcal{K}_{\Gamma\Gamma}$ as its entries are shared between different processes. In that respect the matrix entries of $\mathcal{K}_{\Gamma\Gamma}$ must be weighted so that the sum of the coefficients on the local interface submatrices are equal to one. For that, we introduce the *weighted local interface* matrix $\mathcal{K}^w_{\Gamma_i\Gamma_i}$ that satisfies $\mathcal{K}_{\Gamma\Gamma} = \sum_{i=1}^{N} \mathcal{R}^T_{\Gamma_i} \mathcal{K}^w_{\Gamma_i\Gamma_i} \mathcal{R}_{\Gamma_i}$, where we recall that $\mathcal{R}_{\Gamma_i} : \Gamma \to \Gamma_i$ is the canonical point-wise restriction which maps full vectors defined on Γ into vectors defined on Γ_i. In matrix terms, a subdomain Ω_i may then be represented by the *local matrix* $\mathcal{K}_i$ defined by

$$\mathcal{K}_i = \begin{pmatrix} \mathcal{K}_{\mathcal{I}_i\mathcal{I}_i} & \mathcal{K}_{\mathcal{I}_i\Gamma_i} \\ \mathcal{K}_{\Gamma_i\mathcal{I}_i} & \mathcal{K}^w_{\Gamma_i\Gamma_i} \end{pmatrix}. \tag{5.7}$$

The *weighted local interface* matrix $\mathcal{K}^w_{\Gamma_i\Gamma_i}$ is an algebraic equivalent of the interface block of the local Neumann matrix $\mathcal{K}_{\Gamma_i\Gamma_i}$ (see Section 2.3.2): even though it is not possible, from $\mathcal{K}$, to obtain the true local interface matrix $\mathcal{K}_{\Gamma_i\Gamma_i}$, it is not needed to build an AS preconditioner. Any set of matrices that fulfill the sum condition $\mathcal{K}_{\Gamma\Gamma} = \sum_{i=1}^{N} \mathcal{R}^T_{\Gamma_i} \mathcal{K}^w_{\Gamma_i\Gamma_i} \mathcal{R}_{\Gamma_i}$ can be chosen. The global Schur complement matrix $\mathcal{S}$ from (5.2) can then be written as the sum of elementary matrices

$$\mathcal{S} = \sum_{i=1}^{N} \mathcal{R}^T_{\Gamma_i} \mathcal{S}_i \mathcal{R}_{\Gamma_i} \tag{5.8}$$

where

$$\mathcal{S}_i = \mathcal{K}^w_{\Gamma_i\Gamma_i} - \mathcal{K}_{\Gamma_i\mathcal{I}_i} \mathcal{K}^{-1}_{\mathcal{I}_i\mathcal{I}_i} \mathcal{K}_{\mathcal{I}_i\Gamma_i} \tag{5.9}$$

is the *local Schur complement* associated with subdomain Ω_i. This local expression allows for computing local Schur complements independently from each other.

The $\bar{\mathcal{S}}_i$ matrices, involved in the definition of $\mathcal{M}_{AS/\mathcal{S}}$, are the restriction of the global Schur complement to Γ_i and can actually be built from this data distribution of the $\mathcal{S}_i$ matrices. To illustrate this construction, let us consider a subdomain Ω_i with four neighbors and $\Gamma_i = E_m \cup E_g \cup E_k \cup E_m$ the union of the intersections of the boundary of Ω_i with each of its neighbors (assuming there is no cross-point, i.e., interface variables shared by more than two subdomains) as depicted in Figure 5.4. The local Schur complement

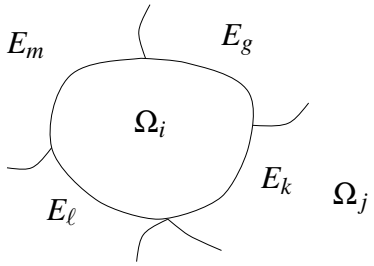


Figure 5.4: A given subdomain Ω_i with four neighbors

matrix associated with Ω_i has the following 4×4 block structure

$$\mathcal{S}_i = \begin{pmatrix} \mathcal{S}_{m,m}^{(i)} & \mathcal{S}_{m,g} & \mathcal{S}_{m,k} & \mathcal{S}_{m,\ell} \\ \mathcal{S}_{g,m} & \mathcal{S}_{g,g}^{(i)} & \mathcal{S}_{g,k} & \mathcal{S}_{g,\ell} \\ \mathcal{S}_{k,m} & \mathcal{S}_{k,g} & \mathcal{S}_{k,k}^{(i)} & \mathcal{S}_{k,\ell} \\ \mathcal{S}_{\ell,m} & \mathcal{S}_{\ell,g} & \mathcal{S}_{\ell,k} & \mathcal{S}_{\ell,\ell}^{(i)} \end{pmatrix} \tag{5.10}$$

where each block row or column is associated with an edge E_j, $j \in \{m, g, k, \ell\}$.

The matrix $\bar{\mathcal{S}}_i$ can be built from the local Schur complement $\mathcal{S}_i$ by collecting and summing (i.e., assembling in a finite element sense) its diagonal blocks thanks to a few neighbor-to-neighbor communications. For instance, the diagonal blocks of $\bar{\mathcal{S}}_i$ associated with the shared interface $E_k = \Gamma_i \cap \Gamma_j$ between Ω_i and Ω_j 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

$$\bar{\mathcal{S}}_i = \begin{pmatrix} \mathcal{S}_{m,m} & \mathcal{S}_{m,g} & \mathcal{S}_{m,k} & \mathcal{S}_{m,\ell} \\ \mathcal{S}_{g,m} & \mathcal{S}_{g,g} & \mathcal{S}_{g,k} & \mathcal{S}_{g,\ell} \\ \mathcal{S}_{k,m} & \mathcal{S}_{k,g} & \mathcal{S}_{k,k} & \mathcal{S}_{k,\ell} \\ \mathcal{S}_{\ell,m} & \mathcal{S}_{\ell,g} & \mathcal{S}_{\ell,k} & \mathcal{S}_{\ell,\ell} \end{pmatrix}.$$

At each PCG iteration, one needs to apply the preconditioner $\mathcal{M}_{AS/\mathcal{S}}$ to a vector r in order to compute $z = \mathcal{M}_{AS/\mathcal{S}}\, r$. The vectors r and z are handled through their restrictions $r_i = \mathcal{R}_{\Gamma_i} r$ and $z_i = \mathcal{R}_{\Gamma_i} z$ inside each subdomain. The preconditioner is applied in parallel as follows

$$z_i = \mathcal{R}_{\Gamma_i} \sum_{j=1}^{N} \mathcal{R}_{\Gamma_j}^T \bar{\mathcal{S}}_j^{-1} \mathcal{R}_{\Gamma_j} r = \sum_{j=1}^{N} \mathcal{R}_{\Gamma_i} \mathcal{R}_{\Gamma_j}^T \bar{\mathcal{S}}_j^{-1} r_j = \bar{\mathcal{S}}_i^{-1} r_i + \sum_{j \in \mathcal{N}(i)} \mathcal{R}_{\Gamma_i} \mathcal{R}_{\Gamma_j}^T \bar{\mathcal{S}}_j^{-1} r_j, \tag{5.11}$$

where $\mathcal{N}(i) = \{j \neq i,\ \Gamma_i \cap \Gamma_j \neq 0\}$ is the set of neighbors of subdomain Ω_i. After solving $z_i' = \bar{\mathcal{S}}_i^{-1} r_i$, the subdomain communicates with each of its neighbors to share the restriction of z_i' to their common interface and compute $z_i = z_i' + \sum_{j \in \mathcal{N}(i)} \mathcal{R}_{\Gamma_i} \mathcal{R}_{\Gamma_j}^T z_j'$.

After convergence of the PCG, the solution in the interiors can be computed in parallel using (5.4). The solution in $\mathcal{I}_i$ is

$$u_{\mathcal{I}_i} = \mathcal{K}_{\mathcal{I}_i \mathcal{I}_i}^{-1} \left(f_{\mathcal{I}_i} - \mathcal{K}_{\mathcal{I}_i \Gamma_i} \mathcal{R}_{\Gamma_i} u_\Gamma \right). \tag{5.12}$$

Algorithm 1 summarizes how the classical parallel implementation of MaPHyS can be decomposed into four main phases:

Algorithm 1: `MaPHyS algorithm`

1 *partitioning step*
2 *factorization of the interiors*
3 *setup of the preconditioner*
4 *solve step*

- (1) the *partitioning step*, consisting of partitioning the adjacency graph $\mathcal{G}$ of $\mathcal{K}$ into several subdomains and distributing the $\mathcal{K}_i$ to different cores. This step is in practice often performed by the application, whose partitioning must match the hypothesis in Equation (3.9) on page 63;

- (2) the *factorization of the interiors* and the computation of the local Schur complement $\mathcal{S}_i$ using $\mathcal{K}_i$. This step is performed independently by each MPI process and is common whether or not the CSC mechanism is applied and is thus not described further;

- (3) the *setup of the preconditioner* by assembling diagonal blocks of $\mathcal{S}_i$ via a few neighbor to neighbor communications and factorization of this one. In the one-level baseline version of MaPHyS, this step corresponds to Algorithm 2;

Algorithm 2: `Baseline one-level` *`setup of the preconditioner`* `(baseline step (3) of Algorithm 1)`

1 Compute $\bar{\mathcal{S}}_i$ from $\mathcal{S}_i$ by assembling diagonal blocks with neighbor to neighbor communications
2 Compute $\bar{\mathcal{S}}_i^{-1}$ (factorize $\bar{\mathcal{S}}_i$)

- (4) the *solve step*, where (4a) a parallel preconditioned Krylov method is performed on the reduced system (Equation (5.2)) to compute x_{Γ_i}, followed by (4b) independent back solves on the interiors to compute each $x_{\mathcal{I}_i}$ (Equation (5.12)). For SPD systems, the Krylov method of choice is the PCG. A PCG iteration is composed of a matrix-vector product $\mathcal{S}x$, an application of the preconditioner $\mathcal{M}_{AS/\mathcal{S}}x$, two dot products (plus one for the stopping criterion) and several vector additions, subtractions and scalar multiplications. The application of the one-level baseline preconditioner (Equation (5.11)) during step (4a) is described in Algorithm 3.

Note that subdomain Ω_j is handled by MPI process $j-1$: as explained in Section 4.3.6, index 0 is reserved for the coarse space so that subdomains are numbered from 1 to N whereas MPI ranks are numbered from 0 to $N-1$.

When the CSC is turned on, the setup (Algorithm 2) and application (Algorithm 3) of the preconditioner are enhanced to compute it and apply it, respectively, as further discussed below.

Algorithm 3: `Baseline one-level` *`application of the preconditioner`* `(during baseline step (4a) of Algorithm 1)`

1 Solve $z_i' = \bar{\mathcal{S}}_i^{-1} r_i$
2 **for** $j \in \mathcal{N}(i)$ **do**
3 | Send $\mathcal{R}_{\Gamma_j} \mathcal{R}_{\Gamma_i}^T z_i'$ to process $j-1$ and receive $\mathcal{R}_{\Gamma_i} \mathcal{R}_{\Gamma_j}^T z_j'$
4 **end**
5 Compute $z_i = z_i' + \sum_{j \in \mathcal{N}(i)} \mathcal{R}_{\Gamma_i} \mathcal{R}_{\Gamma_j}^T z_j'$

5.3 Design of a coarse space correction in MaPHyS

The goal of the CSC mechanism is to improve the numerical quality of the preconditioner to reduce the number of iterations, by controlling the condition number of the preconditioned system. A CSC is defined by its coarse space V_0, and the way it is combined with the first-level preconditioner that it improves. Within a purely algebraic solver, the construction of the solver can only rely on the information provided by the application, which is $\mathcal{K}$ and f. However, in MaPHyS, it is possible to provide the matrix $\mathcal{K}$ in a distributed input mode through the local matrices $\mathcal{K}_i$. If these local matrices are symmetric positive semi-definite (SPSD), we can add a second level of preconditioning such that the condition number and the number of iterations to reach convergence can be bounded, as proved in Chapter 3 following a methodology closely related to the GenEO technique introduced in (Spillane et al., 2014b).

For the purpose of the study conducted in this chapter, we have incorporated such a CSC to the baseline (one-level) version of MaPHyS as follows. First (Section 5.3.1), during the *setup of the preconditioner*, a *local coarse space* V_0^i is computed in each subdomain Ω_i; then, still during the *setup of the preconditioner*, a *coarse matrix* $\mathcal{S}_0$ computed from $\mathcal{S}$ and V_0^i is computed and factorized. Second (Section 5.3.2), each application of the $\mathcal{M}_{AS/\mathcal{S}}$ preconditioner is modified to include a *coarse solve*, leading to the *two-level AS* preconditioner for the Schur complement problem denoted by $\mathcal{M}_{AS,2/\mathcal{S}}$. The *setup* and *application* of the two-level preconditioner are detailed in the following subsections.

5.3.1 Setup of the two-level preconditioner $\mathcal{M}_{AS,2/\mathcal{S}}$

The *setup of the preconditioner* (step (3) of Algorithm 1) is performed as detailed in Algorithm 4 in the case of the two-level preconditioner $\mathcal{M}_{AS,2/\mathcal{S}}$. Additionally to the one-level setup (Algorithm 2 as well as steps 1-2 of Algorithm 4), the CSC is set up as follows (steps 3-10 of Algorithm 4). In each subdomain Ω_i, the following generalized eigenproblem is solved to compute the n_v smallest eigenvalues and their corresponding eigenvectors, thus including the kernel of $\mathcal{S}_i$ if $n_v \geq \text{rank}(\ker(\mathcal{S}_i))$

$$D_i^{-1} \mathcal{S}_i D_i^{-1} \; p_k^i \;=\; \lambda_k^i \; \bar{\mathcal{S}}_i \; p_k^i,$$

where D_i is a *partition of unity*, such that $\sum_{i=1}^N \mathcal{R}_{\Gamma_i}^T D_i \mathcal{R}_{\Gamma_i} = I$, where I the identity matrix (step 3). The number n_v of eigenvectors per subdomain can be chosen a priori or adapted depending on a convergence estimate as shown in Section 3.5.3. The local

coarse space basis can then be defined from these eigenvectors (step 4); in a matrix form it writes

$$V_0^i = \left[p_1^i \; p_2^i \; \cdots \; p_{n_v}^i\right],$$

and the global coarse space basis can be formally defined as

$$V_0 = \left[(\mathcal{R}_{\Gamma_1}^T V_0^1) \; (\mathcal{R}_{\Gamma_2}^T V_0^2) \; \cdots \; (\mathcal{R}_{\Gamma_N}^T V_0^N)\right].$$

In practice, V_0 is never explicitly formed, and no communication is required for this first step. Solving the eigenproblems may take a lot of time, but it is purely local and consequently fully scalable.

The two-level preconditioner we have incorporated into MaPHyS for illustrating the experimental study conducted in this chapter is then defined as

$$\mathcal{M}_{AS,2/\mathcal{S}} = \mathcal{M}_{AS/\mathcal{S}} + V_0 \left(V_0^T \mathcal{S} V_0\right)^{-1} V_0^T = \mathcal{M}_{AS/\mathcal{S}} + \mathcal{M}_0, \tag{5.13}$$

where $\mathcal{M}_0 = V_0 \; \mathcal{S}_0^{-1} \; V_0^T$ and $\mathcal{S}_0 = V_0^T \mathcal{S} V_0$. The coarse matrix $\mathcal{S}_0$ can be computed in parallel using Equation (5.8):

$$\mathcal{S}_0 = V_0^T \mathcal{S} V_0 = V_0^T \left(\sum_{i=1}^{N} \mathcal{R}_{\Gamma_i}^T \mathcal{S}_i \mathcal{R}_{\Gamma_i}\right) V_0 = \sum_{i=1}^{N} \bar{V}_0^{iT} \mathcal{S}_i \bar{V}_0^i = \sum_{i=1}^{N} \mathcal{S}_0^i,$$

where $\bar{V}_0^i = \mathcal{R}_{\Gamma_i} V_0 = \left[(\mathcal{R}_{\Gamma_i} \mathcal{R}_{\Gamma_1}^T V_0^1) \; (\mathcal{R}_{\Gamma_i} \mathcal{R}_{\Gamma_2}^T V_0^2) \; \cdots \; (\mathcal{R}_{\Gamma_i} \mathcal{R}_{\Gamma_N}^T V_0^N)\right]$ and $\mathcal{S}_0^i = \bar{V}_0^{iT} \mathcal{S}_i \bar{V}_0^i$. Since $\mathcal{R}_{\Gamma_i} \mathcal{R}_{\Gamma_j}^T$ is zero if $\Gamma_i \cap \Gamma_j = \emptyset$, only neighbor-to-neighbor communications are needed to compute $\bar{V}_0^i$ (steps 5-8 of Algorithm 4). The local coarse matrix $\mathcal{S}_0^i$ can then be computed without any communication (step 9). However, the assembly (or not) and factorization of $\mathcal{S}_0$ (step 10) to compute $\mathcal{M}_0$ require some global communication and several parallelization strategies for these operations are detailed below in Section 5.4.

5.3.2 Application of the two-level preconditioner $\mathcal{M}_{AS,2/\mathcal{S}}$

The application of the preconditioner (detailed in Equation (5.11) and Algorithm 3 for the baseline preconditioner $\mathcal{M}_{AS/\mathcal{S}}$) needs to be modified to include the coarse solve $\mathcal{M}_0$ required by the two-level preconditioner $\mathcal{M}_{AS,2/\mathcal{S}}$ defined in Equation (5.13).

The restriction of the coarse solve $\mathcal{M}_0 r$ to a subdomain can be computed as

$$\mathcal{R}_{\Gamma_i} \mathcal{M}_0 r = \mathcal{R}_{\Gamma_i} V_0 \; \mathcal{S}_0^{-1} \; V_0^T r = \bar{V}_0^i \mathcal{S}_0^{-1} \begin{pmatrix} V_0^{1T} \mathcal{R}_{\Gamma_1} \\ \vdots \\ V_0^{NT} \mathcal{R}_{\Gamma_N} \end{pmatrix} r = \bar{V}_0^i \mathcal{S}_0^{-1} \begin{pmatrix} V_0^{1T} r_1 \\ \vdots \\ V_0^{NT} r_N \end{pmatrix} \tag{5.14}$$

where $r_i = \mathcal{R}_{\Gamma_i} r$.

Computing the products $V_0^{iT} r_i$ and $\bar{V}_0^i z_0$ (where $z_0 = \mathcal{S}_0^{-1} V_0^T r$) can be done locally and do not present any particular challenge. However, the implementation of the coarse solve (step 7 of Algorithm 5) depends on the strategy used for the parallel factorization of the coarse matrix $\mathcal{S}_0$ (step 10 of Algorithm 4). In the next section, we discuss several parallel implementations for the coarse matrix factorization and corresponding solve steps.

Algorithm 4: `two-level` *setup of the preconditioner* `(two-level step (3) of Algorithm 1)`

// Baseline preconditioner (available before this thesis)
Compute $\bar{\mathcal{S}}_i$ from $\mathcal{S}_i$ by assembling diagonal blocks with neighbor to neighbor communications
Compute $\bar{\mathcal{S}}_i^{-1}$ (i.e., factorize $\bar{\mathcal{S}}_i$)
// CSC (extension designed for this thesis)
Solve $D_i^{-1}\mathcal{S}_i D_i^{-1}\ p_k^i = \lambda_k^i\ \bar{\mathcal{S}}_i\ p_k^i$ for the n_v smallest eigenvalues
Compute $V_0^i = [p_1^i\ p_2^i\ \cdots\ p_{n_v}^i]$
for $j \in \mathcal{N}(i)$ **do**
  Send $\mathcal{R}_{\Gamma_j}\mathcal{R}_{\Gamma_i}^T V_0^i$ to process $j-1$ and receive $\mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_j}^T V_0^j$
end
Compute $\bar{V}_0^i = [\mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_1}^T V_0^1\ \cdots\ \mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_N}^T V_0^N]$
Compute $\mathcal{S}_0^i = \bar{V}_0^{iT}\mathcal{S}_i\bar{V}_0^i$
Factorize $\mathcal{S}_0 = \sum_{i=1}^N \mathcal{S}_0^i$ **(parallelizations discussed in Section 5.4)**

Algorithm 5: `two-level` *application of the preconditioner* `(during two-level step (4a) of Algorithm 1)`

// Baseline preconditioner
Solve $z_i' = \bar{\mathcal{S}}_i^{-1} r_i$
for $j \in \mathcal{N}(i)$ **do**
  Send $\mathcal{R}_{\Gamma_j}\mathcal{R}_{\Gamma_i}^T z_i'$ to process $j-1$ and receive $\mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_j}^T z_j'$
end
Compute $z_i = z_i' + \sum_{j\in\mathcal{N}(i)} \mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_j}^T z_j'$
// CSC
Compute $r_0^i = V_0^{iT} r_i$
Solve $z_0 = \mathcal{S}_0^{-1}\begin{pmatrix} r_0^1 \\ \vdots \\ r_0^N \end{pmatrix}$ **(parallelizations discussed in Section 5.4)**
Update $z_i \leftarrow z_i + \bar{V}_0^i z_0$

5.4 Parallel strategies for the factorization (and corresponding solve step) of the coarse matrix $\mathcal{S}_0$

The CSC we have incorporated into MaPHyS is built in the *setup of the preconditioner* step of the solver. Its application occurs in the *solve step* at each iteration of the PCG algorithm. As stated in Section 5.3, a special care has to be taken in order to favor the parallel scalability when designing the CSC mechanism that must be numerically and computationally effective. We have designed five parallel strategies to build and apply the CSC; they differ in the parallel scheme employed for the factorization of the coarse matrix $\mathcal{S}_0$ (step 10 of Algorithm 4) and the corresponding coarse solve (step 7 of Algorithm 5). Indeed:

- $\mathcal{S}_0$ can be formed as either a dense (D) or a sparse (S) matrix.
- The associated factorization and solve steps can be handled on a single (1) process, on all the N processes (11 in the example of Figure 5.5) of the main MaPHyS communicator (`comm_main`) or on a sub-communicator `comm_CSC` of `CSC_NP` processes (with $1 \leq$ `CSC_NP` $\leq N$).
- Parallel distributed direct solvers (and especially sparse solvers such as Mumps and Pastix) often propose both a centralized (C) and a distributed (D) input format for the matrix; in the case of the centralized interface, note that the solver can be executed in a parallel distributed context (on `CSC_NP` processes), but the matrix and right-hand side must be first centralized (by MaPHyS in our case) on a single process and the solution is finally returned on that process (to MaPHyS).
- If $\mathcal{S}_0$ is handled on a dedicated `comm_CSC` sub-communicator, it can optionally be built hierarchically (H) or replicated (R) on multiple such sub-communicators.

We now give a brief overview of five combinations we have considered (and implemented) in this thesis, which will be further detailed in sections 5.4.1 to 5.4.5.

- **Dense Centralized (DC) strategy:** The coarse matrix $\mathcal{S}_0$ is formed as a dense (D) matrix and processed (factorized and solved) with a centralized (C) input matrix format. In the current version, we rely on `Lapack`, therefore $\mathcal{S}_0$ can be handled by only 1 process. If we note `CSC_NP` the number of processes associated with the factorization (and corresponding solve step) of $\mathcal{S}_0$ and DC(`CSC_NP`) the corresponding DC strategy, we can therefore currently only assess DC(1). Note that it would be possible to consider a centralized interface to a distributed dense solver (for instance using `ScaPAPACK`) on `CSC_NP` > 1 processes but we did not implement that variant. The `comm_main` communicator is the only one used (see Figure 5.5a), $\mathcal{S}_0$ being factorized on its `root` process, `CSC_master` (P0 in Figure 5.5a). Every iteration when the CSC needs to be applied, the residual is indeed first gathered on the `root` process of `comm_main` communicator, a centralized solve is performed and the solution is broadcast back to all the MPI processes (see Section 5.4.1). Such an implementation might be effective for moderate size problems using also a moderate number of MPI processes so that it is not worth exploiting the sparsity of $\mathcal{S}_0$.

- **Sparse Distributed (SD) strategy:** This variant exploits the sparsity (S) of $\mathcal{S}_0$ and relies on *all* (`CSC_NP` $= N$) the MPI processes to factorize $\mathcal{S}_0$ with a sparse direct solver using a distributed (D) input matrix. Similarly to DC(1), it only requires one communicator (see Figure 5.5a once again). However, here the reason is that *all* the N processes of the `comm_main` communicator are involved in the factorization of $\mathcal{S}_0$ (`comm_CSC` would exactly match `comm_main`). Compared to DC, $\mathcal{S}_0$ should be large enough and the number of MPI processes moderate enough to allow for an efficient parallel sparse solution using this SD(N) scheme. Section 5.4.2 further details this strategy.

- **Sparse Centralized (SC) strategy:** Sparse direct solvers may not adopt an optimal strategy for processing small or moderate size matrices such as $\mathcal{S}_0$ with respect to the number of computational units. SC(`CSC_NP`) dedicates a group of `CSC_NP` processes (3 in the example of Figure 5.5b) associated with a `comm_group` sub-communicator (G0 in Figure 5.5b) to factorize $\mathcal{S}_0$. In this strategy, this `comm_group` sub-communicator is employed as the `comm_CSC` communicator. See Section 5.4.3 for the algorithm description.

- **Sparse Hierarchical Distributed (SHD) strategy:** In this variant, all the MPI processes from the original MaPHyS communicator are first split into `NP_CSP` balanced sub-communicators (G0, G1, G2 and G3 in Figure 5.5c). All the processes of a sub-communicator compute their contribution to $\mathcal{S}_0$ and make this contribution available on their master (P0, P3, P6 and P9, respectively). All the `CSC_NP` masters (P0, P3, P6 and P9) of the sub-communicators are hierarchically (H) merged into a new `comm_masters` communicator (in red in Figure 5.5c) that is used to factorize $\mathcal{S}_0$. In the SHD strategy, this `comm_masters` communicator is employed as the `comm_CSC` communicator. Every iteration, each MPI process provides its contribution for the right-hand side of the coarse problem to its master. The masters then altogether solve the coarse problem and broadcast back the solution to the processes associated with their local communicator (G0, G1, G2 and G3, respectively). See Section 5.4.4 for the algorithm description corresponding to this SHD strategy. We note SHD(`CSC_NP`) the strategy set up with a `comm_masters` of `CSC_NP` processes (4 in the example of Figure 5.5c).

- **Sparse Replicated Centralized (SRC) strategy:** This variant is similar to SC, but all the entries of $\mathcal{S}_0$ are first redundantly stored on different MPI processes (P0, P3, P6 and P9 in Figure 5.5c) that will act as masters of different sub-communicators (G0, G1, G2 and G3, respectively) of size `CSC_NP` (equal to 3 for G0, G1 and G2, and to 2 for G3 in this imbalanced example). Each such sub-communicator (G0, G1, G2 and G3) redundantly plays the role of `comm_CSC` communicators (and thus in particular redundantly factorizes $\mathcal{S}_0$). This variant allows one to express more parallelism with a better locality in the communication scheme when broadcasting the solution once the coarse problem is solved. See Section 5.4.5 for the algorithm description corresponding to this SRC(`CSC_NP`) strategy and for the role of the `comm_masters` communicator in that strategy.

The DC(1) and SD(N) strategies do not use another communicator than `comm_main`, as the CSC is handled either by the master process of `comm_main` alone (DC(1)), or by all the processes in `comm_main` (SD(N)). However, in SC(`CSC_NP`), the CSC is handled by a sub-communicator `comm_CSC` (equal to `comm_group` = G0 in Figure 5.5b) of `comm_main` with `CSC_NP` processes. In the SHD(`CSC_NP`) strategy, a sub-communicator `comm_CSC` of `CSC_NP` processes is also defined on the `comm_masters`. The `comm_main` communicator is partitioned into `CSC_NP` sub-communicators `comm_group(g)`, where $1 \leq$ `g` $\leq$ `CSC_NP`, such that each process in `comm_CSC` is the master of a `comm_group`. In the SRC(`CSC_NP`) strategy, the `comm_main` communicator is partitioned in N/`CSC_NP` sub-communicators `comm_group(g)` ($1 \leq$ `g` $\leq N$/`CSC_NP`) of size `CSC_NP` (or possibly `CSC_NP` -1 for some of them if N is not a multiple of `CSC_NP`), each of them playing redundantly the role of a `comm_CSC` communicator. In the algorithms below, `i` is the index of the subdomain (such that the associated MPI rank is `i-1`), and `g` is the index of the group/sub-communicator.

Forming the matrix $\mathcal{S}_0$ or the right-hand side r_0 is performed through *gather*-like MPI collective communications (`MPI_Gather`, `MPI_Allgather` or a combination of both depending on the schemes, see below). Conversely, the redistribution of the solution z_0 would ideally be performed with *scatter*-like MPI collective communications. Because the overlapping regions of the AS preconditioner are stored redundantly on each associated process in the considered baseline MaPHyS parallel scheme (see Section 5.2.2), a *scatter*-like MPI collective handling overlapping regions would be necessary to do so. However, there is no such support in the MPI standard. A two-step scheme could be considered, consisting of a *scatter*-like MPI collective communication followed by neighbor-to-neighbor communications. We did not implement such a scheme. Instead z_0 is simply broadcast (with `MPI_Bcast`). The overhead due to this extra data movement has not been significantly penalizing and we keep the study of alternatives for future work.

Before getting in the details of each of those five considered parallel variants we have implemented in MaPHyS, note that the CSC designed for `ddmpy` in the previous chapter (Section 4.3.12.2) only supported the SD(N), SC(1) and SRC(1) strategies.

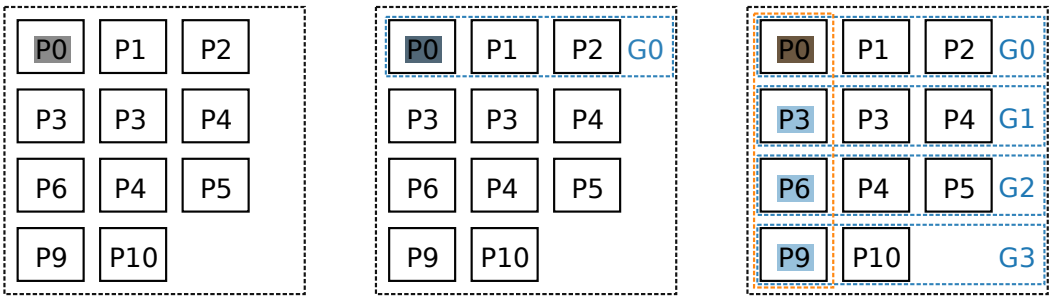
5.4.1 Dense centralized (DC) strategy

This strategy consists of using a dense sequential direct solver (*e.g.*, any Lapack implementation) on a single MPI process to apply the CSC. All the processes compute the contribution of their subdomain to $\mathcal{S}_0$ and these contributions are then gathered and summed (*i.e.,* assembled/reduced) onto a single process.

Figure 5.5a gives an example of MPI configuration for this implementation when using 11 MPI processes in `comm_main` (in black). Only `CSC_NP` = 1 MPI process is used to factorize the coarse matrix (`CSC_master = P0`).

For factorizing the coarse matrix, Algorithm 6 shows at line 1 that a first communication is performed in order to centralize and assemble the coarse matrix on the process of rank `CSC_master` (see Figure 5.6a). Then, the coarse matrix is factorized using a dense solver on the `CSC_master` process (line 3 in the algorithm and Figure 5.6b).

Then, at each iteration, one has to perform a coarse solve (Algorithm 7): the right-hand side is centralized and assembled on process `CSC_master` (line 1 in the algorithm and Figure 5.7a), before the solution can be computed sequentially by this process (line



(a) DC(1) and SD(11). P0 is the master of `comm_main`.

(b) SC(3). P0 is the master of both `comm_main` and G0 = `comm_CSC`.

(c) SHD(4) and SRC(3). P0 is the master of `comm_main`, G0 and `comm_masters`.

Figure 5.5: Communicator setup for the five parallel strategies employed for the CSC design on 11 processes. DC(1) and SD(11) rely only on the `comm_main` communicator. SC(3) needs an additional communicator `comm_group` for the CSC: `comm_group = comm_CSC = G0`. SHD(4) hierarchically builds the CSC on `comm_masters`: `comm_masters = comm_CSC`. SRC(3) redundantly handles the CSC on `comm_group(g)`, $1 \leq g \leq 4$: G0, G1, G2 and G3.

3 and Figure 5.7b) and broadcast back to all the MPI processes that pick up their part of z_0 (line 5 and Figure 5.7c).

Note once again that the subdomains are indexed from 1 to N (index 0 is reserved for the coarse space) whereas the MPI rank are numbered from 0 to $N-1$, the MPI rank associated with subdomain `i` is `i-1`.

Algorithm 6: `DC`: compute $\mathcal{S}_0^{-1}$

1 Gather the $\mathcal{S}_0^i$ and sum them into $\mathcal{S}_0$ on the master `CSC_master` of `comm_main`
2 **if** $i-1 ==$ `CSC_master` **then**
3 | Factorize $\mathcal{S}_0$ using the dense direct solver on `CSC_master`
4 **end**

Algorithm 7: `DC`: $\mathcal{M}_{AS,2/\mathcal{S}}$ application

1 Gather the r_0^i and sum them into r_0 on `CSC_master`
2 **if** $i-1 ==$ `CSC_master` **then**
3 | Solve $z_0 = \mathcal{S}_0^{-1} r_0$ using the dense direct solver on `CSC_master`
4 **end**
5 Broadcast the solution z_0 from `CSC_master` to all MPI processes in `comm_main`

5.4.2 Sparse distributed (SD) strategy

This implementation consists of using a sparse direct solver with a distributed input mode for the matrix (*e.g.* the Mumps solver): all the MPI processes in `comm_main` can take

part in the coarse factorization and solve. Each process only computes its contribution $\mathcal{S}_0^i$ to $\mathcal{S}_0$ and calls the parallel sparse direct solver; the sparse direct solver is then responsible for redistributing the data if needed in order to assemble these contributions and factorize $\mathcal{S}_0$. Algorithm 8 shows the factorization call for this implementation strategy which, as can be seen, involves no additional communication on MaPHyS side in its *setup of the preconditioner* step (see also Figure 5.6c). Indeed, the communications required to factorize the coarse matrix are performed by the sparse direct solver internally (see Figure 5.6d).

Regarding the MPI configuration, Figure 5.5a gives an example when using $N = 11$ MPI processes in `comm_main` (in black), implying the use of the same `CSC_NP` $= 11$ MPI processes for the sparse direct solver to factorize the coarse matrix.

The preconditioner application process is given by Algorithm 9. In this algorithm, we suppose the input mode for the right-hand side of the solver is centralized, as well as for the output of the solution[1], both provided and available on the `CSC_master` MPI process (P0 in Figure 5.5a). This strategy implies to gather the coarse right-hand side on the `CSC_master` MPI process (line 1 of the algorithm and Figure 5.7d) and broadcast the coarse solution in return (line 3 and Figure 5.7f) after the solve step (line 2 and Figure 5.7e).

The main advantage of this strategy is that most of the work is delegated to the distributed sparse direct solver: the implementation in MaPHyS is simpler, and the sparse direct solver, having access to all the resources in the main communicator, should be able (in theory) to optimize the parallel factorization and solve. In practice, it is often beneficial to guide the sparse direct solver by running it on a sub-communicator of `comm_main`, as proposed with the next strategies.

Algorithm 8: `SD`: compute $\mathcal{S}_0^{-1}$

1 Factorize $\mathcal{S}_0$ using the sparse direct solver in distributed input mode on `comm_main`

Algorithm 9: `SD`: $\mathcal{M}_{AS,2/\mathcal{S}}$ application

1 Gather r_0^i and assembly into r_0 on `CSC_master`
2 Solve $z_0 = \mathcal{S}_0^{-1} r_0$ using the sparse direct solver on `comm_main`
3 Broadcast the solution z_0 from `CSC_master` to all MPI processes in `comm_main`

5.4.3 Sparse centralized (SC) strategy

In this strategy, the sparse direct solver runs in a sub-communicator `comm_CSC` of the main communicator `comm_main` to perform the coarse factorization and solve. For the sparse solvers we have considered in this study, it means that the coarse matrix $\mathcal{S}_0$ must first be gathered and summed on the root process of `comm_main` (`CSC_master`).

[1]Notice that the Mumps solver now allows one to provide the right-hand side in distributed input mode, which was not the case at the beginning of this study. Using this feature would remove this gathering step.

Figure 5.5b gives an example of MPI configuration for this CSC implementation when using 11 MPI processes in `comm_main`. The CSC communicator responsible for factorizing and solving the coarse problem is `comm_CSC`. It contains `CSC_NP` = 3 MPI processes in the example of Figure 5.5b.

For the factorization of the coarse matrix, Algorithm 10 shows that the coarse matrix is first centralized and assembled on the `CSC_master` process (line 1 of the algorithm and Figure 5.6e). Then, the assembled coarse matrix is factorized in a distributed way on the communicator `comm_CSC` by the sparse direct solver (line 3 of the algorithm and Figure 5.6f).

The coarse solve in the application of $\mathcal{M}_0$ (Algorithm 11) requires first the centralization and the assembly of the coarse right-hand side on the `CSC_master` MPI process (line 1 of the algorithm and Figure 5.7g). Then, the coarse problem is solved by the sparse direct solver on the `comm_CSC` communicator (line 3 and Figure 5.7h). After this solve, the coarse solution is broadcast on the `comm_main` communicator (line 6 and Figure 5.7i).

Algorithm 10: `SC`: compute $\mathcal{S}_0^{-1}$

1 Gather $\mathcal{S}_0^i$ and assembly into $\mathcal{S}_0$ on `CSC_master` process
2 **if** $i - 1 \in$ `comm_CSC` **then**
3 | Factorize $\mathcal{S}_0$ using the sparse direct solver on `comm_CSC` (with centralized input mode for $\mathcal{S}_0$).
4 **end**

Algorithm 11: `SC`: $\mathcal{M}_{AS,2/\mathcal{S}}$ application

1 Gather r_0^i and assembly into r_0 on `CSC_master`
2 **if** $i - 1 \in$ `comm_CSC` **then**
3 | Solve $z_0 = \mathcal{S}_0^{-1} r_0$ using the sparse direct solver on `comm_CSC`
4 **end**
5 Broadcast the centralized solution z_0 from `CSC_master` to all MPI processes in `comm_main`

In terms of communication scheme, this mode is very close to the dense direct solver mode. On the performance side, using a sparse direct solver instead of a dense direct solver becomes more interesting when the size of the coarse system increases, causing scaling issues with the centralized dense direct solver strategy.

5.4.4 Sparse hierarchical distributed (SHD) strategy

This parallel strategy is similar to a strategy proposed in (Jolivet et al., 2013). The coarse problem is factorized and solved using a parallel sparse direct solver on a sub-communicator `comm_CSC` of `comm_main` of size `CSC_NP`. A distributed input mode for the matrix: `comm_main` is partitioned into `CSC_NP` sub-communicators `comm_group(g)` ($1 \leq$ `g` $\leq$ `CSC_NP`), such that each process in `comm_CSC` is the master of a distinct

comm_group(g). To introduce the communicators involved in this strategy, we consider the example given in Figure 5.5c. The main communicator comm_main with 11 MPI processes is split into 4 sub-communicators (in blue: G0 = comm_group(1), ..., G3 = comm_group(4)). The comm_masters sub-communicator (in red) is formed by the masters (P0, P1, P2 and P3) of group and is employed as comm_CSC communicator.

In this strategy, the coarse factorization (Algorithm 12) is performed in two steps: first, the coarse matrix is *partially* gathered and summed on the master process of each comm_group(g) communicator (line 1 of the algorithm and Figure 5.6g). Then, the sparse direct solver with distributed matrix input is called to run on the sub-communicator comm_CSC of all these local masters (line 2 and Figure 5.6i).

The corresponding coarse solve is given by Algorithm 13. Similarly to the other strategies, we suppose that the input mode for the right-hand side of the solver is centralized as well as the output of the solution. The gathering of the right-hand side can be performed directly in one gathering step as in the previous three strategies (see figures 5.7a, 5.7d or 5.7g). We present an alternative in two steps, consisting of a first gathering step (line 1 of the algorithm and Figure 5.7j) in each comm_group(g) ($1 \leq$ g $\leq$ CSC_NP) followed by another gathering step on the comm_master communicator (line 3 of the algorithm and Figure 5.7k). This two-step variant is equivalent, leading to the centralization and the assembly of the coarse right-hand side on the CSC_master MPI process, but its presentation here will allow to better understand in Section 5.4.5 how the SRC strategy gathering step of the right-hand side differs from the one of the four other variants including the SHD strategy. After that, the coarse problem is solved by the sparse direct solver on the comm_CSC communicator (line 4 and Figure 5.7l). The coarse solution is eventually broadcast on the comm_main communicator (line 5 and Figure 5.7m) as in the previous three strategies.

Notice that if CSC_NP = 1, this SHD(1) strategy is the same as SC(1) (and similar to DC(1) except for the management of the sparsity). On the contrary, if CSC_NP = N, SHD(N) is the same as SD(N).

Algorithm 12: SHD: compute $\mathcal{S}_0^{-1}$

1 Partially gather $\mathcal{S}_0^i$ and assembly into $\mathcal{S}_0^{(g)}$ on the master of comm_group(g)
2 Factorize $\mathcal{S}_0$ with the sparse direct solver in distributed input mode on comm_CSC

Algorithm 13: SHD: $\mathcal{M}_{AS,2/\mathcal{S}}$ application

1 Gather r_0^i and assembly into $r_0^{(g)}$ on the master of comm_group(g)
2 **if** $i - 1 \in$ comm_CSC **then**
3 | Gather $r_0^{(g)}$ and assembly into r_0 on process CSC_master
4 | Solve $z_0 = \mathcal{S}_0^{-1} r_0$ with the sparse direct solver on comm_CSC
5 **end**
6 Broadcast the solution z_0 from CSC_master to all MPI processes in comm_main

5.4.5 Sparse replicated centralized (SRC) strategy

This strategy extends the SC strategy by allowing to replicate the coarse problem on disjoint and equally sized sub-communicators of the `comm_main` MPI communicator. By similarity with the SHD strategy, these communicators are called `comm_group(g)`; however, here, `CSC_NP` represents the size of these sub-communicators instead of the number of groups. This strategy allows us to replace a global scatter/gather-like pattern at each iteration involving all processes of `comm_main` by a more local one in each group.

A parallel distributed solver is used on each sub-communicator `comm_group(g)` to factorize and solve the coarse system. Figure 5.5a shows an example of this strategy with 11 MPI processes in `comm_main`. First, the `comm_main` is split into 4 sub-communicators with `CSC_NP` = 3 MPI processes (in blue): G0 (`comm_group(1)`), ..., G3 (`comm_group(4)`). Notice that G3 has one less MPI process because `CSC_NP` = 3 does not divide $N = 11$. The masters of all groups are assembled into a `comm_masters` sub-communicator (in red).

The centralization of the coarse matrix occurs here in two steps. The coarse matrix is first partially centralized and assembled into each `comm_group(g)` sub-communicator (line 1 in Algorithm 14 and Figure 5.6j). Then, the partially centralized coarse matrices are allgathered and summed on the `comm_masters` communicator (line 3 and Figure 5.6k). After these communications, the entire coarse system is duplicated on the masters of all the `comm_group(g)` sub-communicators. Then, the duplicated coarse system is (redundantly) factorized concurrently on each `comm_group(g)` communicator (line 5 and Figure 5.6l).

The centralization communication scheme of the coarse right-hand side and the solution in Algorithm 15 is performed in a way very similar to the factorization scheme (see figures 5.7n to 5.7p), the coarse solution being eventually broadcast redundantly within each `comm_group(g)` sub-communicator (line 6 and Figure 5.7q).

This strategy was designed to enhance the scalability of the preconditioner application on a large number of MPI processes. Despite the required allgather operation occuring when centralizing and assembling the coarse right-hand side, replacing the broadcast operation on `comm_main` in the SC strategy (Algorithm 11 line 5) by a broadcast on each `comm_CSC(g)` might lead to better parallel performance.

Algorithm 14: `SRC`: compute $\mathcal{S}_0^{-1}$

1 Gather $\mathcal{S}_0^i$ and assembly into $\mathcal{S}_0^{(g)}$ on the master of `comm_group(g)`
2 **if** $i - 1 \in$ `comm_masters` **then**
3 | Allgather $\mathcal{S}_0^{(g)}$ and assembly into $\mathcal{S}_0$ on `comm_masters` communicator
4 **end**
5 Factorize $\mathcal{S}_0$ with the sparse direct solver in centralized input mode on `comm_group(g)`

Algorithm 15: SRC: $\mathcal{M}_{AS,2/\mathcal{S}}$ application

1 Gather r_0^i and assembly into $r_0^{(g)}$ on the master of `comm_group(g)`
2 **if** $i-1 \in$ `comm_masters` **then**
3 | Allgather $r_0^{(g)}$ and assembly into r_0 on `comm_masters` communicator
4 **end**
5 Solve $z_0 = \mathcal{S}_0^{-1} r_0$ with the sparse direct solver on `comm_group(g)`
6 Broadcast the solution z_0 from the master to all MPI processes in `comm_group(g)`

5.5 Experimental study

5.5.1 Experimental setup

All the parallel experiments presented in this study were performed on the GENCI Occigen cluster, hosted at CINES. The part of the cluster in use is composed of 2 dodeca-core (24 cores per node) Haswell Intel Xeon E5-2690 v3 @ 2.6 GHz nodes with 64 and 128 Go RAM per node.

The code was compiled with Intel compiler version 17.0.0 and linked with the Intel MPI version 2017.0.0. The sparse computations were performed using either Pastix or Mumps. All dense computations (including solving the eigenproblems) rely on the multithreaded Intel MKL version 2017.0.0. Using the `dsygvx` Lapack routine, only selected eigenpairs are computed.

All the experiments are designed such that the nodes of the cluster are fully occupied (hence the number of cores is always a multiple of 24). MaPHyS uses one MPI process per subdomain and each thread is bound to one core in the presented experiments. MaPHyS can process each subdomain using multithreading (Agullo et al., 2016b), but unless stated otherwise (Section 5.5.2.3), the solver is set up with one thread per process, resulting in an equal total number of subdomains, MPI processes, threads and CPU cores. For instance, an execution on 12,288 subdomains (x-axis) in figures 5.9 to 5.12 was executed with 12,288 MaPHyS MPI processes on 12,288 CPU cores (512 nodes). Multithreading is only assessed in Section 5.5.2.3 where each subdomain and its corresponding MPI process may use multiple threads and CPU cores. Notice that on the Occigen cluster, memory swapping is disabled by default so that all the runs are performed in memory without time overhead due to the swap. The simulation campaigns were realized using the JUBE Benchmarking Environment (Frings et al., 2010).

The rest of this section is organized as follows. First in Section 5.5.2 we investigate the parallel performance of the various strategies on the Darcy academic test case, consistently with the studies conducted in the previous chapters (see sections 3.5.1 and 4.4.1). The performance of the solver integrated in two applications is then considered. Section 5.5.3 tackles the simulation of an airflow through the nose by solving the incompressible Navier-Stokes equations with the Alya[2] simulation code for high performance computational mechanics developed at the Barcelona Supercomputing Center. In Section 5.5.4, we present

[2] `https://www.bsc.es/research-development/research-areas/engineering-simulations/alya-high-performance-computational`

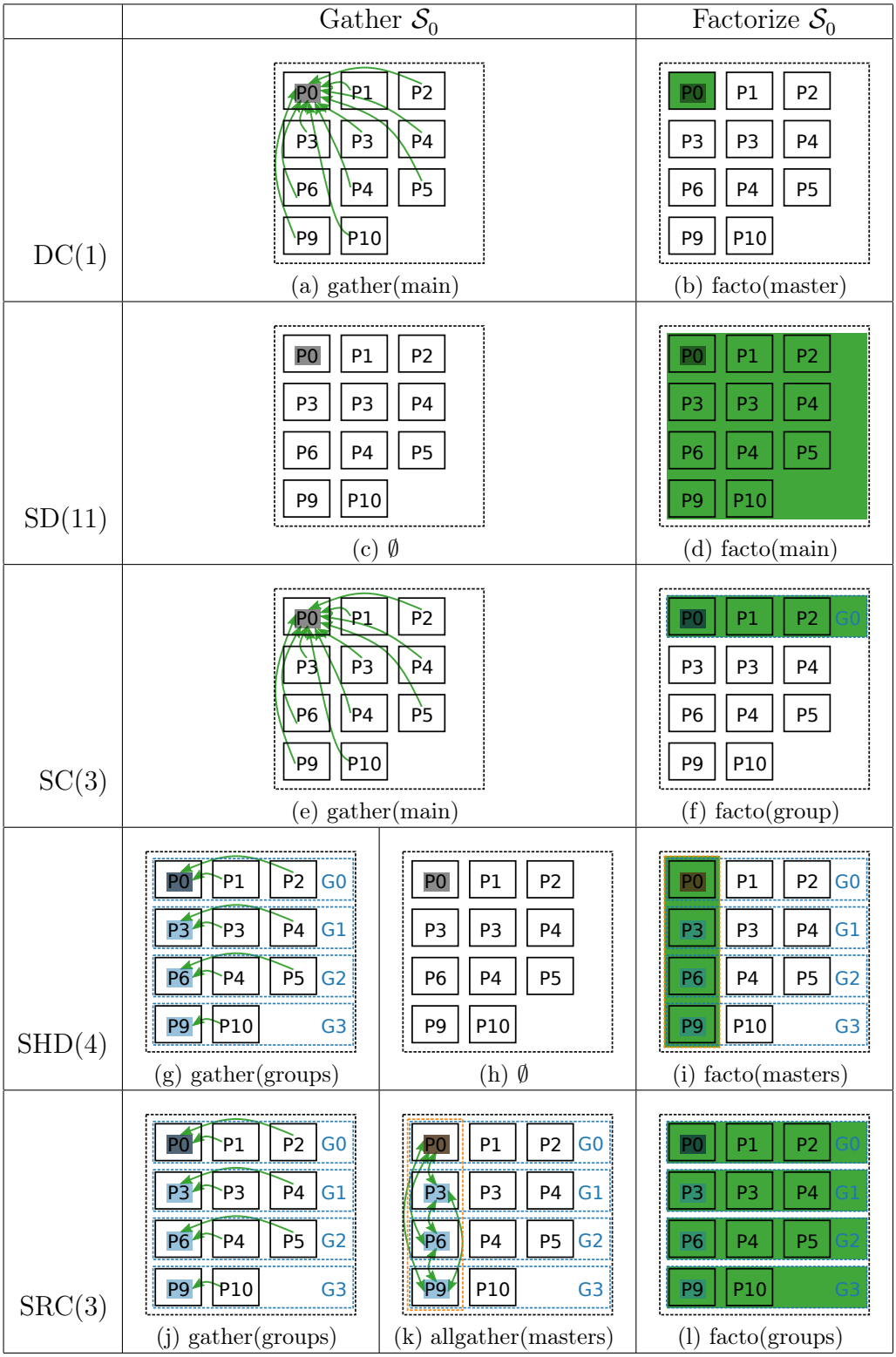


Figure 5.6: Parallel strategies for computing $\mathcal{S}_0^{-1}$.

the performance of the solvers integrated in the plasma propulsion simulation code AVIP jointly developed by CERFACS and the Laboratoire de Physique des Plasmas at École

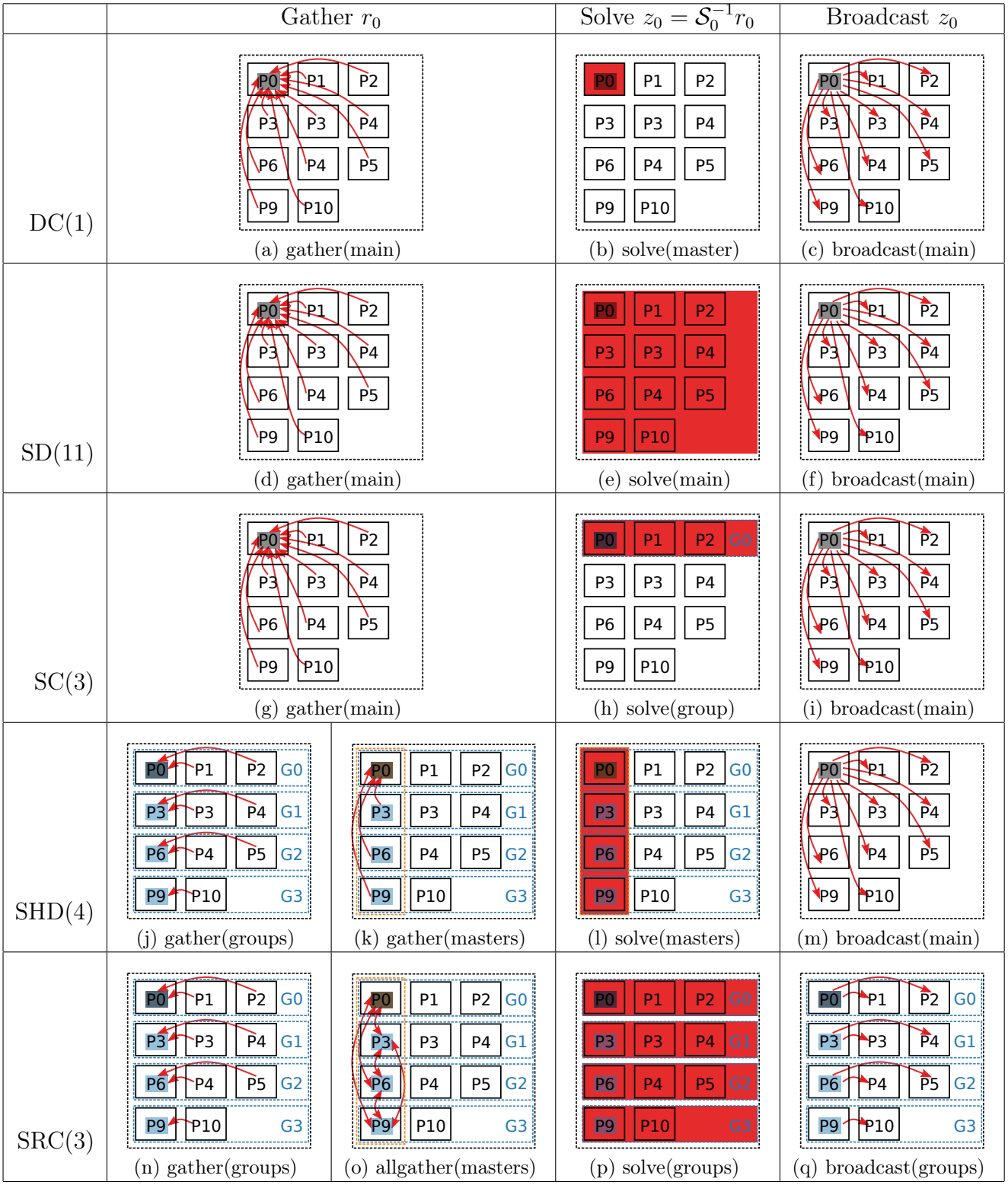
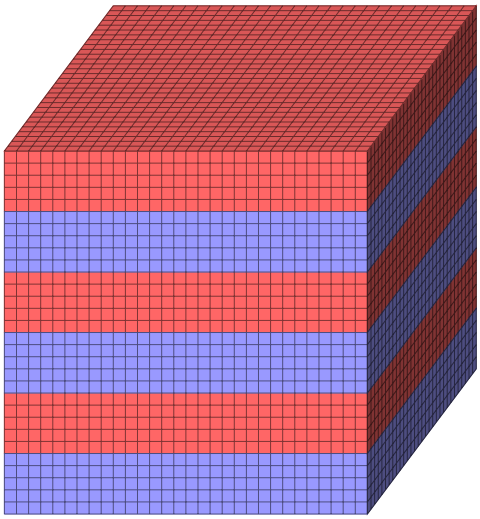
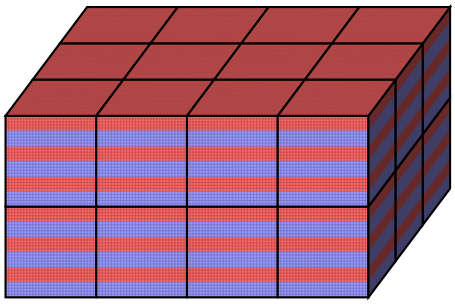


Figure 5.7: Parallel strategies for applying $\mathcal{S}_0^{-1}$.

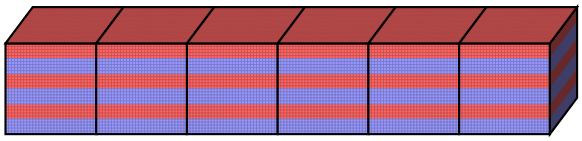
Polytechnique.



(a) One subdomain ($30\times30\times30$ Q_1 elements)



(b) The *cuboid* test case: $4 \times 3 \times 2$ subdomains



(c) The *bâton* test case: $6 \times 1 \times 1$ subdomains

Figure 5.8: Geometry of the Darcy test cases. The conductivity k in Equation (5.15) is either $k = 1$ (blue) or $k = 10,000$ (red).

5.5.2 Darcy academic test case

5.5.2.1 Test case

As in the previous chapters (see sections 3.5.1 and 4.4.1), we consider a stationary heterogeneous diffusion equation (or Darcy equation) in a 3D stratified medium

$$\nabla \cdot (k \nabla u) = 1. \tag{5.15}$$

The equation is discretized using the Finite Element Method. Each local subdomain is a cube of $30 \times 30 \times 30$ elements. Along the vertical dimension, each cube is divided into 6 layers (of $30 \times 5 \times 30$ Q_1 elements) of alternating conductivity $k = 1$ and $k = 10,000$, as presented in Figure 5.8a. The `genfem` python module[3] developed in the context of this thesis is employed for generating the problems.

Two different sets of test cases are built from these local cubic subdomains, only differing in the relative positions of the subdomains:

- the *cuboid* test case (Figure 5.8b), where the total number N of subdomains is decomposed into $N = N_1 \times N_2 \times N_3$ such that the global domain is as cubic as possible. Given a number N of subdomains, we use a pre-processing script to compute its prime number decomposition and group them in order to obtain three factors as close as possible. Table 5.1 gives the values used for N_1, N_2 and N_3 in the experiments.

- the *bâton* test case (Figure 5.8c), where the N subdomains are all aligned on the horizontal axis ($N_1 = N$, $N_2 = N_3 = 1$) in a 1D-decomposition in a similar fashion

[3] `https://gitlab.inria.fr/solverstack/genfem`

to the Darcy test case in (Spillane et al., 2014b). It corresponds to the setup employed for Chapter 4 (see Section 4.4.1) as well as Section 3.5.5 and more details are provided in the present section (in particular in Table 5.1).

In both cases, the global domain is the cuboid $[0, N_1] \times [0, N_2] \times [0, N_3]$ and has $(30\,N_1 + 1) \times (30\,N_2 + 1) \times (30\,N_3 + 1)$ vertices. A Dirichlet boundary condition is applied on the left of the domain ($x = 0$) and a Neumann condition on every other boundary.

	Bâton				Cuboid			
N	N_1	N_2	N_3	ndofs	N_1	N_2	N_3	ndofs
24	24	1	1	0.96M	4	3	2	0.67M
48	48	1	1	1.4M	4	4	3	1.3M
96	96	1	1	2.8M	6	4	4	2.7M
192	192	1	1	5.5M	8	6	4	5.3M
384	384	1	1	11M	8	8	6	11M
768	768	1	1	22M	12	8	8	21M
1,536	1,536	1	1	44M	16	12	8	42M
3,072	3,072	1	1	88M	16	16	12	83M
6,144	-	-	-	-	24	16	16	167M
12,288	-	-	-	-	32	24	16	333M

Table 5.1: Number of subdomains and degrees of freedom per dimension

The tolerance threshold for the CG algorithm is set to 10^{-6}. In the following, we first study the impact of the various parallel strategies for handling the CSC proposed above in Section 5.5.2.2. We then study the opportunity to combine two-level preconditioning (object of this thesis) with two-level (MPI+thread) parallelism (an additional feature of the resulting hybrid solver) in Section 5.5.2.2.

5.5.2.2 Performance results with MaPHyS (without multithreading)

The baseline MaPHyS solver is compared to the two-level extension proposed in this thesis with 1, 3 or 5 eigenvectors per subdomain. Due to the layered geometry, one can expect (Galvis and Efendiev, 2010) that 3 eigenvectors per subdomain are enough to cancel the effect of the heterogeneity. The five parallel strategies introduced in Section 5.4 are compared.

Figure 5.9 shows that the CSC limits the number of iterations to be performed by the conjugate gradient. In the *bâton* problem, it even allows convergence to be reached with a larger number of subdomains ; with 1 eigenvector per subdomain, the number of iterations remains below 167 iterations, while using 3 eigenvectors per subdomain decreases this number to 14. As expected, increasing the number of eigenvectors per subdomain beyond 3 does not significantly improve the convergence. In the *cuboid* case, the effect of CSC is not as definite: the number of iterations to reach convergence keeps growing up to 12,288 subdomains, although it is 4 times less than without CSC. This is not in contradiction with the bound on the condition number in Chapter 3, in which there is a factor N_c^2, where $N_c - 1$ is the maximum number of neighbors of a subdomain. In the *bâton* test case, each

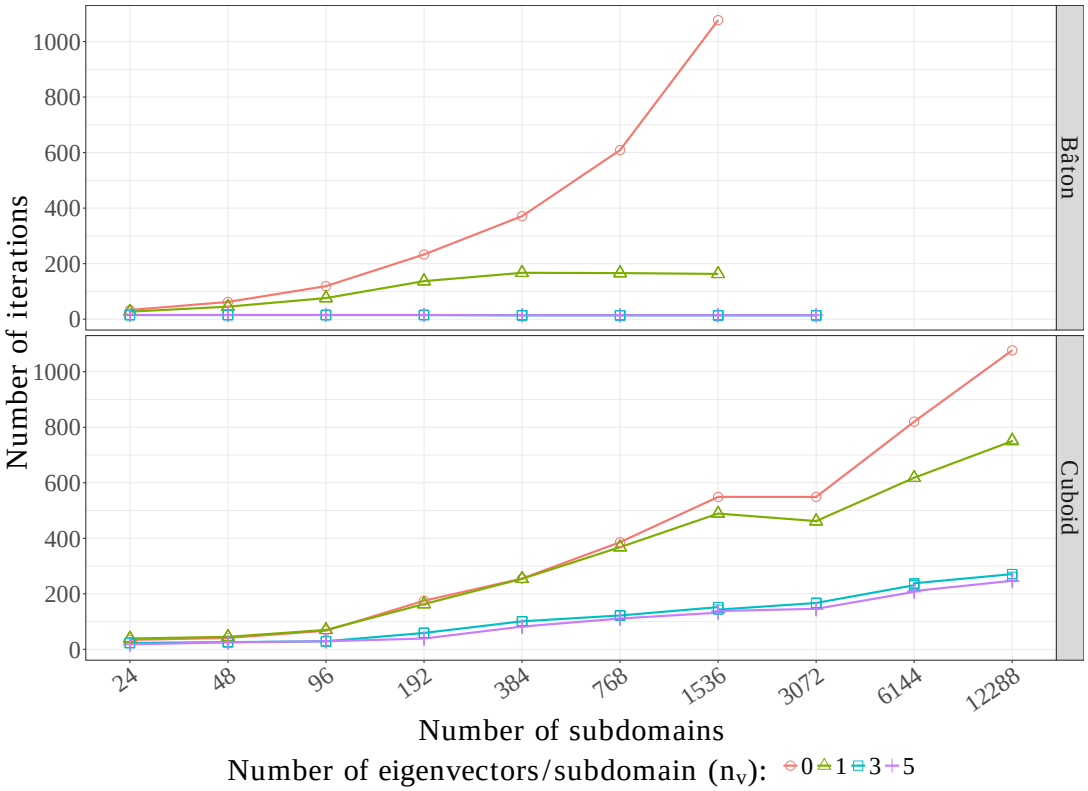


Figure 5.9: Number of iterations to reach convergence ($\|\mathcal{K}u - f\|/\|f\| < 10^{-6}$). Increasing the number of eigenvectors per subdomain from 0 to 3 improves the convergence.

interior subdomain has only two neighbors (one on each side), and $N_c^2 = 3^2 = 9$. In the *cuboid* test case, each interior subdomain has 26 neighbors (including those that only share a corner), and $N_c^2 = 27^2 = 729$. Since the local subdomain geometry is the same in both test cases, all the other factors in the bound are the same. This negative effect of the number of neighbors in the *cuboid* case could be mitigated by applying the CSC using a deflation (or projection) mechanism (AS,D/$\mathcal{S}$, developed for `ddmpy` but not incorporated into MaPHyS) instead of the additive correction (AS,2/$\mathcal{S}$, made available for both `ddmpy` and MaPHyS) from Equation (5.13). One can also note that the maximum number of subdomains in one direction is only 32 in the *cuboid* case; we expect that the number of iterations would stop increasing if experiments with an even larger number of subdomains were to be performed.

The reduction of the number of iterations achieved by using a CSC and increasing the number of eigenvectors per subdomain comes at a cost: one has to factorize and solve a global coarse problem whose size increases with the total number of subdomains and the number of eigenvectors per domain. This reflects in the time spent in the *setup of the preconditioner* (Figure 5.10) and in the *solve step* (Figure 5.11). The total time to solution for 1 right-hand side is given in Figure 5.12.

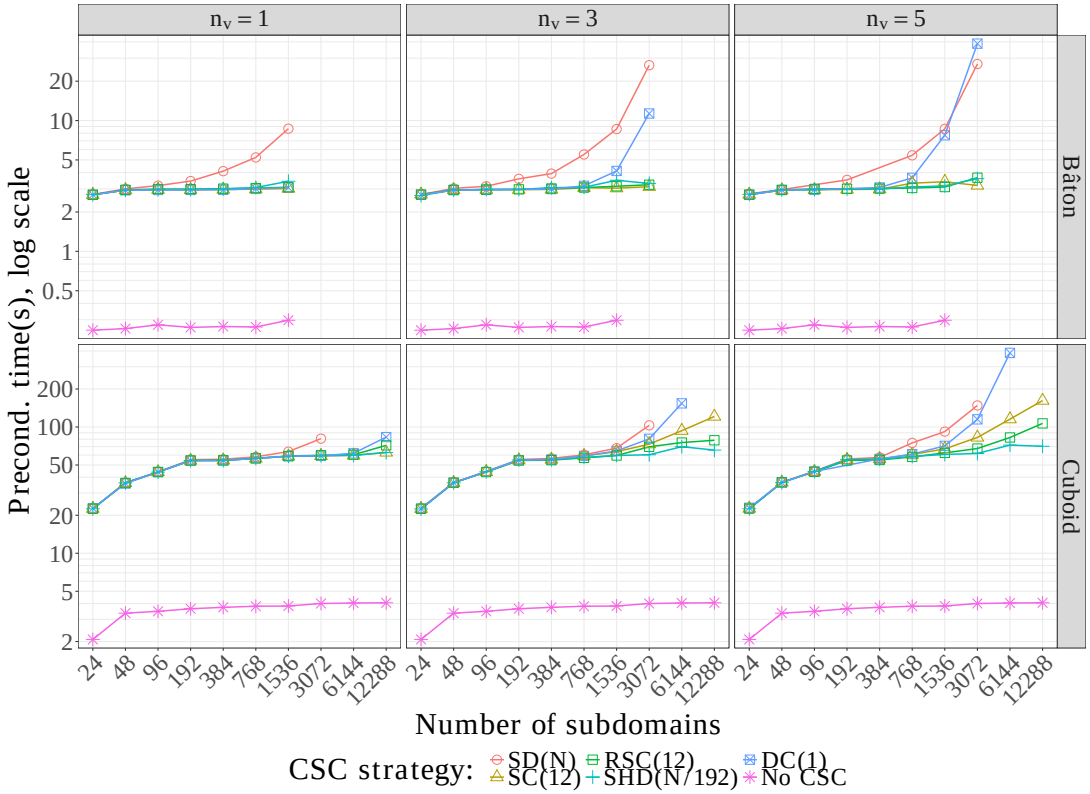


Figure 5.10: Time spent in the *setup of the preconditioner*. The additional cost of CSC depends on the the size of the coarse space (number of subdomains $\times n_v$), the number of available CPU cores (equal to the number of subdomains) and the chosen CSC strategy.

The results show that the SD and DC strategies do not scale and can only be used on a small number of subdomains. For the SD strategy, this is a limitation of distributed

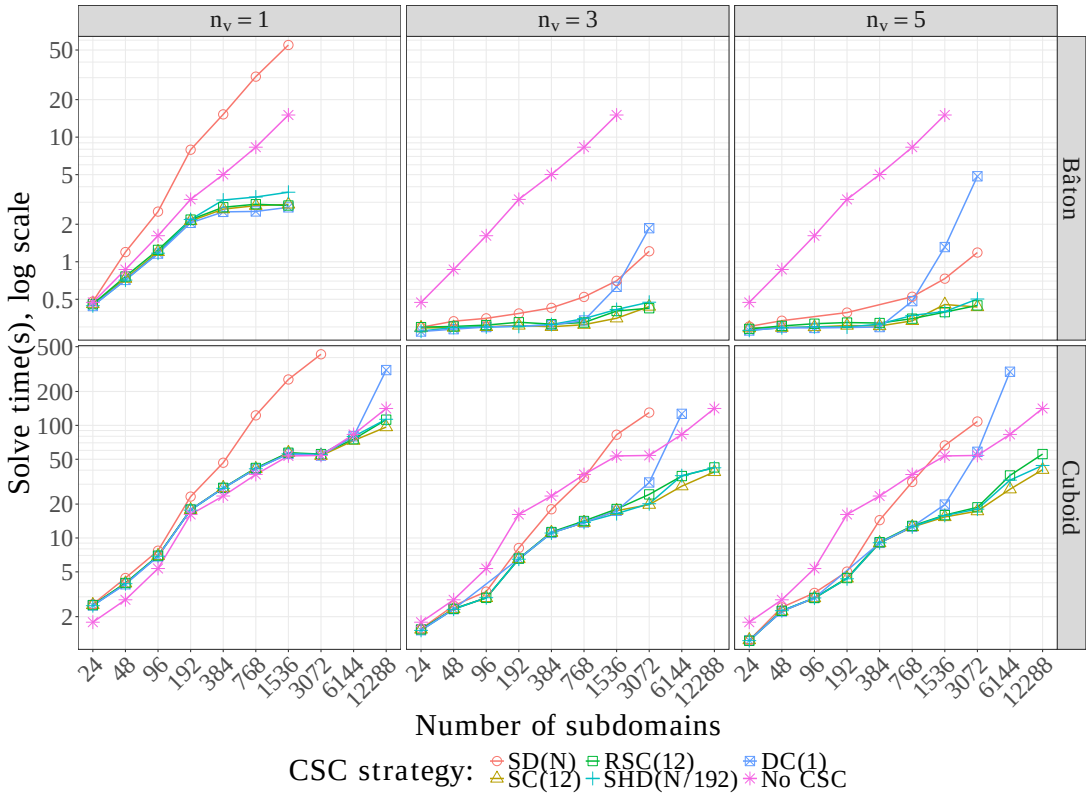


Figure 5.11: Time spent in the *solve step*. Depending on the geometry and the chosen CSC strategy, the solve time can be orders of magnitude lower than in the baseline solver.

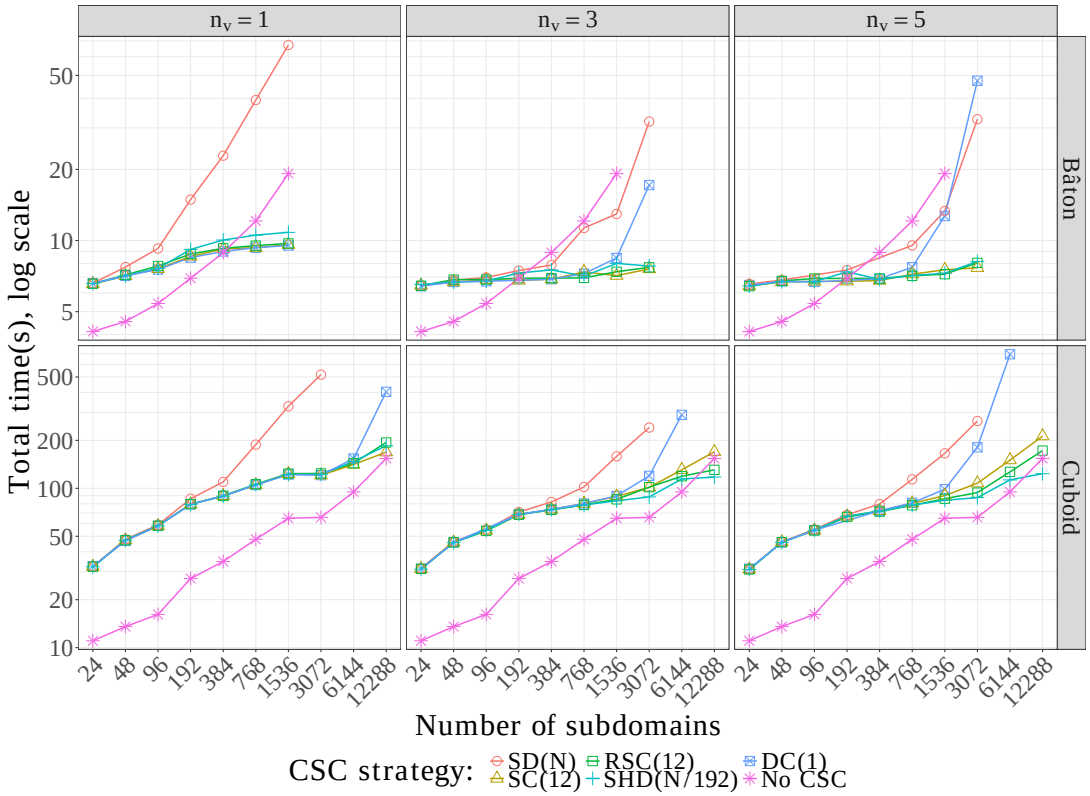


Figure 5.12: Total time to solution for 1 right-hand side. The CSC improves the solver scalability.

direct solvers: the granularity of the problem (between $n_v = 1$ and $n_v = 5$ unknowns per process) is too small and the solver is not able to use efficiently all the CPU cores available. On the contrary, the DC strategy centralizes the problem on one core. The scalability of this strategy is limited by both the ability of one single process to solve the global coarse problem and the communications needed to centralize the right-hand side and broadcast the solution to the other processes.

The SC strategy allows the coarse problem to be solved on a sub-communicator of appropriate size, ensuring that the direct solver is used at the most efficient granularity. However, global communications are still needed since the coarse problem is centralized onto the sub-communicator. The SRC strategy performs slightly better by replicating the coarse problem, splitting the global communications into a set of local communications inside each communicator. The SHD strategy is even faster for the *setup of the preconditioner*, while being very competitive for the *solve step*, yielding the best overall scalability of the total time to solution.

As explained above, the CSC is much more effective on the *bâton* test case compared to the *cuboid* test case, for two reasons: first, each subdomain has 9 times more neighbors in the *cuboid* than in the *bâton*. As a result, both the number of iterations and the cost of all neighbor-to-neighbor communications are higher in the *cuboid* case. The second reason is that the number of iterations needed to reach convergence without CSC grows linearly with the diameter of the domain, which is $N_1 = N$ for the *bâton* and $\max(N_1, N_2, N_3) \approx N^{\frac{1}{3}}$ for the *cuboid*. As a result, the baseline solver without CSC performs much better on the *cuboid* test case, and the CSC is not as much needed as in the *bâton* test case.

5.5.2.3 Performance results with MaPHyS, combining two-level preconditioning with two-level parallelism

MaPHyS can benefit from two-level (MPI+thread) parallelism by using several cores to handle each subdomain. Keeping the number of cores constant, two-level parallelism makes possible to reduce the total number of subdomains (while increasing their respective sizes) and can improve the robustness of the iterative solve in MaPHyS. This feature can be used together with the newly implemented CSC described in the present chapter.

We first consider experiments on the test example associated with a long *bâton* as described on Figure 5.8. The dimensions of the bâton are: $(30 \times 3072 + 1) \times 31 \times 31$, that results in a linear system of size larger than 88 million. We vary the number of subdomains from 3,072 down to 128, while increasing the number of threads per subdomain from 1 up to 24, so that the total number of cores is kept constant equal to 3,072; the number of unknowns per core remains constant. The heterogeneity of the problem has been set to 10,000. The sparse direct solver employed both within the subdomains and for the CSC is Mumps. We display in Figure 5.13 the total time to solution decomposed into the time to factorize the local matrices and compute the local Schur complements (denoted Schur Factorization, in blue), the time spent in setting up the preconditioner (denoted Pcd Setup, in green) and eventually the time in the iteration loop plus the solution inside the subdomains (denoted Iterative Solve, in red); the number of PCG iterations needed to reach convergence is displayed above the bars representing the time of these three steps. The first set of experiments was conducted without CSC (first subfigure on the

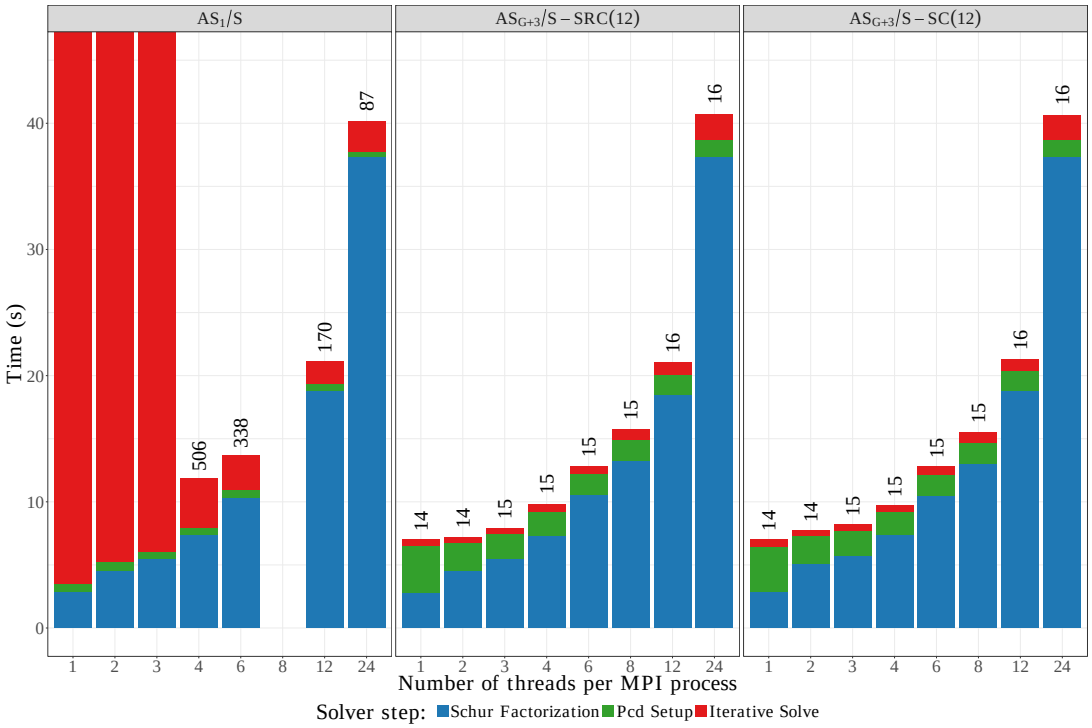


Figure 5.13: Step by step time for MaPHyS with two-level parallelism on 3,072 cores for solving a bâton with heterogeneity 10,000 and 88 million unknowns.

left: $AS_1/\mathcal{S}$). It can be observed that the MPI+threads implementation allows us to solve linear systems that are not tractable without this two-level implementation because the numerical behaviour of the scheme is too poor on a large number of domains. The second observation is that the best balance between the number of processes and threads depends on whether the linear system has to be solved once or for several right-hand sides given in sequence. For the first situation, 4 threads per MPI process gives the shortest time to solution. For multiple right-hand sides, 12 threads per process exhibits the best trade off as the time in the setup is reasonable and the iterative part is the fastest. This is particularly interesting on non SPD problems where generic scalable preconditioners might not exist. Although not discussed here, this MPI+thread implemenation also allows for a better memory scalability. We refer to the PhD thesis of Nakov (2015) for a complete description of this two-level MPI+thread implementation.

We have considered two other sets of experiments with CSC, referenced to as $AS_{G+3}/\mathcal{S}$, since: (1) MaPHyS is based on $AS/\mathcal{S}$, (2) we have incorporated an additive (+) correction, (3) with a GenEO (G) coarse space, (4) parametrized here with 3 vectors per subdomain. The first other set is based on the SRC parallel strategy (second subfigure in the middle: $AS_{G+3}/\mathcal{S} - SRC(12)$). The second other set is based on the SC parallel strategy (third subfigure on the right: $AS_{G+3}/\mathcal{S} - SC(12)$). In both cases, `CSC_NP` $= 12$ cores are dedicated for the CSC. A few observations can be made. On that example the two strategies perform very similarly in terms of performance. Second, for this scale of problem and platform sizes, the CSC with one thread per process achieves the best performance as the CSC already performs a very good numerical job.

The weak scalability of the CSC used together with two-level parallelism is assessed on a *bâton* as described on Figure 5.8 but with larger subdomains. Each subdomain is a cube of $90 \times 90 \times 90$ elements. The number of threads per subdomain is set to 12, which corresponds to half the number of cores per node on the GENCI Occigen platform, and allows MaPHyS to be tested on a larger number of unknowns. The total number of unknowns of the problem is $(N \times 90 + 1) \times 91^2$ where N is the number of subdomains. For $N = 1,344$, there are more than 1 billion unknowns. The performance results of MaPHyS are displayed in Figure 5.14, where the numbers on top of the bars are the numbers of iterations. We compare the performance of the baseline MaPHyS version (left part "No_csc" of the figure) relying on the Mumps internal sparse direct solver to the performance of MaPHyS with a CSC relying on a sparse centralized parallel strategy handled on a sub-communicator of 24 processes (SC(24)), relying on either Mumps (center) or Pastix (right) internal sparse direct solver for performing the Schur Factorization step.

Several observations can be made. First, because we consider weak scaling, the time spent in the Schur factorization is constant. Second, we can observe some artefacts when there are only two subdomains. Indeed, the local Schur complement are twice smaller than in the other test cases (the subdomain interface is composed of only one face, instead of two in the other experiments). Furthermore, the local component of the one-level preconditioner is equal to the global Schur complement, so that CG converges in one iteration. Third, without the CSC (left part "No_csc"), one can observe the expected growth of the number of iterations that is essentially proportional to the number of domains, yet limited thanks to the usage of multithreading which allows for reducing the number of subdomains (thus by a factor of 12 in this setup). With the CSC preconditioner (center and right), we can see that the number of iterations remains constant while most of the time is spent in the setup of the preconditioner, mainly in the solution of the local generalized eigenvalue problem. All the numerical kernels scale well as the time does not depend on the number of subdomains. The usage of Mumps (center) and Pastix (right) internal sparse direct solvers both led to very high performance, with a slight advantage when relying on Pastix. The results obtained with Pastix correspond the latest research version at the moment the experiments were conducted while the ones obtained with Mumps correspond to the community released version of the direct solver (and does not include all the multithreading features designed in Sid Lakhdar (2014)).

This figure also illustrates the hybrid nature of the linear solver that bears some resemblance to the direct methods where the setup, that corresponds to the factorization for direct methods, is also the most time consuming part of the solution and can be amortized if several right-hand sides have to be solved. For instance, on 1344 subdomains, the problem with 10^9 unknowns has a setup time of 19 min. for a solve time of 15.5 s. per right-hand side. On the other hand, the cost of the iterative part of the hybrid solver, because the number of iterations is monitored by the CSC mechanism, does scale perfectly which is usually not true for the forward/backward substitution phases of the sparse direct solvers.

Both the *bâton* and *cuboid* Darcy test cases are very regular due to the geometric domain decomposition: all the subdomains are identical to one another, thus eliminating all load-balancing issues from the analysis. We therefore now consider more irregular test cases arising from airflow and plasma propulsion applications from the Alya and AVIP

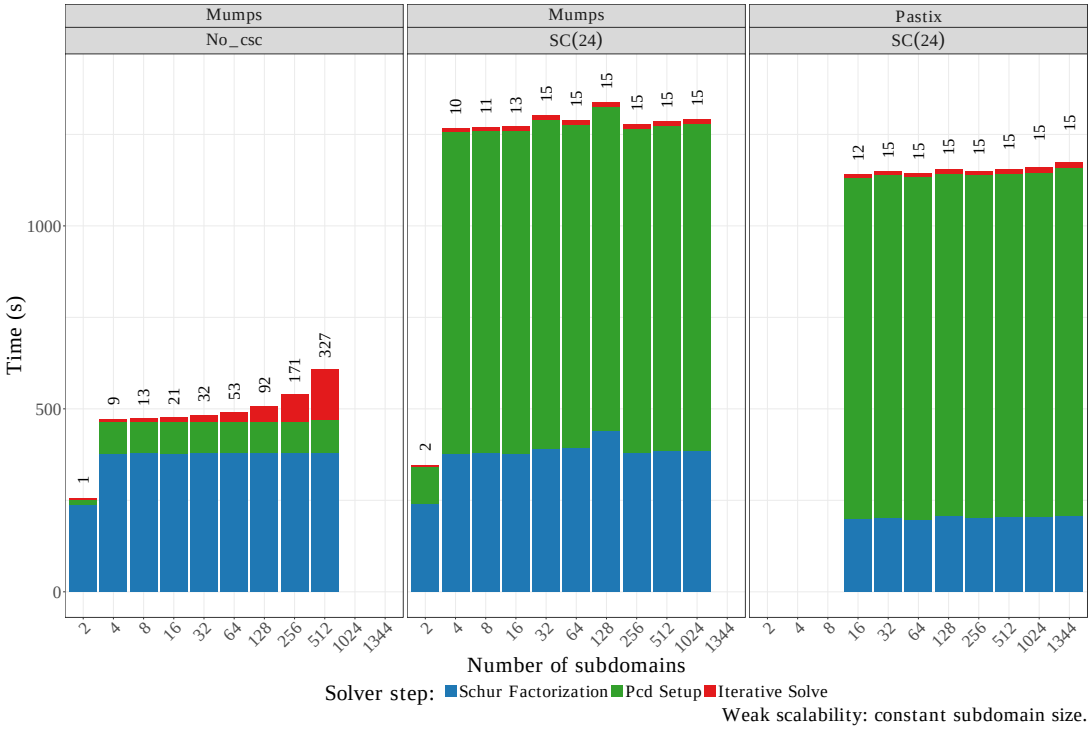
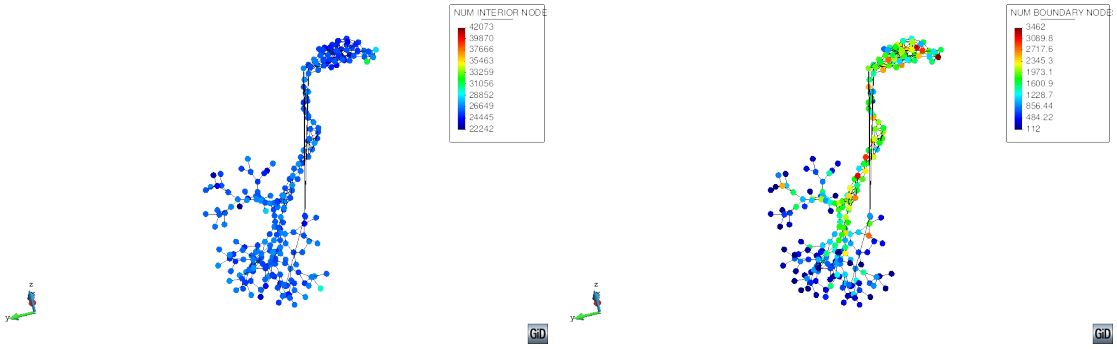


Figure 5.14: Step by step time for MaPHyS, with either Mumps (left and center) or Pastix (right) internal sparse direct solver, on a large *bâton* test case with $91 \times 91 \times 91 = 7.5\ 10^5$ unknowns per subdomain ($1.0\ 10^9$ unknowns for $N = 1,344$ subdomains). Either no CSC is employed (left) or the SC(24) parallel strategy is employed for handling the CSC (center and right).



(a) Number of interior vertices per subdomain (b) Number of interface vertices per subdomain

Figure 5.15: Respiratory test case: pseudo-1D domain decomposition into 255 subdomains.

simulation software, respectively. We do not study the combined effects of two-level preconditioning with two-level parallelism in the following but focus only on the former effect, main object of this thesis.

5.5.3 A respiratory airflow application using the Alya code

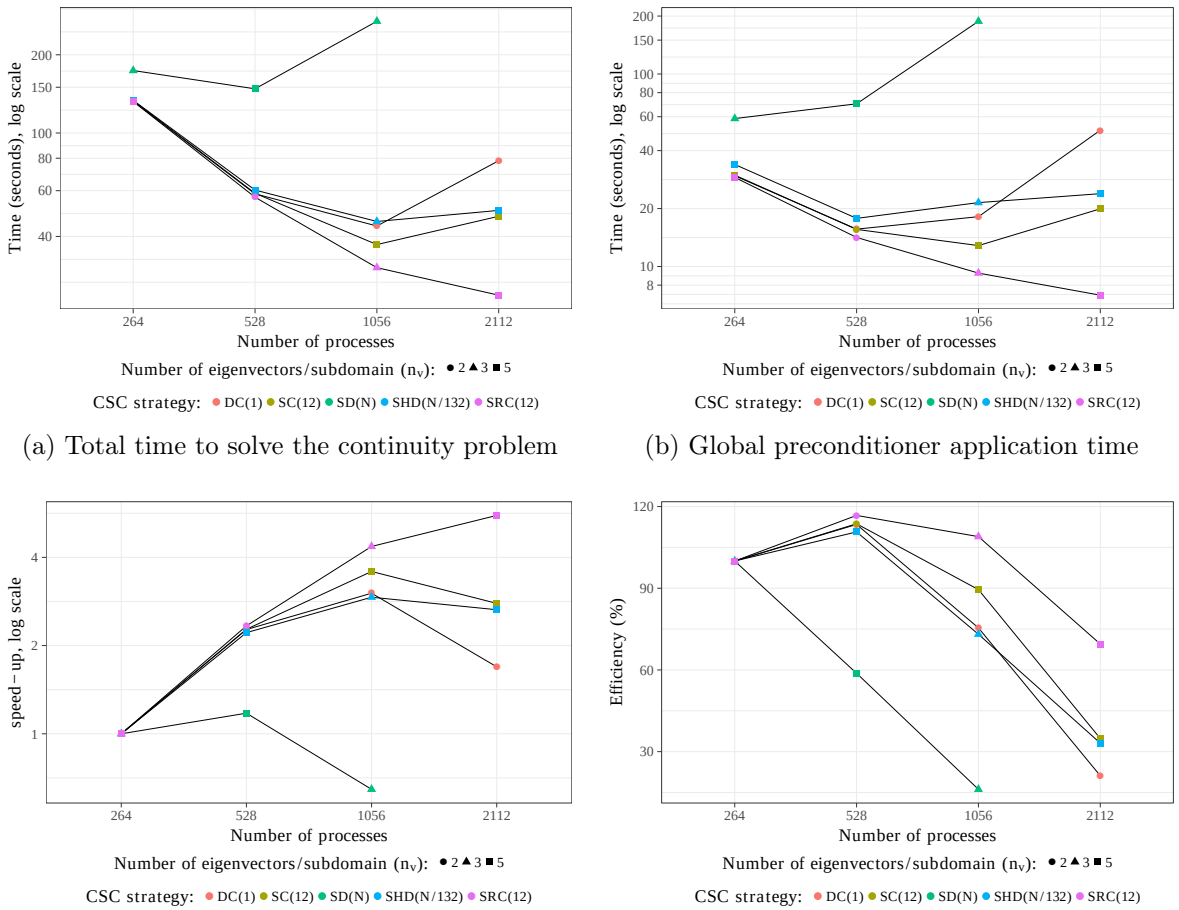
5.5.3.1 Test case

Alya is a high performance computational mechanics code to solve coupled multi-physics / multi-scale problems, which are mostly coming from engineering applications (Vázquez et al., 2016). The physics considered in this study is the incompressible Navier-Stokes equations. They are solved implicitly, using an algebraic fractional step based strategy described in (Houzeaux et al., 2011). We consider here the simulation of the airflow through the nose. The mesh is characterised by a very elongated geometry with small passages in the nasal cavity, leading to a pseudo-1D elongated domain decomposition when parallelizing through partitioning the mesh, see Figure 5.15. Due to the elongated geometry, low frequencies are hardly damped with a one-level DDM approach, motivating the use of a CSC.

The discretization of the problem leads to a coupled algebraic system to be solved at each time step. This algebraic system is split to solve independently the momentum and the continuity equations. At each time step, the momentum and continuity equations are solved repeatedly until the solution converges to the monolithic solution. Due to the splitting strategy, it is necessary to solve the momentum and the continuity equations twice per time step. A in-house Alya solver is used to solve the unsymmetric momentum equation, while the continuity equation is SPD and is solved using the MaPHyS solver. For a complete description of this test case, we refer to (Calmet et al., 2016).

5.5.3.2 Performance results with MaPHyS

The parallel benchmarks have been conducted on 264, 528, 1,056 and 2,112 CPU cores. In this configuration, we have thus employed 264, 528, 1,056 and 2,112 MPI processes, respectively. Since Alya has a dedicated "master" MPI process, these calculations were performed with 263, 527, 1,055 and 2,111 subdomains, respectively. The stopping criterion is set to 10^{-6} with a maximum of 2,000 iterations. For each experiment, 10 time steps are performed, each time step requiring to solve the linear systems twice.



(a) Total time to solve the continuity problem

(b) Global preconditioner application time

(c) Speedup in solving the continuity equation

(d) Efficiencies in solving the continuity equation

Figure 5.16: Evaluation of MaPHyS strong scaling with different CSC implementations on the respiratory test case.

The results are presented in Figure 5.16, consisting of four quadrants, showing the MaPHyS solver total time (Figure 5.16a), the global preconditioner application time for MaPHyS (Figure 5.16b), the speedups (Figure 5.16c) and the parallel efficiencies normalized with the execution on 264 processes (Figure 5.16d) of the MaPHyS solver, respectively. The five parallel strategies proposed in Section 5.4 for handling the CSC are assessed, with a number of $n_v = 2$, 3 and 5 eigenvalues per subdomain. We only report the performance for the number of eigenvalues n_v leading to the lowest total computation time, the shape of the points indicating the associated n_v value (circle, triangle and square for $n_v = 2$,

3 and 5, respectively). As the matrix changes between each time step, MaPHyS has to perform several times its factorization step in order to factorize the local interior problems and compute the local Schur complements. However, MaPHyS has been set up so that the preconditioner (both for the local and coarse components) remains fixed through the ten considered time steps. If necessary, it could be set up to be recomputed at a given frequency but that was not critical in our experiments and we do not report further on that option.

Consistently with the results presented in Section 5.5.2, SD strategy does not scale: the distributed sparse direct solver is not able to efficiently use all the available computing units without further guidance. The DC strategy does not scale well either when the size of the coarse space increases. Instead of providing the global communicator to the direct solver and letting it handle data distribution as in SD, the SC strategy centralizes the coarse matrix on a process and provides a smaller sub-communicator of size 12 to the sparse direct solver. This strategy scales well up to 1,056 processes, but its parallel efficiency drops from 90% to 50% on 2,112 processes. The hierarchical SHD strategy does not perform better. The replicated SRC strategy outperforms all the other strategies with a parallel efficiency of 70% on 2,112 processes with respect to 264 processes.

5.5.3.3 Performance results with `ddmpy`

The performance of 14 `ddmpy` solvers was also assessed on the same respiratory test case with 1,055 subdomains (Figure 5.17). Besides the preconditioners already introduced in Section 4.4.1, the AS_{G+3} preconditioner is an additive two-level AS preconditioner that uses a GenEO coarse space with 3 vectors per subdomain (see Section 3.4). These results confirm the relevance of two-level methods based on the Schur complement system. However, on this test case, using the adaptative coarse space with 3 vectors per subdomains instead of the simpler partition-of-unity coarse space does not improve significantly the convergence, as can be seen by comparing the results of the $AS_2/\mathcal{S}$ and $AS_{G+3}/\mathcal{S}$ methods for additive CSC or the $AS_D/\mathcal{S}$ and $AS_{GD3}/\mathcal{S}$ methods for deflated CSC.

Note that the deflated preconditioners in `ddmpy` are based on $\mathcal{A}$-orthogonal projections and should be recomputed if the matrix $\mathcal{A}$ changes. As a result, the best method for this airflow test case is the additive two-level AS preconditioner based on a partition-of-unity coarse space applied on the Schur complement system ($AS_2/\mathcal{S}$).

5.5.4 A plasma propulsion application using the AVIP code

5.5.4.1 Test case

Electric propulsion can reach higher exhaust velocities compared to chemical systems, resulting in lower propellant mass requirements and a significant possible saving for the space industry. Among the different electric propulsion systems, Hall effect thrusters are used for spatial propulsion since the 1970s. However inside a Hall thruster, complex physical phenomena such as erosion or electron anomalous transport which may lower thruster efficiency and lifetime, are not yet fully understood. Thanks to HPC, numerical simulations are now considered for understanding the plasma behavior. With the renewed interest for such electric propulsion to supply small satellites, parallel numerical solvers

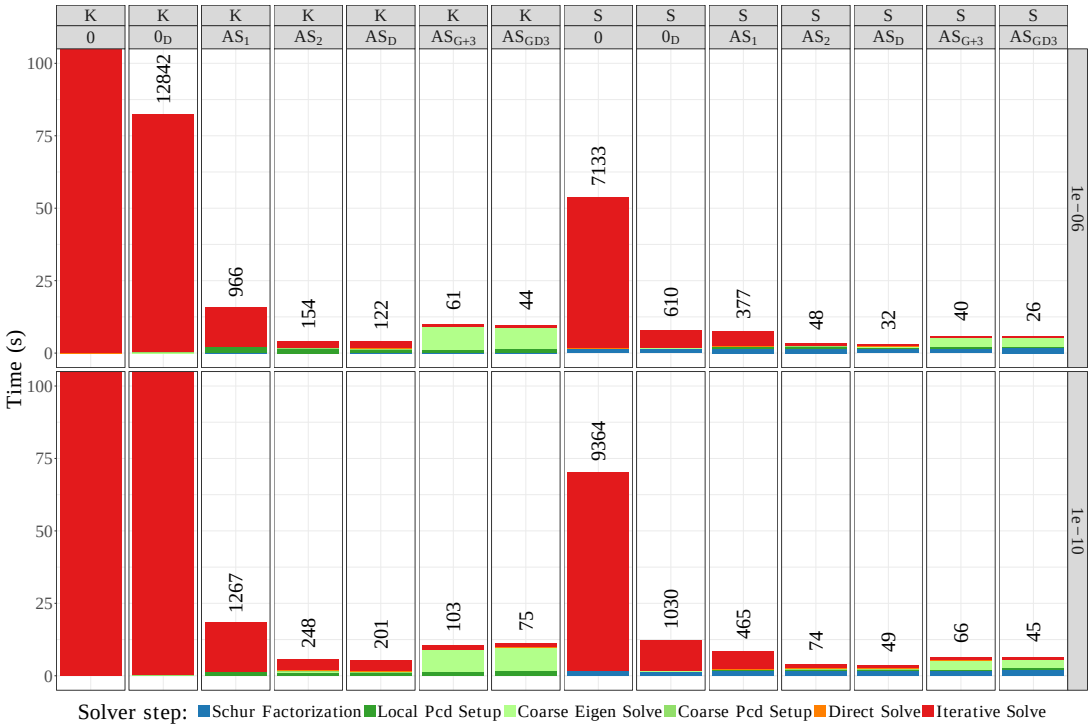


Figure 5.17: Step by step time for `ddmpy` on Alya respiratory test case (1,055 domains).

able to predict accurately the real thruster efficiency have become crucial for industry. Accuracy and reliability are of course essential and different numerical plasma models with various orders of accuracy have emerged. On the one hand, Particle-In-Cell (PIC) models follow the trajectories of a representative number of physical particles called macro-particles in the presence of electromagnetic fields. They are often coupled to a Monte Carlo (MCC) module to deal with statistical collisions and a Maxwell or Poisson solver to solve electromagnetic fields. On the other hand, fluid models describe the macroscopic behavior of each population of particles.

The design and development of the AVIP code is an on-going joint effort between CERFACS and the Laboratoire de Physique des Plasmas at École Polytechnique. The AVIP code solves the plasma equations in complex industrial geometries using 3D unstructured parallel efficient schemes based either a PIC approach or a fluid methodology. While full 3D PIC simulations of a Hall thruster still require very high computational resources, fluid models provide 3D results on the plasma behavior inside the discharge channel in a reasonable time . On the other hand the PIC approach is more accurate and does not rely on additional hypotheses as the fluid model does. Finally, the electric potential is resolved via the solution of a Poisson equation. For more details on AVIP and the selected models we refer to (Joncquieres et al., 2018).

When the plasma is simulated using the PIC model, highly computing intensive calculation on the particles are involved that further parallelizes naturally. Consequently, the solution of the linear systems resulting from the discretization of the Poisson equation becomes rapidly a bottleneck on large computing platforms. Its solution requires the use of a scalable linear solver in the strong sense; that is, that remains effective when the

number of unknowns per CPU core shrinks while the number of cores increases to keep the PIC calculation affordable. Figure 5.18 shows the strong scalability experiments performed on a mesh with 12 million cells using PRACE Tier-0 computing resource, leading to a linear system with about 6 million unknowns for the Poisson system, and 2.5 million particles. For those experiments, the linear system is solved with the "best" PETSc solver selected by the AVIP team after many experiments with various solvers available in PETSc (Balay et al., 2018) (it corresponds to a MinRes with local SOR preconditioner). It can be seen that most of the numerical kernels involved in the PIC calculation scale almost perfectly (sorting, MCC, interpolation, transport) while the time to solution for the Poisson problem increases for a number of cores larger that 2,000; that is, when the number of unknowns per core becomes lower that 3,000. Figure 5.19 shows a strong scalability comparison between the PETSc solver and MaPHyS (without CSC, as this feature was not available at the time the collaboration between the AVIP and MaPHyS teams started), still on a PRACE Tier-0 machine. We can first observe that MaPHyS performs better than PETSc on this half million problem and that MaPHyS also exhibits strong scalability issues when the number of unknowns per core becomes smaller than 1,000 (daily calculations of AVIP use a few thousands of unknowns per core). Although not described here, not only did MaPHyS outperform PETSc in terms of time to solution, but MaPHyS was also much more robust to the heterogeneity in the mesh size of the considered test case. This heterogeneity translates into heterogeneity in the linear system entries, which often prevents the PETSc solver to converge to the targeted accuracy. Those results and observations motivated the integration of MaPHyS in AVIP and the complementary performance study presented in the next section.

5.5.4.2 Performance results with MaPHyS

Since each iteration in AVIP requires a solution with the same matrix, the setup step is performed only once and its cost becomes negligible in comparison with the accumulated time of solving for a different right-hand side at each iteration. As a result, the performance of the Poisson solver in AVIP is governed by the performance of the solve step.

The performance of the five CSC strategies applied on the AVIP test case with $n_v = 3$, 5 and 10 eigenvectors per subdomain are presented in Table 5.2 and Figure 5.20. Using a tolerance of 10^{-10} as a threshold for the stopping criterion in MaPHyS (as needed for AVIP simulations), the best performance is achieved using the SC(24) strategy with a coarse space of $n_v = 3$ eigenvectors per subdomain. The solve time with this CSC strategy is 4.004 s. which is an improvement of 40% from the solve time using the baseline one-level version of MaPHyS (6.831 s.). The four other strategies have a solve step that is about 25% longer than SC.

Within a PRACE Tier-0 project the AVIP team had been allocated 30 million core-hours; most of the simulations were performed using MaPHyS without the CSC feature presented in this chapter. If the CSC implementation in MaPHyS had been available at the beginning of the PRACE Tier-0 project, extrapolating the MaPHyS performance displayed in Table 5.2, the use of AS_{G+3}/S instead of AS_1/S would have allowed to save about 6 million core-hours (a fifth of the total allocation), if all the runs had been

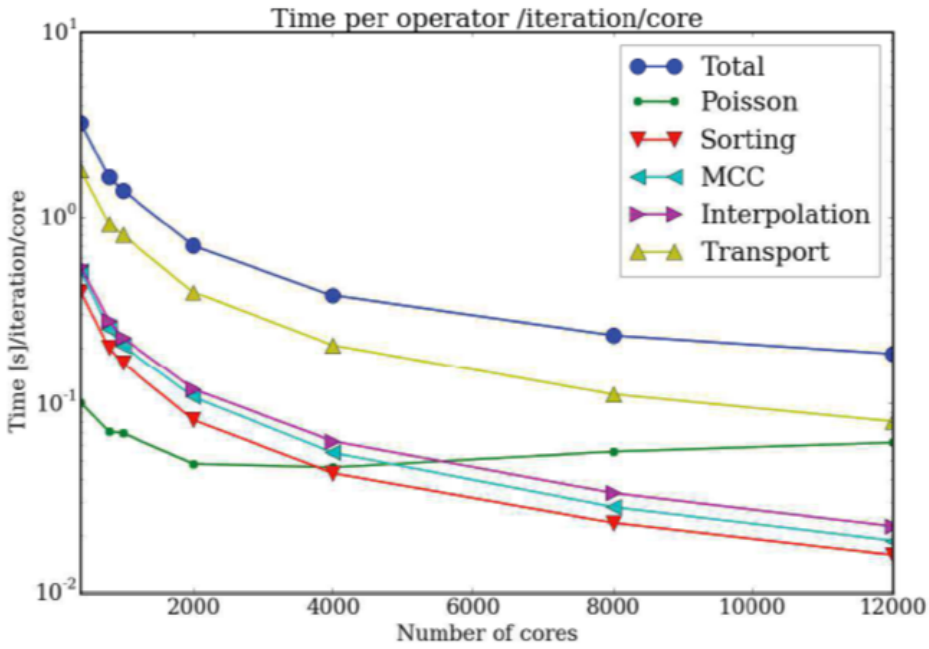


Figure 5.18: Parallel performance of the AVIP code on a mesh with 12 million elements and 2.5 million particles

performed on about a thousand of cores where the solution of the Poisson equation roughly corresponds to half of the time to solution (the additional setup cost is negligible due to the number of successive solves). This interpolation is computed based on the performance observed with a stopping criterion threshold equal to 10^{-10}, that covers most of the AVIP calculation. This simple extrapolation exercise highlights how crucial is the choice of efficient parallel linear solvers in computational sciences and the continuous need to design numerically and computationally efficient parallel solvers.

5.5.4.3 Performance results with `ddmpy`

The performance of 13 `ddmpy` solvers was also assessed on the same test case and is reported in Figure 5.21 and Table 5.3. Unfortunately, the computation of the GenEO coarse space for the Schur complement system on that test case resulted in an error (not further investigated at the time of writing this thesis), and the preconditioners AS_{G+3}, AS_{GD3} and NN_{G+3} could therefore only be applied on $\mathcal{K}$ and not on the interface system $\mathcal{S}$.

For a tolerance of 10^{-10}, the best solver regarding the total time to solution is a deflated AS preconditioner based on a partition-of-unity coarse space ($AS_D/\mathcal{K}$). The solver with the fastest solve step is $AS_{GD3}/\mathcal{K}$ solver, which is 4 times faster that the baseline $AS_1/\mathcal{S}$

Table 5.2: MaPHyS coarse space correction implementations tested on the AVIP test case (1,008 domains). The Setup, Solve and Total times are the max among all subdomains, in seconds (s). Each subdomain is associated with one MPI process binded on one CPU core.

		Tolerance 10^{-6}				Tolerance 10^{-10}			
	n_v	# it.	*Setup*	*Solve*	*Total*	# it.	*Setup*	*Solve*	*Total*
No csc	*0*	*15*	11.37	1.672	13.06	*64*	11.17	6.831	18.03
DC(1)	*3*	*5*	72.35	0.892	73.26	*27*	74.10	5.016	79.14
SD(1008)	*3*	*5*	77.31	0.939	78.26	*27*	78.43	4.975	83.43
SC(24)	*3*	*5*	72.57	0.874	73.47	*27*	74.09	4.004	78.12
SRC(24)	*3*	*5*	72.19	1.037	73.25	*27*	73.83	5.264	79.12
SHD(42)	*3*	*5*	73.44	1.077	74.53	*27*	74.15	5.137	79.33
DC(1)	*5*	*5*	74.91	0.910	75.84	*25*	75.44	4.868	80.33
SD(1008)	*5*	*5*	88.03	1.134	89.19	*25*	87.98	5.247	93.24
SC(24)	*5*	*5*	72.43	1.170	73.63	*25*	74.08	4.755	78.85
SRC(24)	*5*	*5*	74.17	0.938	75.13	*25*	73.48	4.618	78.12
SHD(42)	*5*	*5*	74.24	0.985	75.24	*25*	74.64	4.086	78.75
DC(1)	*10*	*5*	86.76	1.520	88.31	*23*	86.44	6.624	93.11
SD(1008)	*10*	*5*	94.54	1.178	95.74	*23*	93.07	4.851	97.94
SC(24)	*10*	*5*	78.30	0.837	79.17	*23*	77.39	4.383	81.82
SRC(24)	*10*	*5*	76.06	1.111	77.19	*23*	76.58	4.359	80.95
SHD(42)	*10*	*5*	74.77	1.065	75.86	*23*	76.52	4.538	81.08

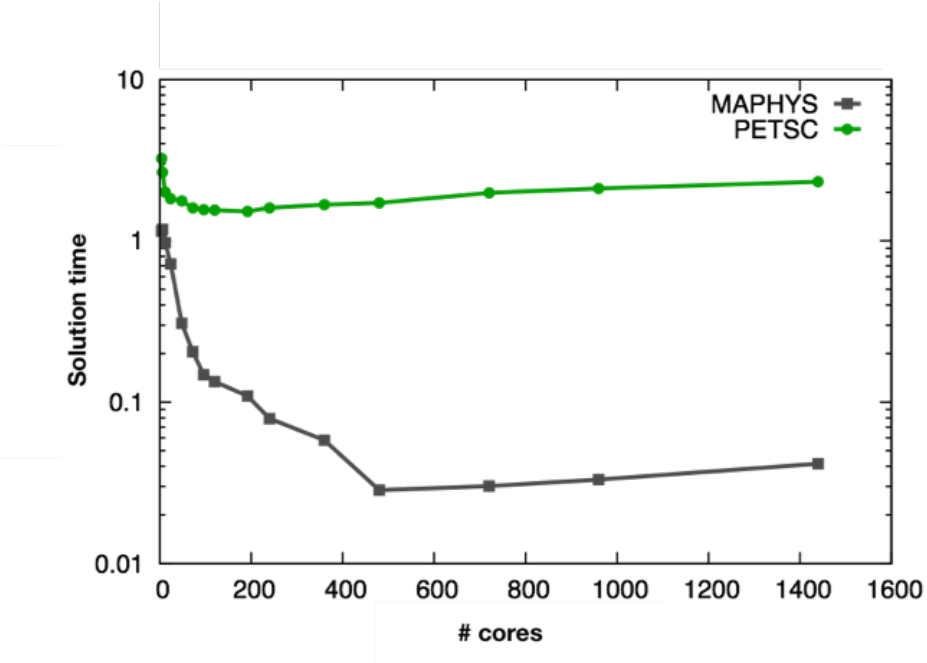


Figure 5.19: Strong scalability comparison between the AVIP-PETSc solver and Maphys on a 500,000 problem size

MaPHyS solver. Based on the same interpolation as in the previous section, using the $AS_{GD3}/\mathcal{K}$ `ddmpy` solver would have saved more than 11 million core-hours (a third of the total allocation).

5.6 Conclusion

In this chapter, we introduced and compared five parallel strategies for the coarse space correction presented in Chapter 3. After the algorithms corresponding to these strategies are exposed, they are tested on 3 different test cases, including a heterogeneous diffusion problem with up to one billion unknowns on 16,128 cores.

These parallel strategies for two-level hybrid solvers are not restricted to the computation of coarse space correction. They can be applied for any application where a sparse linear system needs to be factorized and solved with a very small granularity (only a few unknowns per MPI process). Our experiments show that distributed sparse direct solvers should not be expected to handle well such extreme data distributions. On the contrary, the corresponding *Sparse Distributed* strategy was always one of the two least scalable strategies.

Good parallel performance is achieved by redistributing the data and handling the

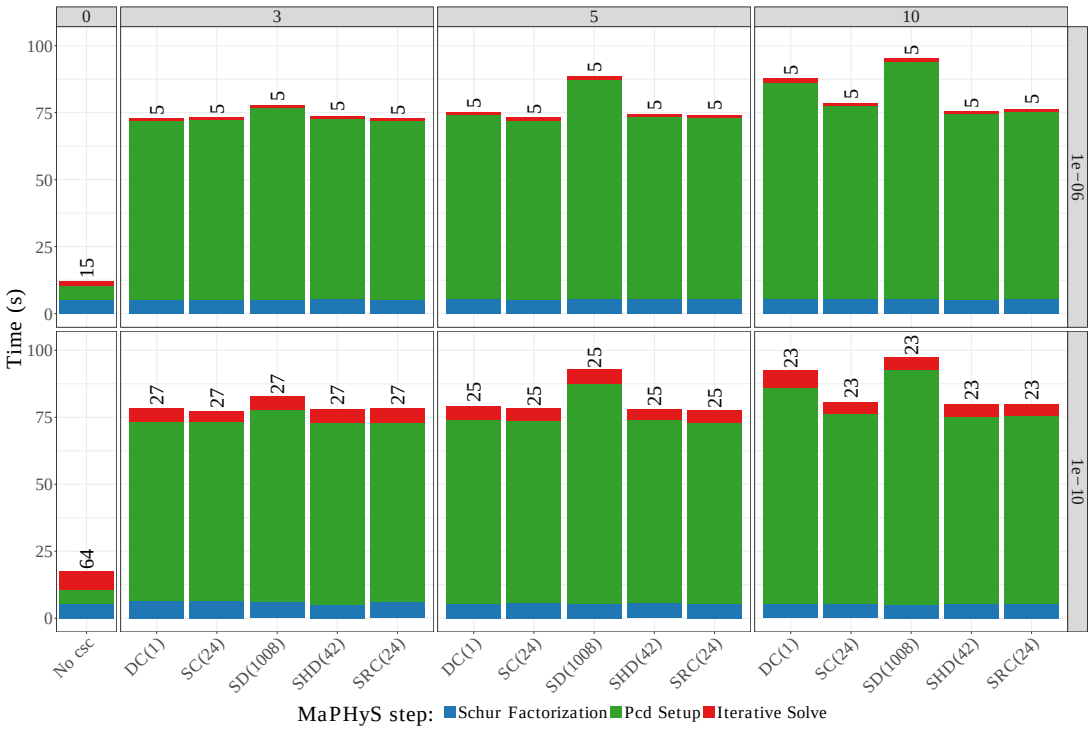


Figure 5.20: Step by step time for MaPHyS on AVIP test case

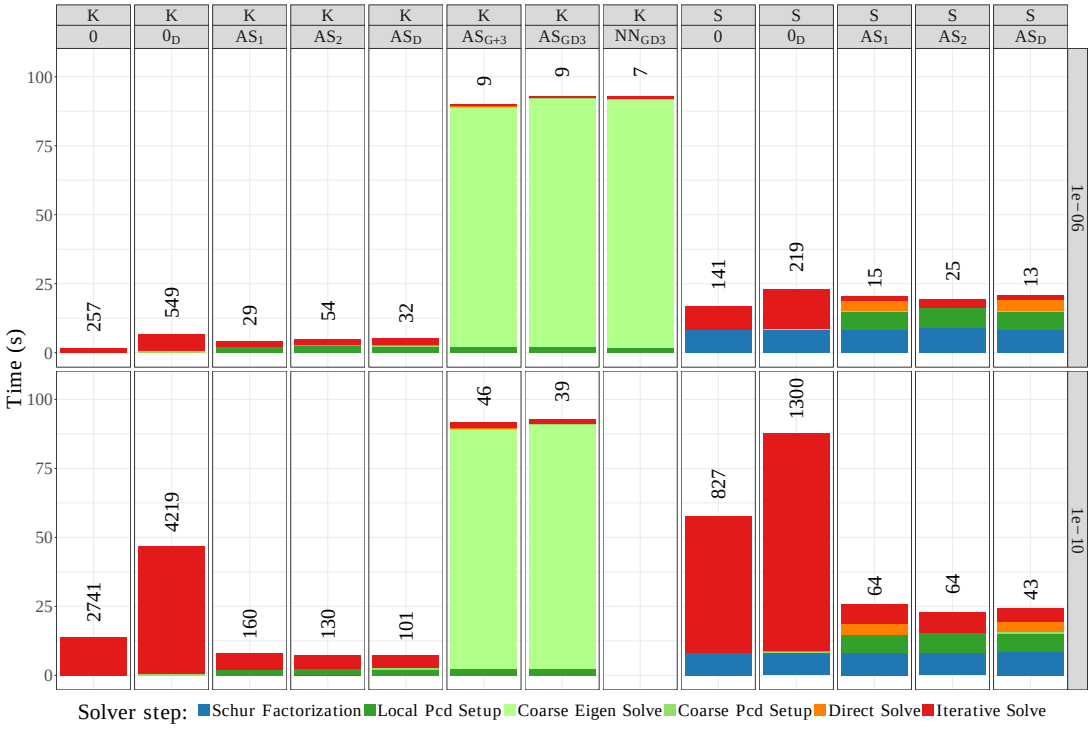


Figure 5.21: Step by step time for `ddmpy` on AVIP test case

Table 5.3: Various `ddmpy` solvers tested on the AVIP test case (1008 domains). The Setup, Solve and Total times are the max among all subdomains, in seconds (s). Each subdomain is associated with one MPI process binded on one CPU core. The best times obtained with MaPHyS are given below as a reference.

		Tolerance 10^{-6}				Tolerance 10^{-10}			
Solver	Method	# it.	*Setup*	*Solve*	*Total*	# it.	*Setup*	*Solve*	*Total*
`ddmpy`	$0/\mathcal{K}$	*257*	0	1.509	1.509	*2741*	0	13.919	13.919
	$0_D/\mathcal{K}$	*549*	0.624	6.096	6.707	*4219*	0.510	46.124	46.626
	$AS_1/\mathcal{K}$	*29*	2.196	1.996	3.237	*160*	2.194	5.942	7.147
	$AS_2/\mathcal{K}$	*54*	2.757	1.969	4.715	*130*	2.475	4.893	7.292
	$AS_D/\mathcal{K}$	*32*	2.736	2.332	4.031	*101*	2.668	4.656	6.277
	$AS_{G+3}/\mathcal{K}$	*9*	89.173	0.483	89.604	*46*	89.180	1.974	91.147
	$AS_{GD3}/\mathcal{K}$	*9*	92.284	0.547	92.773	*39*	90.973	1.652	92.610
	$NN_{GD3}/\mathcal{K}$	*7*	91.975	0.735	92.683	n.c.			
	$0/\mathcal{S}$	*141*	8.292	8.595	16.882	*827*	8.235	49.740	57.970
	$0_D/\mathcal{S}$	*219*	8.692	14.476	23.157	*1300*	8.717	78.923	87.621
	$AS_1/\mathcal{S}$	*15*	15.103	7.158	16.912	*64*	14.841	12.515	22.284
	$AS_2/\mathcal{S}$	*25*	16.307	2.971	19.257	*64*	15.396	7.432	22.808
	$AS_D/\mathcal{S}$	*13*	15.361	7.126	17.170	*43*	15.735	10.443	20.860
MaPHyS	AS_1/S	*15*	11.37	1.672	13.06	*64*	11.17	6.831	18.03
	AS_{G+3}/S	*5*	78.30	0.837	79.17	*27*	74.09	4.004	78.12

coarse problem on a sub-communicator of the main communicator. This way, the size of the sub-communicator can be adjusted such that the granularity of the coarse problem is optimal with regards to the sparse direct solver in use. Depending on the test case, different strategies following this principle achieve the best performance.

This optimized two-level version of MaPHyS in Fortran 90 was compared with solvers provided by the `ddmpy` toolbox exposed in Chapter 4. Although, for any particular method, the Fortran implementation is faster than the python implementation, a greater range of methods were implemented in python thanks to the language expressivity. The flexibility offered by the `ddmpy` toolbox makes up for the difference in term of pure performance, as it is easier to adjust the method for a particular test case.

Because there is no support for *scatter*-like MPI collective handling overlapping regions in the MPI standard, we instead broadcast z_0. For ensuring further scaling, one option would be to design a two-step scheme consisting of a *scatter*-like MPI collective communication followed by neighbor-to-neighbor communications. In any case, while we have considered the opportunity to provide the CSC matrix $\mathcal{S}_0$ in distributed input mode (SD and SHD schemes), we considered only a centralized input mode for handling the right-hand side r_0 and the solution z_0. Indeed, the Mumps solver, onto which we rely to handle the CSC in a parallel distributed context, did not provide a distributed input mode for those vectors. This option is now available and we plan to investigate it in the future work.

A new implementation in modern C++ is underway as mentioned in Section 6.3. Some recent features of this language should make it possible to combine the flexibility of

modern high-level languages (such as python) with the robust performance of compiled languages (such as Fortran).

Chapter 6

Conclusion and perspectives

6.1 Conclusion

In this thesis manuscript, we investigated different ways to design algebraic hybrid solvers that use domain decomposition ideas and principles. In Chapter 2, we proposed an overview of classic domain decomposition methods in a continuous setting as well as their algebraic counterparts. Then, in Chapter 3, we analyzed the convergence of a subset of these methods known as two-level abstract Schwarz methods for symmetric positive definite (SPD) problems and proposed a new coarse space for these methods based on the GenEO methodology (Nataf et al., 2011; Spillane and Rixen, 2013; Spillane et al., 2014a) such that the convergence rate is independent of the number of subdomains and the difficulty of the original problem. Interestingly, the proposed methodology turns out to be a comparison of the considered particular abstract Schwarz method with generalized versions of both Neumann-Neumann and Additive Schwarz for tackling the lower and upper part of the spectrum, respectively. We furthermore showed that the application of the considered coarse space corrections in an additive fashion is robust in the Additive Schwarz case although it is not robust for abstract Schwarz methods in general. In Chapter 4, the relevance of python as a language for designing HPC domain decomposition solvers has been assessed and a new parallel toolbox `ddmpy` was presented using a literate programming approach for reproducible science. Its flexibility allowed us to design, study and compare a large spectrum of solvers. In Chapter 5, we considered the fully-featured MaPHyS solver, implementing an Additive Schwarz on the Schur. We incorporated a coarse space correction in an additive fashion (shown to be robust in the Additive Schwarz case in Chapter 3). We designed multiple parallel strategies whose parallel efficiencies were assessed thanks to the integration of MaPHyS in two large application codes, namely, Alya developed at BSC and AVIP jointly developed by CERFACS and Laboratoire de Physique des Plasmas at École Polytechnique. For this latter code, part of the parallel experiments were conduced on a PRACE Tier-0 machine within a PRACE Project Access.

This work opens new challenges and perspectives that are detailed in the following sections:

- the robust preconditioners presented in Chapter 3 can only be applied on SPD problems and two different ideas for solving general (non SPD) problems are the

following:

  - use a N-Lagrange formulation with a transmission matrix $\mathcal{T}_i$ supplied by the application alongside the local matrix $\mathcal{K}_i$. With a well chosen transmission matrix, no coarse space is needed to ensure good convergence (Japhet, 1998)
  - use a multipreconditioned approach (Bridson and Greif, 2006; Greif et al., 2011; Spillane, 2016) to enrich the search space of iterative methods during the iterations instead of building a coarse space beforehand.

- the implementations presented in chapters 4 and 5 use the Fortran 90 and python languages, respectively. Whereas the python language brings agility, flexibility and expressivity without compromising on the performance, we experienced some difficulties for deploying the python ecosystem on clusters for large-scale experiments. Although we solved the encountered issues, it results in a fragile environment. This led us to consider of rewriting of both the MaPHyS solver and the `ddmpy` toolbox in C++ in order to benefit from the strengths of both high-level languages (such as python) and compiled languages (such as Fortran). The new library will be available at `https://gitlab.inria.fr/solverstack/maphys/maphyspp`.

6.2 Robust methods for general (non SPD) problems

6.2.1 Algebraic interface for the N-Lagrange and for the Optimized Schwarz methods

The N-Lagrange method (Nataf et al., 1995) introduced in sections 2.5.7 and 2.6.5 is a method of choice for unsymmetric problems (Gander et al., 2002; El Bouajaji et al., 2015). Both the N-Lagrange and the Optimized Restricted Additive Schwarz (St-Cyr et al., 2007) methods are based on a parallel Schwarz method with Robin transmission conditions as introduced by Lions (1988, 1989, 1990). The algebraic N-Lagrange method and the ORAS method can be applied algebraically as long as a transmission matrix $\mathcal{T}_i$ is provided to the solver alongside the subdomain matrix $\mathcal{K}_i$. The N-Lagrange method is implemented (Listing 57), and a symmetric version of ORAS is implemented as a Robin-Robin preconditioner (Listing 62) in `ddmpy` and also available in the latest releases of MaPHyS. They could however not be tested with an optimized transmission matrix for lack of a test case: discretizing a transmission condition was outside the scope of the work presented in this thesis, and we were not yet able to obtain it from any of the applications that use MaPHyS as an internal solver. In an on-going work in collaboration between the Inria HiePACS and Nachos teams, Cristobal Samaniego Alvarado is comparing the Robin-Robin preconditioner in MaPHyS with the internal N-Lagrange solver considered in (Li et al., 2014).

6.2.2 Multipreconditioned solvers

Another promising method for improving the convergence of an algebraic DD solver is to use multipreconditioning (Bridson and Greif, 2006; Greif et al., 2011). Spillane (2016)

showed that in the SPD case, this method could be used to generate a coarse space for abstract Schwarz preconditioners during the CG iterations, eliminating the need for the costly solution of the generalized eigenproblems in the GenEO coarse space. After a sufficient number of these multipreconditioned iterations, one can automatically switch to regular deflated PCG iterations.

Multipreconditioning in the context of aS methods is akin to a form of optimized variable preconditioning: the aS preconditioner

$$\mathcal{M}_{aS} = \sum_{i=1}^{N} \mathcal{R}_i^T \widehat{\mathcal{A}}_i^{\dagger} \mathcal{R}_i$$

is replaced by

$$\mathcal{M}_{aS}^{\mu} = \sum_{i=1}^{N} \mu_i \mathcal{R}_i^T \widehat{\mathcal{A}}_i^{\dagger} \mathcal{R}_i$$

where the vector $\mu = (\mu_1, \ldots, \mu_N)^T$ is chosen at each iteration as to minimize a norm of the error.

6.3 C++ implementation

The methods introduced in chapters 2 and 3 were implemented in chapters 4 in python and 5 in Fortran 90, respectively.

Although it is much easier and more pleasant to develop and maintain a module in python, we encountered some challenges using this language, as explained in Section 4.5. This led us to initiate the development of a new library[1] in C++. Most of the structures presented in Section 4.3.2 for python have C++ equivalents. Most notably, it is possible in C++20 to use *templates* and *concepts* to have duck-typing at compilation. Using the boost.python library (Abrahams and Grosse-Kunstleve, 2003) or the SWIG interface compiler (Beazley, 1996), this new library will be available from `ddmpy`.

[1] `https://gitlab.inria.fr/solverstack/maphys/maphyspp`

Chapter 7

Acknowledgments

The work presented in this document was performed in very close collaboration with my supervisors Luc Giraud and Emmanuel Agullo.

We would like to thank Nicole Spillane for proofreading an early draft of Chapter 3. This chapter was submitted for publication and we also thank the anonymous referees whose constructive comments enabled us to significantly improve the manuscript. The experiments presented in sections 3.5.2 to 3.5.4 were carried out using the PLAFRIM experimental testbed, being developed under the Inria PlaFRIM development action with support from Bordeaux INP, LABRI and IMB and other entities: Conseil Régional d'Aquitaine, Université de Bordeaux and CNRS (and ANR in accordance to the programme d'investissements d'Avenir).

The experiments in Section 4 were performed on the GENCI Occigen cluster at CINES. Part of the Chapter 5 is extracted from an article in preparation, written in collaboration with Emmanuel Agullo, Matthieu Kuhn, Gilles Marait and Luc Giraud; the experiments in this chapter were also carried out on the Occigen cluster, by Matthieu Kuhn and Gilles Marait. We would also like to thank the AVIP team at CERFACS for close scientific exchanges and collaborations as well as the opportunity we had to access PRACE Tier-0 computing resource; we acknowledge PRACE for awarding us access to Curie at GENCI@CEA, France. We also thank the Alya team at BSC for the fruitful collaboration and access to the BSC computing facilities.

Bibliography

Abrahams, D. and Grosse-Kunstleve, R. W. (2003). Building hybrid systems with Boost. Python. *CC Plus Plus Users Journal*, 21(7):29–36.

Achdou, Y., Le Tallec, P., Nataf, F., and Vidrascu, M. (2000). A domain decomposition preconditioner for an advection–diffusion problem. *Computer Methods in Applied Mechanics and Engineering*, 184(2):145–170.

Achdou, Y. and Nataf, F. (1997). A Robin-Robin preconditioner for an advection-diffusion problem. *Comptes Rendus de l'Académie des Sciences - Series I - Mathematics*, 325(11):1211–1216.

Agullo, E., Darve, E., Giraud, L., and Harness, Y. (2018). Low-rank Factorizations in Data Sparse Hierarchical Algorithms for Preconditioning Symmetric Positive Definite Matrices. Report, Inria Bordeaux Sud-Ouest.

Agullo, E., Giraud, L., Guermouche, A., and Roman, J. (2011). Parallel hierarchical hybrid linear solvers for emerging computing platforms. *Comptes Rendus Mécanique*, 339(2):96–103.

Agullo, E., Giraud, L., Guermouche, A., Roman, J., and Zounon, M. (2013). *Towards Resilient Parallel Linear Krylov Solvers: Recover-Restart Strategies*. PhD Thesis, INRIA.

Agullo, E., Giraud, L., and Nakov, S. (2016a). Task-based sparse hybrid linear solver for distributed memory heterogeneous architectures.

Agullo, E., Giraud, L., Nakov, S., and Roman, J. (2016b). Hierarchical hybrid sparse linear solver for multicore platforms. Report, INRIA Bordeaux.

Agullo, E., Giraud, L., and Yetkin, E. F. (2017). Soft Error in PCG: Sensitivity, Numerical Detections and Possible Recoveries. In *SIAM Conference on Computational Science and Engineering, CSE'17*.

Agullo, E., Kuhn, M., Lanteri, S., and Moya, L. (2016c). High order scalable HDG method for frequency-domain electromagnetics.

Amestoy, P. R., Duff, I. S., L'Excellent, J.-Y., and Koster, J. (2001). A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41.

Anderson, E., Bai, Z., Bischof, C., Blackford, L. S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., and McKenney, A. (1999). *LAPACK Users' Guide*. SIAM.

Balay, S., Abhyankar, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., Dalcin, L., Dener, A., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., May, D. A., McInnes, L. C., Mills, R. T., Munson, T., Rupp, K., Sanan, P., Smith, B. F., Zampini, S., Zhang, H., and Zhang, H. (2018). PETSc Web page.

Barrett, R., Berry, M. W., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and Van der Vorst, H. (1994). *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, volume 43. Siam.

Beazley, D. M. (1996). SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++. In *Tcl/Tk Workshop*.

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S., and Smith, K. (2011). Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39.

Bourgat, J.-F., Glowinski, R., Le Tallec, P., and Vidrascu, M. (1989). Variational Formulation and Algorithm for Trace Operator in Domain Decomposition Calculations. In *Proceedings of the Second International Conference on Domain Decomposition Methods in Los Angeles, California (January 14-16 1988)*. Siam.

Bramble, J. H., Pasciak, J. E., and Schatz, A. H. (1986). The construction of preconditioners for elliptic problems by substructuring. I. *Mathematics of Computation*, 47(175):103–134.

Bramble, J. H., Pasciak, J. E., and Schatz, A. H. (1989). The Construction of Preconditioners for Elliptic Problems by Substructuring, IV. *Mathematics of Computation*, 53(187):1.

Brenner, S. C. (1999). The condition number of the Schur complement in domain decomposition. *Numerische Mathematik*, 83(2):187–203.

Bridson, R. and Greif, C. (2006). A multipreconditioned conjugate gradient algorithm. *SIAM Journal on Matrix Analysis and Applications*, 27(4):1056–1068.

Burkitt, A. N. and Irving, A. C. (1988). Identity of the conjugate gradient and Lanczos algorithms for matrix inversion in lattice fermion calculations. *Physics Letters B*, 205(1):69–72.

Cai, X.-C. and Sarkis, M. (1999). A restricted additive Schwarz preconditioner for general sparse linear systems. *Siam journal on scientific computing*, 21(2):792–797.

Calmet, H., Gambaruto, A. M., Bates, A. J., Vázquez, M., Houzeaux, G., and Doorly, D. J. (2016). Large-scale CFD simulations of the transitional and turbulent regime for the large human airways during rapid inhalation. *Computers in biology and medicine*, 69:166–180.

Carvalho, L., Giraud, L., and Le Tallec, P. (2001a). Algebraic Two-Level Preconditioners for the Schur Complement Method. *SIAM Journal on Scientific Computing*, 22(6):1987–2005.

Carvalho, L. M., Giraud, L., and Meurant, G. (2001b). Local preconditioners for two-level non-overlapping domain decomposition methods. *Numerical linear algebra with applications*, 8(4):207–227.

Casadei, A., Giraud, L., Ramet, P., and Roman, J. (2013). Towards domain decomposition with balanced halo. In *Workshop Celebrating 40 Years of Nested Dissection*.

Chaitin-Chatelin, F. and Frayssé, V. (1996). *Lectures on Finite Precision Computations*. SIAM, Philadelphia.

Chan, T. F. and Mathew, T. P. (1992). The interface probing technique in domain decomposition. *SIAM Journal on Matrix Analysis and Applications*, 13:212–238.

Chevalier, C. and Pellegrini, F. (2008). PT-Scotch: A tool for efficient parallel graph ordering. *Parallel computing*, 34(6-8):318–331.

Cockburn, B., Gopalakrishnan, J., and Lazarov, R. (2009). Unified hybridization of discontinuous Galerkin, mixed, and continuous Galerkin methods for second order elliptic problems. *SIAM Journal on Numerical Analysis*, 47(2):1319–1365.

Dalcin, L. D., Paz, R. R., Kler, P. A., and Cosimo, A. (2011). Parallel distributed computing using Python. *Advances in Water Resources*, 34(9):1124–1139.

Davis, T. A. (2008). User Guide for CHOLMOD: A sparse Cholesky factorization and modification package. *Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, USA*.

De Roeck, Y.-H. and Le Tallec, P. (1991). Analysis and test of a local domain decomposition preconditioner. In *Fourth International Symposium on Domain Decomposition Methods for Partial Differential Equations*, volume 4.

Demmel, J. W., Eisenstat, S. C., Gilbert, J. R., Li, X. S., and Liu, J. W. (1999). A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755.

Dohrmann, C. R. (2003). A preconditioner for substructuring based on constrained energy minimization. *SIAM Journal on Scientific Computing*, 25(1):246–258.

Dolean, V., Gander, M. J., Lanteri, S., Lee, J.-F., and Peng, Z. (2015a). Effective transmission conditions for domain decomposition methods applied to the time-harmonic curl–curl Maxwell's equations. *Journal of computational physics*, 280:232–247.

Dolean, V., Jolivet, P., and Nataf, F. (2015b). *An Introduction to Domain Decomposition Methods: Algorithms, Theory, and Parallel Implementation*, volume 144. SIAM.

Dominik, C. (2010). *The Org Mode 7 Reference Manual-Organize Your Life with GNU Emacs*. Network Theory Ltd.

Dryja, M. and Widlund, O. (1987). An Additive Variant of the Schwarz Alternating Method for the Case of Many Subregions. Technical Report 339, also Ultracomputer Note 131, Department of Computer Science, Courant Institute.

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

Duff, I. S., Erisman, A. M., and Reid, J. K. (2017). *Direct Methods for Sparse Matrices*. Oxford University Press.

Efendiev, Y., Galvis, J., Lazarov, R., and Willems, J. (2012). Robust domain decomposition preconditioners for abstract symmetric positive definite bilinear forms. *ESAIM: Mathematical Modelling and Numerical Analysis*, 46(5):1175–1199.

Eisenstat, S. C., Elman, H. C., and Schultz, M. H. (1983). Variational iterative methods for nonsymmetric systems of linear equations. *SIAM Journal on Numerical Analysis*, 20(2):345–357.

El Bouajaji, M., Dolean, V., Gander, M. J., Lanteri, S., and Perrussel, R. (2015). Discontinuous Galerkin discretizations of Optimized Schwarz methods for solving the time-harmonic Maxwell equations. *Electronic Transactions on Numerical Analysis*, 44:572–592.

Enkovaara, J. and Louhivuori, M. (2017). Scalable-python: Python interpreter for massively parallel HPC systems. CSC - IT Center for Science.

Farhat, C., Chen, P.-S., and Roux, F.-X. (1993). The dual Schur complement method with well-posed local Neumann problems: Regularization with a perturbed Lagrangian formulation. *SIAM Journal on Scientific Computing*, 14(3):752–759.

Farhat, C., Lesoinne, M., LeTallec, P., Pierson, K., and Rixen, D. (2001). FETI-DP: A dual–primal unified FETI method—part I: A faster alternative to the two-level FETI method. *International journal for numerical methods in engineering*, 50(7):1523–1544.

Farhat, C. and Roux, F.-X. (1991). A method of finite element tearing and interconnecting and its parallel solution algorithm. *International Journal for Numerical Methods in Engineering*, 32(6):1205–1227.

Feng, Y. (2018). Mpiimport: Python Import with MPI.

Feng, Y. and Hand, N. (2016). Launching Python Applications on Peta-scale Massively Parallel Systems. pages 137–143.

Fourier, J. (1822). *Theorie Analytique de La Chaleur, Par M. Fourier*. Chez Firmin Didot, père et fils.

Frayssé, V. and Giraud, L. (2000). A set of conjugate gradient routines for real and complex arithmetics. *CERFACS Technical Report TR/PA/00/47*.

Frings, W., Ahn, D. H., LeGendre, M., Gamblin, T., de Supinski, B. R., and Wolf, F. (2013). Massively Parallel Loading. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 389–398, New York, NY, USA. ACM.

Frings, W., Schnurpfeil, A., Meier, S., Janetzko, F., and Arnold, L. (2010). A flexible, application-and platform-independent environment for benchmarking. *Parallel Computing: From Multicores and GPU's to Petascale*, 19:423.

Gaidamour, J. and Hénon, P. (2008). HIPS: A parallel hybrid direct/iterative solver based on a Schur complement approach. *Proceedings of PMAA*.

Gaidamour, J. and Hénon, P. (2008). A parallel direct/iterative solver based on a Schur complement approach. *2013 IEEE 16th International Conference on Computational Science and Engineering*, 0:98–105.

Galerkin, B. G. (1915). Series solution of some problems of elastic equilibrium of rods and plates. *Vestn. Inzh. Tekh*, 19:897–908.

Galvis, J. and Efendiev, Y. (2010). Domain Decomposition Preconditioners for Multiscale Flows in High-Contrast Media. *Multiscale Modeling & Simulation*, 8(4):1461–1483.

Gander, M., Halpern, L., Magoulès, F., and Roux, F.-X. (2007). Analysis of Patch Substructuring Methods. *International Journal of Applied Mathematics and Computer Science*, 17(3):395–402.

Gander, M. J. (2006). Optimized Schwarz methods. *SIAM Journal on Numerical Analysis*, 44(2):699–731.

Gander, M. J. (2008). Schwarz methods over the course of time. *Electron. Trans. Numer. Anal*, 31(5):228–255.

Gander, M. J., Magoules, F., and Nataf, F. (2002). Optimized Schwarz methods without overlap for the Helmholtz equation. *SIAM Journal on Scientific Computing*, 24(1):38–60.

Gander, M. J. and Wanner, G. (2012). From Euler, Ritz, and Galerkin to Modern Computing. *SIAM Review*, 54(4):627–666.

Gander, M. J. and Wanner, G. (2014). The origins of the alternating Schwarz method. In *Domain Decomposition Methods in Science and Engineering XXI*, pages 487–495. Springer.

Gauss, C. F. (1811). Disquisitio de elementis ellipticis Palladis ex oppositionibus annorum 1803, 1804, 1805, 1807, 1808, 1809. *Commentationes societatis regiae scientiarum Gottingensis recentiores*, 1:1–24.

Giraud, L., Haidar, A., and Watson, L. T. (2008). Parallel scalability study of hybrid preconditioners in three dimensions. *Parallel Computing*, 34(6):363–379.

Giraud, L., Ruiz, D., and Touhami, A. (2006). A comparative study of iterative solvers exploiting spectral information for SPD systems. *SIAM J. Scientific Computing*, 27(5):1760–1786.

Glowinski, R., Golub, G. H., Meurant, G. A., and Périaux, J. (1988). *Proceedings of the First International Conference on Domain Decomposition Methods in Paris, France (January 7-9 1987)*. Siam.

Golub, G. H. and Van Loan, C. F. (1996). *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA.

Grasedyck, L. and Hackbusch, W. (2003). Construction and Arithmetics of H-Matrices. *Computing*, 70(4):295–334.

Grcar, J. F. (2011). How ordinary elimination became Gaussian elimination. *Historia Mathematica*, 38(2):163–218.

Greif, C., Rees, T., and Szyld, D. B. (2011). Multi-preconditioned GMRES. *Technical report: UBC CS TR-2011–12*.

Guelton, S., Brunet, P., Amini, M., Merlini, A., Corbillon, X., and Raynaud, A. (2015). Pythran: Enabling static optimization of scientific python programs. *Computational Science & Discovery*, 8(1):014001.

Haferssas, R., Jolivet, P., and Nataf, F. (2017). An Additive Schwarz Method Type Theory for Lions's Algorithm and a Symmetrized Optimized Restricted Additive Schwarz Method. *SIAM Journal on Scientific Computing*, 39(4):A1345–A1365.

Hénon, P., Ramet, P., and Roman, J. (2002). PASTIX: A high-performance parallel direct solver for sparse symmetric positive definite systems. *Parallel Computing*, 28(2):301–321.

Hénon, P. and Saad, Y. (2006). A parallel multistage ILU factorization based on a hierarchical graph decomposition. *SIAM J. Sci. Comput.*, 28(6):2266–2293.

Hestenes, M. R. and Stiefel, E. (1952). *Methods of Conjugate Gradients for Solving Linear Systems*, volume 49. NBS Washington, DC.

Higham, N. J. (2002). *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition.

Houzeaux, G., Aubry, R., and Vázquez, M. (2011). Extension of fractional step techniques for incompressible flows: The preconditioned orthomin (1) for the pressure schur complement. *Computers & Fluids*, 44(1):297–313.

Irony, D., Toledo, S., and Tiskin, A. (2004). Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64(9):1017–1026.

Japhet, C. (1998). *Méthode de décomposition de domaine et conditions aux limites artificielles en mécanique des fluides: méthode Optimisée d'Orde 2.* PhD thesis, Université Paris-Nord - Paris XIII.

Jia-Wei, H. and Kung, H.-T. (1981). I/O complexity: The red-blue pebble game. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, pages 326–333. ACM.

Jolivet, P., Hecht, F., Nataf, F., and Prud'homme, C. (2013). Scalable Domain Decomposition Preconditioners for Heterogeneous Elliptic Problems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 80:1–80:11, New York, NY, USA. ACM.

Joncquieres, V., Pechereau, F., Lagunas, A. A., Bourdon, A., Vermorel, O., and Cuenot, B. (2018). A 10-moment fluid numerical solver of plasma with sheaths in a Hall Effect Thruster. *2018 Joint Propulsion Conference, AIAA Propulsion and Energy Forum*, AIAA 2018-4905.

Karypis, G. and Kumar, V. (2009). MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0.

Klawonn, A., Kuhn, M., and Rheinbach, O. (2016a). Adaptive coarse spaces for FETI-DP in three dimensions. *SIAM Journal on Scientific Computing*, 38(5):A2880–A2911.

Klawonn, A., Kühn, M., and Rheinbach, O. (2018). Adaptive FETI-DP and BDDC methods with a generalized transformation of basis for heterogeneous problems. *Electronic Transactions on Numerical Analysis*, 49:1–27.

Klawonn, A., Radtke, P., and Rheinbach, O. (2016b). A comparison of adaptive coarse spaces for iterative substructuring in two dimensions. *Electron. Trans. Numer. Anal*, 45:75–106.

Knuth, D. E. (1984). Literate Programming. *The Computer Journal*, 27(2):97–111.

Lam, S. K., Pitrou, A., and Seibert, S. (2015). Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, page 7. ACM.

Langou, J. (2014). Improving Communication Lower Bounds for Matrix-Matrix Multiplication.

Le Tallec, P. and Vidrascu, M. (1998). Generalized Neumann-Neumann preconditioners for iterative substructuring. In *Domain Decomposition Methods in Sciences and Engineering*.

LeGendre, M. and Frings, W. (2018). Spindle: Scalable dynamic library and python loading in HPC environments. high performance computing.

Lehoucq, R. B., Sorensen, D. C., and Yang, C. (1998). *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*, volume 6. Siam.

Leibniz, G. (1684). Nova Methodus pro Maximis et Minimis. *Acta Eruditorum*.

Li, L., Lanteri, S., and Perrussel, R. (2014). A hybridizable discontinuous Galerkin method combined to a Schwarz algorithm for the solution of 3d time-harmonic Maxwell's equations. *Journal of Computational Physics*, 256:563–581.

Li, X. S. and Demmel, J. W. (2003). SuperLU_DIST: A Scalable Distributed-memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM Trans. Math. Softw.*, 29(2):110–140.

Li, X. S., Shao, M., Yamazaki, I., and Ng, E. G. (2009). Factorization-based sparse solvers and preconditioners. *Journal of Physics: Conference Series*, 180(1):012015.

Lions, P.-L. (1988). On the Schwarz alternating method I. In Glowinski, R., Golub, G. H., Meurant, G. A., and Périaux, J., editors, *First International Symposium on Domain Decomposition Methods for Partial Differential Equations, Held in Paris, France, January 7-9, 1987*, pages 1–42, Philadelphia, PA. SIAM.

Lions, P. L. (1989). On the Schwarz alternating method II: Stochastic interpretation and order properties. In Chan, T., Glowinski, R., Périaux, J., and Widlund, O., editors, *Second International Symposium on Domain Decomposition Methods for Partial Differential Equations , Held in Los Angeles, California, January 14-16, 1988*, pages 47–70, Philadelphia, PA. SIAM.

Lions, P. L. (1990). On the Schwarz alternating method III: A variant for nonoverlapping subdomains. In Chan, T. F., Glowinski, R., Périaux, J., and Widlund, O., editors, *Third International Symposium on Domain Decomposition Methods for Partial Differential Equations , Held in Houston, Texas, March 20-22, 1989*, Philadelphia, PA. SIAM.

Maday, Y. and Magoules, F. (2006). Absorbing interface conditions for domain decomposition methods: A general presentation. *Computer methods in applied mechanics and engineering*, 195(29-32):3880–3900.

Mandel, J. (1993). Balancing domain decomposition. *International Journal for Numerical Methods in Biomedical Engineering*, 9(3):233–241.

Mansfield, L. (1990). On the Conjugate Gradient Solution of the Schur Complement System Obtained from Domain Decomposition. *SIAM Journal on Numerical Analysis*, 27(6):1612–1620.

Mathew, T. P. A. (2008). *Domain Decomposition Methods for the Numerical Solution of Partial Differential Equations*, volume 61. Springer Science & Business Media.

Matsokin, A. M. and Nepomnyaschikh, S. V. (1985). The Schwarz alternation method in a subspace. *Izvestiya Vysshikh Uchebnykh Zavedenii. Matematika*, (10):61–66.

Nakov, S. (2015). *On the design of sparse hybrid linear solvers for modern parallel architectures*. Theses, Université de Bordeaux.

Nataf, F., Rogier, F., and de Sturler, E. (1995). Domain Decomposition Methods for Fluid Dynamics. In *Navier—Stokes Equations and Related Nonlinear Problems*, pages 367–376. Springer, Boston, MA.

Nataf, F., Xiang, H., Dolean, V., and Spillane, N. (2011). A Coarse Space Construction Based on Local Dirichlet-to-Neumann Maps. *SIAM Journal on Scientific Computing*, 33(4):1623–1642.

Newton, I. (1687). *Philosophiae Naturalis Principia Mathematica*. Royal Society.

Nicolaides, R. A. (1987). Deflation of conjugate gradients with applications to boundary value problems. *SIAM Journal on Numerical Analysis*, 24(2):355–365.

Quarteroni, A. and Valli, A. (1999). *Domain Decomposition Methods for Partial Differential Equations*. Oxford University Press.

Rajamanickam, S., Boman, E. G., and Heroux, M. A. (2012a). ShyLU: A hybrid-hybrid solver for multicore platforms. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 631–643. IEEE.

Rajamanickam, S., Boman, E. G., and Heroux, M. A. (2012b). ShyLU: A hybrid-hybrid solver for multicore platforms. *Parallel and Distributed Processing Symposium, International*, 0:631–643.

Ritz, W. (1908). über eine neue Methode zur Lösung gewisser Variationsprobleme der mathematischen Physik. *Journal für die reine und angewandte Mathematik*, 135:1–61.

Saad, Y. (2003). *Iterative Methods for Sparse Linear Systems*, volume 82. siam.

Saad, Y. and Schultz, M. H. (1986). GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing*, 7(3):856–869.

Saad, Y. and Sosonkina, M. (1999). Distributed Schur complement techniques for general sparse linear systems. *SIAM Journal on Scientific Computing*, 21(4):1337–1356.

Saad, Y., Yeung, M., Erhel, J., and Guyomarc'h, F. (2000). A deflated version of the conjugate gradient algorithm. *SIAM Journal on Scientific Computing*, 21(5):1909–1926.

Sarkis, M. (2003). Partition of unity coarse spaces: Enhanced versions, discontinuous coefficients and applications to elasticity. *Domain decomposition methods in science and engineering*, pages 149–158.

Schenk, O., Gärtner, K., Fichtner, W., and Stricker, A. (2000). PARDISO: A High-Performance Serial and Parallel Sparse Linear Solver in Semiconductor Device Simulation.

Schulte, E. and Davison, D. (2011). Active documents with org-mode. *Computing in Science & Engineering*, 13(3):66–73.

Schwarz, H. A. (1870). *Über Einen Grenzübergang Durch Alternirendes Verfahren*. Zürcher u. Furrer.

Sid Lakhdar, M. W. (2014). *Scaling the solution of large sparse linear systems using multifrontal methods on hybrid shared-distributed memory architectures*. Theses, Ecole normale supérieure de lyon - ENS LYON.

Smith, B. F. (1991). A domain decomposition algorithm for elliptic problems in three dimensions. *Numerische Mathematik*, 60(1):219–234.

Spillane, N. (2014). *Méthodes de Décomposition de Domaine Robustes Pour Les Problèmes Symétriques Définis Positifs*. PhD Thesis, Paris 6.

Spillane, N. (2016). An adaptive multipreconditioned conjugate gradient algorithm. *SIAM journal on Scientific Computing*, 38(3):A1896–A1918.

Spillane, N., Dolean, V., Hauret, P., Nataf, F., Pechstein, C., and Scheichl, R. (2014a). Abstract robust coarse spaces for systems of PDEs via generalized eigenproblems in the overlaps. *Numerische Mathematik*, 126(4):741–770.

Spillane, N., Dolean, V., Hauret, P., Nataf, F., Pechstein, C., and Scheichl, R. (2014b). Achieving robustness through coarse space enrichment in the two level Schwarz framework. In *Domain Decomposition Methods in Science and Engineering XXI*, pages 447–455. Springer.

Spillane, N. and Rixen, D. J. (2013). Automatic spectral coarse spaces for robust finite element tearing and interconnecting and balanced domain decomposition algorithms. *International Journal for Numerical Methods in Engineering*, 95(11):953–990.

St-Cyr, A., Gander, M. J., and Thomas, S. J. (2007). Optimized multiplicative, additive, and restricted additive Schwarz preconditioning. *SIAM Journal on Scientific Computing*, 29(6):2402–2425.

Stanisic, L. and Legrand, A. (2014). Effective reproducible research with org-mode and git. In *European Conference on Parallel Processing*, pages 475–486. Springer.

The MPI Forum, C. (1993). MPI: A Message Passing Interface. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, pages 878–883, New York, NY, USA. ACM.

Toselli, A. and Widlund, O. (2006). *Domain Decomposition Methods-Algorithms and Theory*, volume 34. Springer Science & Business Media.

Vázquez, M., Houzeaux, G., Koric, S., Artigues, A., Aguado-Sierra, J., Arís, R., Mira, D., Calmet, H., Cucchietti, F., and Owen, H. (2016). Alya: Multiphysics engineering simulation toward exascale. *Journal of computational science*, 14:15–27.

Wilkinson, J. H. (1963). *Rounding Errors in Algebraic Processes*. Prentice-Hall, Englewood Cliffs, New Jersey.

Yamazaki, I. and Li, X. S. (2010). On techniques to improve robustness and scalability of a parallel hybrid linear solver. In *International Conference on High Performance Computing for Computational Science*, pages 421–434. Springer.