# On the design of sparse hybrid linear solvers for modern parallel architectures

Stojce Nakov


## ▶ To cite this version:

Stojce Nakov. On the design of sparse hybrid linear solvers for modern parallel architectures. Other [cs.OH]. Université de Bordeaux, 2015. English. NNT : 2015BORD0298 . tel-01304315

## HAL Id: tel-01304315
## https://theses.hal.science/tel-01304315v1

Submitted on 19 Apr 2016

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

université de BORDEAUX

THÈSE PRÉSENTÉE

POUR OBTENIR LE GRADE DE

# DOCTEUR D'INFORMATIQUE

# L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE

SPÉCIALITÉ : INFORMATIQUE

Par Stojce NAKOV

# On the design of sparse hybrid linear solvers for modern parallel architectures

Sous la direction de : Roman Jean
co-directeur : Emanuel Agullo

Soutenue le : 14 décembre 2015

Membres du jury :

| | | |
|---|---|---|
| M. Marc Casas | Senior researcher, BSC, Espagne | Président |
| Mme. Sherry Li Xiaoye | Senior Scientist, Lawrence Berkeley National Laboratory, USA | rapporteur |
| M. George Bosilca | Research Director, ICL, Univ. of Tennessee, USA | rapporteur |
| M. Henri Calandra | Docteur, expert HPC TOTAL, USA | Examinateur |
| M. Julien Diaz | Chargé de recherche, Inria | Examinateur |

## Titre : Sur la conception de solveurs linéaires hybrides pour les architectures parallèles modernes

## Résumé :

Dans le contexte de cette thèse, nous nous focalisons sur des algorithmes pour l’algèbre linéaire numérique, plus précisément sur la résolution de grands systèmes linéaires creux. Nous mettons au point des méthodes de parallélisation pour le solveur linéaire hybride MaPHyS. Premièrement nous considerons l'aproche MPI+threads. Dans MaPHyS, le premier niveau de parallélisme consiste au traitement indépendant des sous-domaines. Le second niveau est exploité grâce à l’utilisation de noyaux multithreadés denses et creux au sein des sous-domaines. Une telle implémentation correspond bien à la structure hiérarchique des supercalculateurs modernes et permet un compromis entre les performances numériques et parallèles du solveur. Nous démontrons la flexibilité de notre implémentation parallèle sur un ensemble de cas tests. Deuxièmement nous considérons un approche plus innovante, où les algorithmes sont décrits comme des ensembles de tâches avec des inter-dépendances, i.e., un graphe de tâches orienté sans cycle (DAG). Nous illustrons d’abord comment une première parallélisation à base de tâches peut être obtenue en composant des librairies à base de tâches au sein des processus MPI illustrer par un prototype d’implémentation préliminaire de notre solveur hybride. Nous montrons ensuite comment une approche à base de tâches abstrayant entièrement le matériel peut exploiter avec succès une large gamme d’architectures matérielles. À cet effet, nous avons implanté une

version à base de tâches de l’algorithme du Gradient Conjugué et nous montrons que l’approche proposée permet d’atteindre une très haute performance sur des architectures multi-GPU, multicoeur ainsi qu’hétérogène.

**Mots clés :** Calcul haute performance' ;multi-cœur ; solveur linéaires creux ; méthodes hybride ; programmation en tâche ; architecture hétérogène.

## Title :

**On the design of sparse hybrid linear solvers for modern parallel architectures**


## Abstract :

In the context of this thesis, our focus is on numerical linear algebra, more precisely on solution of large sparse systems of linear equations. We focus on designing efficient parallel implementations of MaPHyS, an hybrid linear solver based on domain decomposition techniques. First we investigate the MPI+threads approach. In MaPHyS, the first level of parallelism arises from the independent treatment of the various subdomains. The second level is exploited thanks to the use of multi-threaded dense and sparse linear algebra kernels involved at the subdomain level. Such an hybrid implementation of an hybrid linear solver suitably matches the hierarchical structure of modern supercomputers and enables a trade-off between the numerical and parallel performances of the solver. We demonstrate the flexibility of our parallel implementation on a set of test examples. Secondly, we follow a more disruptive approach where the algorithms are described as sets of tasks with data inter-dependencies that leads to a directed acyclic graph (DAG) representation. The tasks are handled by a runtime system. We illustrate how a first task-based parallel implementation can be obtained by composing task-based parallel libraries within MPI

processes throught a preliminary prototype implementation of our hybrid solver. We then show how a task-based approach fully abstracting the hardware architecture can successfully exploit a wide range of modern hardware architectures. We implemented a full task-based Conjugate Gradient algorithm and showed that the proposed approach leads to very high performance on multi-GPU, multicore and heterogeneous architectures.

**Keywords :** High Performance Computing (HPC); multicore; sparse linear solver; hybrid method; task ; moteur d'exécution ; heterogeneous architectures.

---

**Inria Bordeaux - Sud-Ouest**

200 Avenue de la Vieille Tour, 33405 Talence

# Contents

**Résumé en Français** 1

**Introduction** 3

**1 Introduction to the field** 5

1.1 Some insights on numerical linear algebra in seismic imaging . . . . . . . . . 5

1.2 Brief introduction to sparse linear algebra . . . . . . . . . . . . . . . . . . . . 7

1.2.1 Sparse matrices . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

1.2.2 Solutions for large sparse systems of linear equations . . . . . . . . . 9

1.3 Some basics on hybrid linear solvers . . . . . . . . . . . . . . . . . . . . . . . 12

1.3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

1.3.2 A brief overview of overlapping domain decomposition . . . . . . . . 14

1.3.3 A brief overview of non-overlapping domain decomposition . . . . . . 15

1.4 Some background on Krylov subspace methods . . . . . . . . . . . . . . . . . 18

1.4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

1.4.2 The unsymmetric problems . . . . . . . . . . . . . . . . . . . . . . . 19

1.4.3 The symmetric positive definite problems . . . . . . . . . . . . . . . . 21

1.4.4 Stopping criterion for convergence detection . . . . . . . . . . . . . . 23

1.5 Sparse linear solvers on modern supercomputers . . . . . . . . . . . . . . . . 24

1.5.1 Manycore architectures . . . . . . . . . . . . . . . . . . . . . . . . . . 24

1.5.2 Evolutionary parallel programming paradigms . . . . . . . . . . . . . 26

1.5.3 Revolutionary programming paradigms . . . . . . . . . . . . . . . . . 27

1.6 Positioning of the thesis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

**2 A hierarchical hybrid sparse linear solver for multicore platforms 31**

2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
2.2 Parallel algebraic non-overlapping domain decomposition methods . . . . . . 33
2.2.1 Governing ideas . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
2.2.2 Related work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
2.3 Design of parallel implementations of MaPHyS . . . . . . . . . . . . . . . . 36
2.3.1 Baseline MPI implementation of the MaPHyS solver . . . . . . . . . 36
2.3.2 Design of a 2-level MPI+thread extension of MaPHyS . . . . . . . . 43
2.4 Performance analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
2.4.1 Experimental setup . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
2.4.2 Impact of the thread binding strategies . . . . . . . . . . . . . . . . 49
2.4.3 Multithreading performance . . . . . . . . . . . . . . . . . . . . . . 52
2.4.4 Numerical and parallel flexibility of the 2-level implementation . . . . 56
2.5 Case study from geoscience applications . . . . . . . . . . . . . . . . . . . . 62
2.6 Performance sensitivity with respect to the partitioning quality . . . . . . . . 71
2.7 Comparison of MaPHyS with the PDSLin hybrid-solver . . . . . . . . . . . 78
2.8 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83

**3 Towards task-based hybrid sparse linear solvers 85**

3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
3.2 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89
3.2.1 Task-based runtime systems and related programming models . . . . 89
3.2.2 Sparse linear algebra on modern architectures . . . . . . . . . . . . . 94
3.2.3 The StarPU task-based runtime system . . . . . . . . . . . . . . . . . 96
3.3 Prototype design of an MPI+task extension of MaPHyS . . . . . . . . . . . 98
3.4 Towards a full task-based version of MaPHyS: case study with the CG algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101
3.4.1 Baseline STF conjugate gradient algorithm . . . . . . . . . . . . . . . 102
3.4.2 Experimental setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105
3.4.3 Scheduling and mapping strategy . . . . . . . . . . . . . . . . . . . . 106
3.4.4 Building block operations . . . . . . . . . . . . . . . . . . . . . . . . 106
3.4.5 Achieving efficient software pipelining . . . . . . . . . . . . . . . . . . 108
3.4.6 Performance analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . 114
3.4.7 Combining software pipelining with numerical pipelining . . . . . . . 120

3.5 Conclusion . . . . . . . . . . 124

**Perspectives and concluding remarks** **127**

**Bibliography** **129**

# Résumé en Français

Dans les quelques décennies passées, il y a eu d'innombrables avancées scientifiques, techniques et sociétales permises par la simulation numérique grâce au développement d'applications, d'algorithmes et d'architectures de calcul haute performance (HPC). Ces outils de simulation numérique puissants ont fourni aux chercheurs la possibilité de trouver des solutions calculatoires pour de nombreux questions et problèmes scientifiques conséquents en médecine, biologie, climatologie, nanotechnologie, énergie et environnement. Dans le contexte de cette thèse, nous nous focalisons sur des algorithmes pour l'algèbre linéaire numérique, plus précisément sur la résolution de grands systèmes linéaires creux. Nous mettons au point des méthodes de parallélisation efficaces pour l'outil MaPHyS, un solveur linéaire hybride basé sur des techniques de décomposition de domaines algébrique. Deux approches sont considérées.

La première approche consiste à proposer une implantation combinant MPI+threads. Dans MaPHyS, le premier niveau de parallélisme consiste au traitement indépendant des sous-domaines et est implémenté avec un paradigme à base d'échanges de passage de messages. Le second niveau est exploité grâce à l'utilisation de noyaux d'algèbre linéaire multithreadés denses et creux au sein des sous-domaines. Une telle implémentation hybride hiérarchique correspond bien à la structure hiérarchique des supercalculateurs modernes et permet un compromis entre les performances numériques et parallèles du solveur. Nous décrivons comment l'interopérabilité entre les différents noyaux doit être appliquée afin d'assurer le passage à l'échelle du solveur parallèle. Nous démontrons la flexibilité de notre implémentation parallèle sur un ensemble de cas tests provenant de collections classiques de matrices creuses issues de problèmes 3D ainsi que de configurations géophysiques délicates fournies par notre partenaire industriel, Total.

Dans une seconde partie, nous considérons un approche plus innovante, où les algorithmes sont décrits comme des ensembles de tâches avec des inter-dépendances, i.e., un graphe de tâches orienté sans cycle (DAG). L'ordonnancement et l'affectation de ces tâches sont pris en charge par un moteur d'exécution. Une telle approche permet de maintenir un description avec haut-niveau d'abstraction des algorithmes et ne nécessite pas d'enchevêtrer les complexités numérique et de mise en œuvre parallèle. Plutôt que de ré-écrire entièrement le code d'un solveur hybride suivant un tel paradigme, ce qui représenterait une tâche considérable, nous procédons à une étude incrémentale de faisabilité. Nous illustrons d'abord comment une première parallélisation à base de tâches peut être obtenue en composant

des librairies à base de tâches au sein des processus MPI. Nous illustrons notre discussion avec un prototype d'implémentation préliminaire d'un tel solveur hybride. Nous montrons ensuite comment une approche à base de tâches abstrayant entièrement le matériel peut exploiter avec succès une large gamme d'architectures matérielles dans le cas d'un composant clé d'un solveur hybride qui la méthode itérative de Krylov. À cet effet, nous avons implanté une version entièrement à base de tâches de l'algorithme du gradient conjugué et nous montrons que l'approche proposée permet d'atteindre une très haute performance sur des architectures multi-GPU, multicoeur ainsi qu'hétérogène. Cette étude préliminaire motive la mise au point à base de tâches d'un solveur hybride dans son entier, ce qui sera l'objet d'un travail futur.

# Introduction

Over the last few decades, there have been innumerable science, engineering and societal breakthroughs enabled by the development of High Performance Computing (HPC) applications, algorithms and architectures. These powerful tools have provided researchers with the ability to computationally find efficient solutions for some of the most challenging scientific questions and problems in medicine and biology, climatology, nanotechnology, energy and environment. It is admitted today that *numerical simulation is the third pillar for the development of scientific discovery at the same level as theory and experimentation.* While the hardware becomes more and more complex, a significant effort must be devoted to the design and the implementation of novel numerical schemes. In the context of this thesis, our focus is on numerical linear algebra algorithms that appear in many large scale simulations and are often the most time consuming numerical kernel; more precisely we consider numerical schemes for the solution of large sparse systems of linear equations.

One route to the parallel scalable solution of large sparse linear systems in parallel scientific computing is the use of hybrid methods that hierarchically combine direct and iterative methods. These techniques inherit the advantages of each approach, namely the limited amount of memory and natural parallelization for the iterative component and the numerical robustness of the direct part. The general underlying ideas are not new since they have been intensively used to design domain decomposition techniques; those approaches cover a fairly large range of computing techniques for the numerical solution of partial differential equations (PDEs) in time and space. Generally speaking, it refers to the splitting of the computational domain into subdomains with or without overlap. The splitting strategy is generally governed by various constraints/objectives but the main is to express parallelism. In this thesis, we focus on designing efficient parallel implementations of a hybrid solver, namely MaPHyS. Different approaches are considered in that perspective. In Chapter 1, we describe the general scientific computational framework and introduce the main numerical ingredients as well as the key computing components and programming paradigms. The main contributions of this work are detailed in the next two chapters.

Chapter 2 is devoted to the design of a 2-level parallel algorithm. The first level of parallelism arises from the independent treatment of the various subdomains and is managed using message passing. The second level is exploited thanks to the use of multi-threaded dense and sparse linear algebra kernels involved at the subdomain level. Such an hybrid implementation of an hybrid linear solver suitably matches the hierarchical structure of

modern supercomputers and enables a trade-off between the numerical and parallel performances of the solver. We describe how the interoperability between the various kernels has to be mastered to ensure the scalability of the parallel solver. We demonstrate the flexibility of our parallel implementation on a set of test examples coming from classical test matrices as well as from geoscience challenging test cases provided by our industrial partner Total. We furthermore perform a preliminary comparison with another state of the art hybrid solver PDSLin that also implements a 2-level parallelism scheme. This latter activity was developed in the framework of the FAST-LA associate team in collaboration with S. Li's group in LBNL.

In Chapter 3, we follow a more disruptive approach where the algorithms are described as sets of tasks with data inter-dependencies that leads to a directed acyclic graph (DAG) representation. The scheduling and mapping of these tasks are handled by a runtime system. Such an approach permits to keep a high level description of the algorithms and does not require to interleave their numerical and parallel complexities. While designing from scratch an hybrid solver based on this paradigm would be a huge development effort, we perform an incremental feasibility study. We first illustrate how a first task-based parallel implementation can be obtained by composing task-based parallel libraries within MPI processes. We illustrate our discussion with a preliminary prototype implementation of such an hybrid solver. We then show how a task-based approach fully abstracting the hardware architecture can successfully exploit a wide rande of modern hardware architectures in the case of a key component of the hybrid solver that is a Krylov method. We implemented a full task-based conjugate gradient algorithm and showed that the proposed approach leads to very high performance on multi-GPU, multicore and heterogeneous architectures. This preliminary study motivates the design of the whole hybrid solver as a full task-based algorithm, which will be the focus of a future work.

Finally, we should mention that the work presented within this manuscript was conducted within the HiePACS[1] project-team at Inria Bordeaux Sud-Ouest with the financial support of TOTAL.

[1]https://team.inria.fr/hiepacs/

Chapter 1

# Introduction to the field

## 1.1 Some insights on numerical linear algebra in seismic imaging

While the work described in this manuscript can be applied (and has been assessed, see Section 2.4) to any sparse linear system, its main motivation came from the geophysic's context that we briefly describe in this section. In particular, we give brief insights on the underlying partial differential equation (PDE) and approximation schemes that give rise to the large sparse linear systems that are the main challenges (see Section 2.5) considered to assess the parallel performance and numerical robustness of the hybrid solver that is the core of this work.

The development of robust, accurate and efficient solution methodologies for 3D wave propagation problems appears in many applications and is critical for oil companies. In particular, the solution of the wave propagation problems is a key component for seismic imaging that solves an inverse problem to find the best representation of the subsurface that best matches to the data recorded during acquisition campaigns. In such an inverse calculation, the forward problem (i.e., the wave propagation problem) has to be solved many times. Therefor it is of paramount importance to design effective parallel techniques to address its solution.

The simplest wave propagation model describes an acoustic wave in a fluid. The equation resulting from this model writes

$$\frac{1}{c^2}\frac{\partial^2 p}{\partial t^2} - \Delta p = f \tag{1.1}$$

where $c$ is the wave velocity in the fluid, $p$ the unknown pressure field that varies in time and location and depends on the source term $f$. The equation is closed using boundary conditions; for geophysics applications they often are absorbing conditions enabling to restrict the calculation to a bounded domain. This hyperbolic PDE can be solved either in a time domain or in a frequency domain. The two approaches require to approximate some space derivatives whose discretizations lead to sparse matrices.

The time domain approach directly solves (1.1). Due to the unsteady nature of the

solution, explicit schemes are often considered. From a linear algebra view point, once the space and time discretization have been performed, the selected explicit scheme mainly reduces to sparse matrix-vector product calculations plus the solution of an essentially diagonal linear system (i.e., the mass matrix). With such an approach, all the frequencies that compose the solution are computed as a function of time.

For the frequency domain approach, a Fourier transform is applied to Equation (1.1) that translates into the Helmholtz equation associated with the Fourier mode $\omega$:

$$-\Delta u - \frac{\omega^2}{c^2} u = g(\omega). \tag{1.2}$$

The space discretization of (1.2) using finite differences, finite elements or finite volumes leads to the solution of a large sparse linear system.

With the increasing scarcity of oil, the oil companies explore ever more challenging geological features. To achieve high fidelity computation, more detailed models than the pure acoustic model must be considered. Furthermore, advanced discretization techniques based on fully unstructured meshes must be selected. With these constraints, the elastodynamic equations are chosen to model seismic waves. In the frequency domain, these equations are

$$\begin{cases} \nabla \cdot \sigma(u) + \omega^2 \rho u = f & \text{in } \Omega, \\ \sigma(u) n = c_p u \cdot n \, n + c_s u \wedge n & \text{on } \partial\Omega \end{cases} \tag{1.3}$$

where $u$ is the unknown displacement field in the solid domain $\Omega$, $\omega$ is the circular frequency, $\rho$ is a positive real number denoting the density of $\Omega$, $\sigma$ is the stress tensor. Equation (1.3) is then discretized using a discontinuous Galerkin finite element method that exhibits attractive features including adaptivity and flexibility. We give below a brief sketch of the approach; we refer to [26] and the references therein for a complete and detailed description of the approximation procedure.

First, Equation (1.3) has to be written in a weak form and an approximation space $V_h$ defined by a set of linearly independent test functions. The variational form associated with (1.3) can be written as follows

$$\begin{cases} \text{find } u_h \in V_h \text{ such that} \\ a(u_h, v_h) = f_1(v_h), & \forall\, v_h \in V_h \end{cases} \tag{1.4}$$

where $a$ is a bilinear and Hermitian form and $f_1$ is a complex-valued linear form.

The test functions are piece-wise polynomials of degree $p$ in each element. However, unlike standard finite elements, such functions are not continuous over the computational domain $\Omega$ and might jump across the faces between the elements. Given a mesh on $\Omega$ and associated finite element space, the variational problem can be recast at the algebraic level as follows:

$$\mathcal{A} x = b \tag{1.5}$$

where $\mathcal{A}$ is an Hermitian matrix, $b$ the discretization of the source term and $x$ the vectors whose entries are the coordinates of the solution in the approximation space $V_h$.

Matrices arising from the discretization of the elastodynamic system on 3D meshes will be considered in Section 2.5 to benchmark the parallel performance and robustness of our implementation of the hybrid solver considered here.

## 1.2 Brief introduction to sparse linear algebra

Numerical linear algebra plays a central role in solving many real-world problems. To understand phenomena or to solve problems, scientists use mathematical models. Solutions obtained from mathematical models are often satisfactory solutions to complex problems in fields such as seismic imaging (see above), weather prediction, trajectory of a spacecraft, car crashes simulation, etc. In many scientific fields such as electrostatics, electrodynamics, fluid flow, elasticity, or quantum mechanics, problems are broadly modeled by partial differential equations (PDEs). The common way to solve PDEs is to approximate the solution, which is continuous, by discrete equations that involve a finite, but often large, number of unknowns [128, Chapter 2]. This strategy is called discretization. There are many ways to discretize a PDE, the three most widely used being the finite element method (FEM), finite volume methods (FVM) and finite difference methods (FDM). These discretization strategies lead to large and sparse matrices. Thus real-word linear applications translate into numerical linear algebra problems. There are many linear algebra problems, but this work focuses on the resolution of linear systems of equations $\mathcal{A}x = b$.

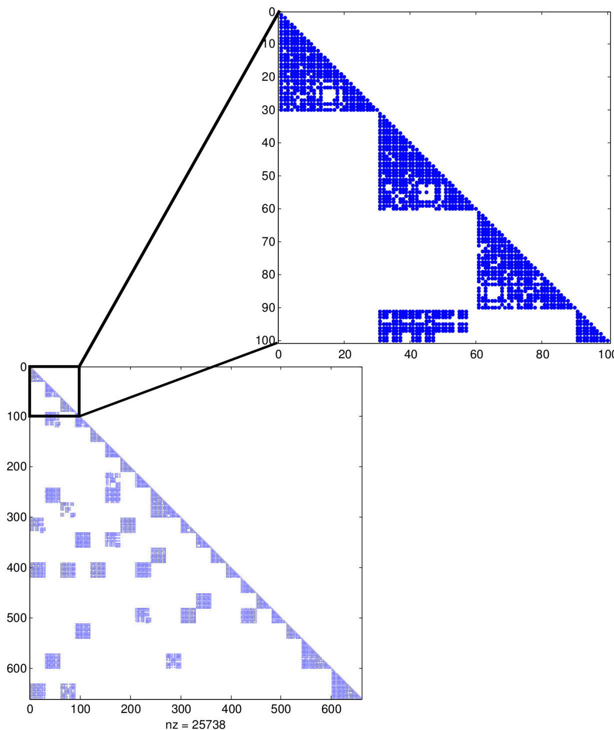
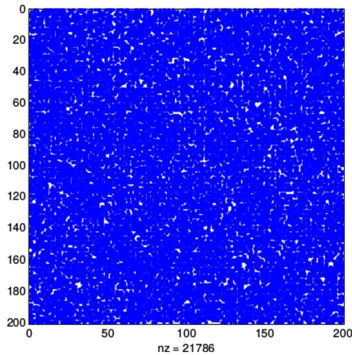



Figure 1.1: Pattern of a sparse matrix involved in seismic imaging.

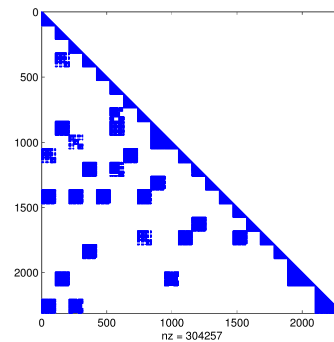
For example, we display in Figure 1.1 the pattern of a sparse matrix which is obtained by discretization of an elastodynamic system on a 3D mesh.

### 1.2.1 Sparse matrices

A matrix is said to be sparse if it contains only very few nonzero elements, as depicted in Figure 1.2(b), 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 1.2(a) contains 54% of nonzero elements, it cannot be termed sparse, althought the one presented in Figure 1.2(b) can be clearly considered as sparse. As reported for example by Yousef Saad [128, Chapter 2], 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 the basic idea is to store only nonzero elements.




(a) This matrix contains 54% of nonzero elements.




(b) This matrix contains 3% of nonzero elements.

Figure 1.2: 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.

The main goal of these data structures is to store only non-zero elements while at the same time facilitate sparse matrix operations. The most general sparse matrix storage is called coordinate (COO) format and consists of three arrays of size $nnz$, where $nnz$ is the number of nonzero elements. As illustrated in Figure 1.3, the first array (`AA`) contains the nonzero elements of the sparse matrix, the second array (`JR`) contains the corresponding row indices and the third array (`JC`) contains the corresponding column indices.

The COO format is the most flexible but possibly not optimized since row indices and column indices may be stored redundantly. In the example depicted in Figure 1.3, the row index “3”, is stored 4 times in `JR`, and the column index “4” is also stored 4 times. It is possible to compress row indices, which leads to compressed sparse row (CSR) format. Alternatively the column indices can also be compressed, this format is called compressed

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

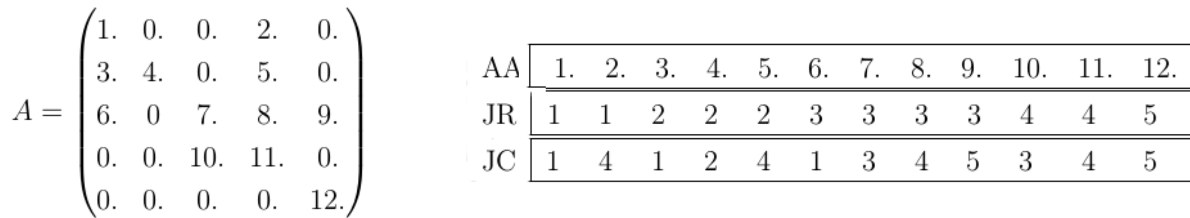



Figure 1.3: Coordinate (COO) format for sparse matrix representation.

sparse column (CSC). Other sparse data structures exist to further exploit particular situations. We refer to [128, Chapter 2] for a detailed description of possible data structures for sparse matrices.

## 1.2.2 Solutions for large sparse systems of linear equations

### 1.2.2.1 Direct methods for linear systems of equations

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 (1.6),

$$\begin{pmatrix} 1. & 0. & 0. & 2. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 6. & 0 & 7. & 8. & 9. \\ 0. & 0. & 10. & 11. & 0. \\ 0. & 0. & 0. & 0. & 12. \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 5 \\ 4 \\ 3 \\ 2 \\ 1 \end{pmatrix} \tag{1.6}$$

a broadly class of methods is based on Gaussian elimination. 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 (1.6) those factors are:

$$L = \begin{pmatrix} 1.00 & 0 & 0 & 0 & 0 \\ 3.00 & 1.00 & 0 & 0 & 0 \\ 0 & 1.50 & 1.00 & 0 & \\ 0 & 0 & 0.56 & 1.00 & 0 \\ 0 & 0 & 0 & 0.77 & 1.00 \end{pmatrix} \quad U = \begin{pmatrix} 1.00 & 0 & 2.00 & 0 & 0 \\ 0 & 4.00 & -6.00 & 5.00 & 0 \\ 0 & 0 & 16.00 & 14.21 & -4.50 \\ 0 & 0 & 0 & -7.50 & 8.00 \\ 0 & 0 & 0 & 0 & 15.48 \end{pmatrix}.$$

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 $y = (5.00, -11.00, 19.50, -8.96, 7.93)^T$, which leads to the solution $x = (3.51, -1.05, .074, -0.46, 0.51)^T$. The advantage of this approach is that most of the work is performed in the decomposition step ($\mathcal{O}(n^3)$ for dense matrices) and very little in the forward and backward substitutions ($\mathcal{O}(n^2)$). 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 SPD), 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 [86]. 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. Because of the fill-in, solving large sparse linear algebra problems using direct methods is very challenging in terms of memory usage.

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` [50], MUMPS [13, 14], PARDISO [133], PASTIX [81], `SuperLU` [100], to name a few. Sparse methods work well for 2D PDE discretizations, but they may be very penalizing in terms of memory usage and computational time for large 3D test cases.

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. Successive iterations implemented by iterative methods require a small amount of storage and floating point operations, but might converge slowly or not converge at all. On the other hand iterative methods are generally less robust than direct solvers for general sparse linear systems.

#### 1.2.2.2 Iterative methods for linear systems of equations

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

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

$$Mx^{(k+1)} = Nx^{(k)} + b \tag{1.7}$$

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 (1.7) when $k$ will go to infinity. Given any initial guess $x^{(0)}$, the stationary method described in Equation (1.7) converges if and only if $\rho(M^{-1}N) < 1$, where the spectral radius $\rho(\mathcal{A})$ of a given matrix $\mathcal{A}$ with eigenvalues $\lambda_i$ is defined by $\rho(\mathcal{A}) = max(|\lambda_i|)$ [128, Chapter 4].

Typical iterative methods for linear systems are Gauss-Seidel, Jacobi, successive over relaxation etc., as described in Table 1.1 according to the choice of $M$ and $N$.

| $M$ | $N$ | 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.

**Krylov subspaces.** 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 [128]. Let $\mathcal{A} \in \mathbb{C}^{n\times n}$ and $v \in \mathbb{C}^n$, let $m \leq n$, the space denoted by $\mathcal{K}_m(\mathcal{A}, v) = Span\{v, \mathcal{A}v, ..., \mathcal{A}^{m-1}v\}$ is referred to as the Krylov space of dimension $m$ associated with $\mathcal{A}$ and $v$. In order words, 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 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 $\mathcal{M}^{-1}\mathcal{A}$ has *better* convergence properties for the selected solver. The linear systems $\mathcal{M}^{-1}\mathcal{A}x = \mathcal{M}^{-1}b$ has the same solution as the original linear system. This method is called preconditioning and the matrix $\mathcal{M}$ is called an implicit (i.e., $\mathcal{M}$ attempts to somehow approximate $\mathcal{A}$) left preconditioner. On the other hand, linear systems of equations can also be preconditioned from the right: $\mathcal{A}\mathcal{M}^{-1}y = b$, and $x = \mathcal{M}^{-1}y$. One can also consider split preconditioning that is expressed as follows: $\mathcal{M}_1^{-1}\mathcal{A}\mathcal{M}_2^{-1}y = \mathcal{M}_1^{-1}b$, and $x = \mathcal{M}_2^{-1}y$, where the preconditioner is $\mathcal{M} = \mathcal{M}_1\mathcal{M}_2$. It is important to notice that Krylov subspace methods do not compute explicitly the matrices $\mathcal{M}^{-1}\mathcal{A}$ and $\mathcal{A}\mathcal{M}^{-1}$, in order to avoid the associated extra cost and preserve sparsity.

Krylov methods do not require the matrices $\mathcal{A}$ or $\mathcal{M}$ to be explicitly formed. Instead, procedures for applying $\mathcal{A}$ and $\mathcal{M}^{-1}$ 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 [74]:

$$||e^{(k)}||_{\mathcal{A}} \leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k ||e^{(0)}||_{\mathcal{A}},$$

where $e^{(k)} = x^* - x^{(k)}$ denotes the error associated with the iterate at step $k$ and $\kappa$ 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. Similar arguments exist for applying Krylov solvers to unsymmetric systems (e.g. GMRES) [128]. 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.

## 1.3 Some basics on hybrid linear solvers

In this section we briefly describe the basic ingredients that are involved in the hybrid linear solvers considered in this manuscript. The hybrid solver considered in this work borrows ideas to some classical domain decomposition techniques. In this section some popular and well-known domain decomposition preconditioners are described from an algebraic perspective. Numerical techniques that rely on decompositions with overlap are described in Section 1.3.2 and some approaches with non-overlapping domains are discussed in Section 1.3.3. Furthermore, these methods are most often used to accelerate Krylov subspace methods. In that respect, we briefly present the Krylov subspace solvers we have considered for our numerical experiments. Both symmetric positive definite (SPD) problems and unsymmetric problems are encountered that are solved using the conjugate gradient method [85], described in Section 1.4.3, or variants of the GMRES technique [126, 129], described in Section 1.4.2. Because we investigate multiple variants of the preconditioners and intend to compare their numerical behaviors, a particular attention should be paid to the stopping criterion, which should be independent from the preconditioner while ensuring that the computed solutions have similar quality with respect to a certain metric. Consequently, in Section 1.4.4 we introduce some basic concepts of the backward error analysis that enables us to ensure fairness of the comparaison.

### 1.3.1 Introduction

As pointed in [71], the term *domain decomposition* covers a fairly large range of computing techniques for the numerical solution of partial differential equations (PDE's) in time and space. Generally speaking, it refers to the splitting of the computational domain into subdomains with or without overlap. The splitting strategy is generally governed by various constraints/objectives. It might be related to

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

In this work we consider the last situtation, where the overall mesh is split into several sub-meshes. We focus specifically on the associated domain decomposition techniques for

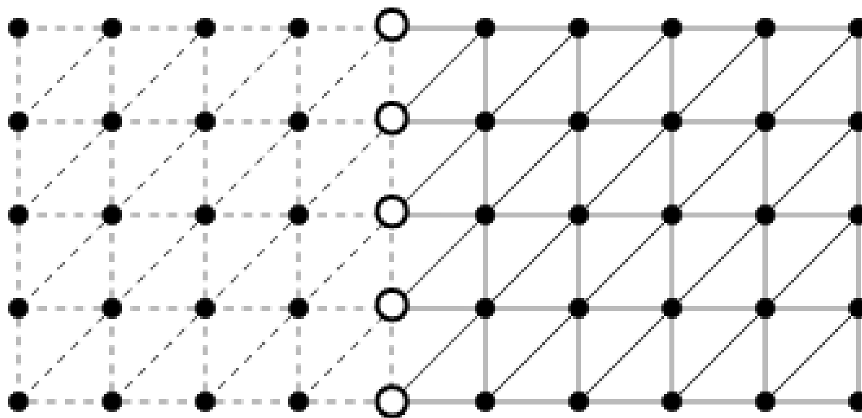


Figure 1.4: Partition of the domain based on a element splitting. Shared vertices are indicated by a circle.

the parallel solution of large linear systems, $\mathcal{A}x = b$, arising from PDE discretizations. Some of the presented techniques can be used as stationary iterative schemes that converge to the linear system solution by properly tuning their governing parameters to ensure that the spectral radius of the iteration matrix is less than one. However, domain decomposition schemes are most effective and require less tuning when they are employed as a preconditioner to accelerate the convergence of a Krylov subspace method [75, 128].

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

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

Corresponding to each subdomain $\Omega_i^0$, we define a rectangular extension matrix ${\mathcal{R}_i^0}^T$ whose action extends by zero a vector of values defined at *mesh* vertices associated with the finite elements contained in $\Omega_i^0$. The entries of ${\mathcal{R}_i^0}^T$ are zeros and ones. For simplicity, we omit in all the following the *0* superscripts and define $\mathcal{R}_i = \mathcal{R}_i^0$ and $\Omega_i = \Omega_i^0$. Notice that the

columns of a given $\mathcal{R}_k$ are orthogonal, but that between the different $\mathcal{R}_i$'s some columns are no longer orthogonal. This is due to the fact that some mesh vertices overlap even though the graph vertices defined by $\Omega_i$ are non-overlapping (shared mesh vertices see Figure 1.4). Let $\Gamma_i$ be the set of all mesh vertices belonging to the interface of $\Omega_i$ (mesh vertices lying on $\partial\Omega_i \backslash \partial\Omega$). Similarly, let $\mathcal{I}_i$ be the set of all remaining mesh vertices within the subdomain $\Omega_i$ (i.e., interior vertices). Considering only the discrete matrix contributions arising from finite elements in $\Omega_i$ gives rise to the following local discretization matrix

$$\mathcal{A}_i = \begin{pmatrix} \mathcal{A}_{\mathcal{I}_i\mathcal{I}_i} & \mathcal{A}_{\mathcal{I}_i\Gamma_i} \\ \mathcal{A}_{\Gamma_i\mathcal{I}_i} & \mathcal{A}_{\Gamma_i\Gamma_i} \end{pmatrix} \tag{1.8}$$

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

$$\mathcal{A} = \sum_{i=1}^{\mathcal{N}} \mathcal{R}_i^T \mathcal{A}_i \mathcal{R}_i. \tag{1.9}$$

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

For the $\delta$-overlap partition, we can define a corresponding restriction operator $\mathcal{R}_i^\delta$ which maps mesh vertices in $\Omega$ to the subset of mesh vertices associated with finite elements contained in $\Omega_i^\delta$. Corresponding definitions of $\Omega_i^\delta$ and $\mathcal{I}_i^\delta$ follow naturally as the boundary and interior mesh vertices associated with finite elements in $\Omega_i^\delta$. The discretization matrix on $\Omega_i^\delta$ has a similar structure to the one given by (1.8) and is written as

$$\mathcal{A}_i^\delta = \begin{pmatrix} \mathcal{A}_{\mathcal{I}_i^\delta\mathcal{I}_i^\delta} & \mathcal{A}_{\mathcal{I}_i^\delta\Gamma_i^\delta} \\ \mathcal{A}_{\Gamma_i^\delta\mathcal{I}_i^\delta} & \mathcal{A}_{\Gamma_i^\delta\Gamma_i^\delta} \end{pmatrix}. \tag{1.10}$$

### 1.3.2 A brief overview on domain decomposition techniques with overlapping domains

The domain decomposition methods based on overlapping subdomains are most often referred to as Schwarz methods due to the pioneering work of Schwarz in 1870 [134]. This work was not intended as a numerical algorithm, but was instead developed to show the existence of the elliptic problem solution on a complex geometry formed by overlapping two simple geometries where solutions are known. With the advent of parallel computing, this basic technique known as the alternating Schwarz method, has motivated considerable research activity. In this section, we do not intend to give an exhaustive presentation of all work devoted to Schwarz methods. Only additive variants that are well-suited for parallel implementations are considered. Within additive variants, computation on all subdomains

are performed simultaneously while multiplicative variants require some subdomain calculations to wait for results from other subdomains. The multiplicative versions often have connections to block Gauss-Seidel methods while the additive variants correspond more closely to block Jacobi methods. We do not further pursue this description but refer the interested reader to [135].

With these notations the additive Schwarz preconditioner is given by

$$\mathcal{M}_{AS}^{\delta} = \sum_{i=1}^{\mathcal{N}} \left(\mathcal{R}_i^{\delta-1}\right)^T \left(\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i^\delta}\right)^{-1} \mathcal{R}_i^{\delta-1}. \tag{1.11}$$

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

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

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

We notice that this technique makes use of a matrix inverse (i.e., a direct solver or an exact factorization) of a local Dirichlet matrix. In practice, it is common to replace this with an incomplete factorization [127, 128] or an approximate inverse [28, 29, 54, 76, 91]. While this usually slightly deteriorates the convergence rate, it can lead to a faster method due to the fact that each iteration is less computationally expensive. Finally, we mention that these techniques based on Schwarz variants are available in several large parallel software libraries; see for instance [24, 83, 84, 101, 140].

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

In this section, methods based on non-overlapping regions are described. Such domain decomposition algorithms are often referred to as sub-structuring schemes. This terminology comes from the structural mechanics discipline where non-overlapping ideas were first developed. In this early work the primary focus was on direct solvers. Associating one frontal matrix with each subdomain allows for coarse grain multiple front direct solvers [60]. Motivated by parallel distributed computing and the potential for coarse grain parallelism, a considerable research activity has been developed to iterative domain decomposition schemes. A very large number of methods have been proposed and we cannot cover all of them. Therefore, the main highlights are surveyed.

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

$$\begin{pmatrix} \mathcal{A}_{\mathcal{I}\mathcal{I}} & \mathcal{A}_{\mathcal{I}\Gamma} \\ \mathcal{A}_{\Gamma\mathcal{I}} & \mathcal{A}_{\Gamma\Gamma} \end{pmatrix} \begin{pmatrix} x_{\mathcal{I}} \\ x_{\Gamma} \end{pmatrix} = \begin{pmatrix} b_{\mathcal{I}} \\ b_{\Gamma} \end{pmatrix}, \tag{1.12}$$

where $x_\Gamma$ contains all unknowns associated with subdomain interfaces and $x_{\mathcal{I}}$ contains the remaining unknowns associated with subdomain interiors. The matrix $\mathcal{A}_{\mathcal{I}\mathcal{I}}$ is block diagonal where each block corresponds to a subdomain interior. Eliminating $x_{\mathcal{I}}$ from the second block row of Equation (1.12) leads to the reduced system

$$\mathcal{S}x_\Gamma = b_\Gamma - \mathcal{A}_{\Gamma\mathcal{I}}\mathcal{A}_{\mathcal{I}\mathcal{I}}^{-1}b_{\mathcal{I}} \text{ where } \mathcal{S} = \mathcal{A}_{\Gamma\Gamma} - \mathcal{A}_{\Gamma\mathcal{I}}\mathcal{A}_{\mathcal{I}\mathcal{I}}^{-1}\mathcal{A}_{\mathcal{I}\Gamma} \tag{1.13}$$

and $\mathcal{S}$ is referred to as the *Schur complement matrix*. This reformulation leads to a general strategy for solving (1.12). Specifically, an iterative method can be applied to (1.13). Once $x_\Gamma$ is determined, $x_{\mathcal{I}}$ can be computed with one additional solve on the subdomain interiors. Further, when $\mathcal{A}$ is symmetric positive definite (SPD), the matrix $\mathcal{S}$ inherits this property and so a conjugate gradient method can be employed.

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

Let $\Gamma$ denote the entire interface defined by $\Gamma = \cup\,\Gamma_i$ where $\Gamma_i = \partial\Omega_i\backslash\partial\Omega$. As interior unknowns are no longer considered, new restriction operators must be defined as follows. Let $\mathcal{R}_{\Gamma_i} : \Gamma \to \Gamma_i$ be the canonical point-wise restriction which maps full vectors defined on $\Gamma$ into vectors defined on $\Gamma_i$. Analogous to (1.9), the Schur complement matrix (1.12) can be written as the sum of elementary matrices

$$\mathcal{S} = \sum_{i=1}^{\mathcal{N}} \mathcal{R}_{\Gamma_i}^T \mathcal{S}_i \mathcal{R}_{\Gamma_i} \tag{1.14}$$

where

$$\mathcal{S}_i = \mathcal{A}_{\Gamma_i\Gamma_i} - \mathcal{A}_{\Gamma_i\mathcal{I}_i}\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}^{-1}\mathcal{A}_{\mathcal{I}_i\Gamma_i} \tag{1.15}$$

is a local Schur complement and is defined in terms of sub-matrices from the local Neumann matrix $\mathcal{A}_i$ given by (1.8). Notice that this form of the Schur complement has only one layer of interface unknowns between subdomains and allows for a straight-forward parallel implementation.

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

#### 1.3.3.1 The Neumann-Dirichlet preconditioner

When a symmetric constant coefficient problem is sub-divided into two non-overlapping domains such that the subdomains are exact mirror images, it follows that the Schur complement contribution from both the left and right domains is identical. That is, $\mathcal{S}_1 = \mathcal{S}_2$. Consequently, the inverse of either $\mathcal{S}_1$ or $\mathcal{S}_2$ are ideal preconditioners as the preconditioned linear system is well-conditioned, e.g. $\mathcal{S}\mathcal{S}_1^{-1} = 2I$. A factorization can be applied to the local Neumann problem (1.8) on $\Omega_1$ :

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

to obtain

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

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

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

so that each Krylov iteration solves a Dirichlet problem on $\Omega_2$ (to apply $\mathcal{S}_2$) followed by a Neumann problem on $\Omega_1$ to invert $\mathcal{S}_1$. The Neumann-Dirichlet preconditioner was introduced in [30].

Generalization of the Neumann-Dirichlet preconditioner to multiple domains can be done easily when a coloring of subdomains is possible such that subdomains of the same color do not share an interface. For Cartesian decomposition in 2D, a red-black coloring is enough and the preconditioner is just the sum of the inverses corresponding to the black subdomains:

$$\mathcal{S} = \sum_{i \in B} \mathcal{R}_{\Gamma_i}^T (\mathcal{S}_i)^{-1} \mathcal{R}_{\Gamma_i} \tag{1.16}$$

where $B$ corresponds to the set of all black subdomains.

#### 1.3.3.2 The Neumann-Neumann preconditioner

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

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

This motivates using the following preconditioner with multiple subdomains :

$$\mathcal{M}_{NN} = \sum_{i=1}^{\mathcal{N}} \mathcal{R}_{\Gamma_i}^T D_i \mathcal{S}_i^{-1} D_i \mathcal{R}_{\Gamma_i} \tag{1.17}$$

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

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

The simplest choice for $D_i$ is the diagonal matrix with entries equal to the inverse of the number of subdomains to which an unknown belongs. The Neumann-Neumann preconditioner was first discussed in [37] and further studied in [137] where different choices for weight matrices are discussed. It should be noted that the matrices $\mathcal{S}_i$ can be singular for internal subdomains because they correspond to pure Neumann problems. The Moore-Penrose pseudo-inverse is often used for the inverse local Schur complements in (1.17) but other choices are possible such as inverting $\mathcal{A}_i + \epsilon I$ where $\epsilon$ is a small shift.

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

## 1.4 Some background on Krylov subspace methods

### 1.4.1 Introduction

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

$$\mathcal{A}x = b \tag{1.18}$$

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

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

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

The various Krylov solvers differ in the constraints or optimality conditions associated with the computed solution. In the sequel, we describe in some details the GMRES method [129] where the solution selected in the Krylov space corresponds to the vector that minimizes the Euclidean norm of the residual. This method is well-suited for unsymmetric problems. We also briefly present the oldest Krylov techniques that is the Conjugate Gradient method, where the solution in the Krylov space is chosen so that the associated residual is orthogonal to the space.

Many other techniques exist that we will not describe in this section; we rather refer the reader to the books [75, 128].

In many cases, such methods converge slowly, or even diverge. The convergence of iterative methods may be improved by transforming the system (1.18) into another system which is easier to solve. A preconditioner is a matrix that realizes such a transformation. If $\mathcal{M}$ is a non-singular matrix which approximates $\mathcal{A}^{-1}$, then the transformed linear system

$$\mathcal{M}\mathcal{A}x = \mathcal{M}b \tag{1.19}$$

might be solved faster. The system (1.19) is preconditioned from the left, but one can also precondition from the right :

$$\mathcal{A}\mathcal{M}t = b. \tag{1.20}$$

Once the solution $t$ is obtained, the solution of the system (1.18) is recovered by $x = \mathcal{M}t$.

### 1.4.2 The unsymmetric problems

The Generalized Minimum RESidual (GMRES) method was proposed by Saad and Schultz in 1986 [129] for the solution of large non hermitian linear systems.

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

Let $x_0 \in \mathbb{C}^n$ be an initial guess for the linear system (1.18) and $r_0 = b - \mathcal{A}x_0$ be its corresponding residual. At step $k$, the GMRES algorithm builds an approximation of the solution of (1.18) under the form

$$x_k = x_0 + V_k y_k \tag{1.21}$$

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

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

The vector $y_k$ is determined so that the 2–norm of the residual $r_k = b - \mathcal{A}x_k$ is minimized over $x_0 + \mathcal{K}(\mathcal{A}, r_0, k)$. The basis $V_k$ for the Krylov subspace $\mathcal{K}(\mathcal{A}, r_0, k)$ is obtained via the well-known Arnoldi process [17]. The orthogonal projection of $\mathcal{A}$ onto $\mathcal{K}(\mathcal{A}, r_0, k)$ results in an upper Hessenberg matrix $H_k = V_k^H \mathcal{A} V_k$ of order $k$. The Arnoldi process satisfies the relationship

$$\mathcal{A}V_k = V_k H_k + h_{k+1,k} v_{k+1} e_k^T \tag{1.22}$$

where $e_k$ is the $k^{\text{th}}$ canonical basis vector. Equation (1.22) can be rewritten in a matrix form as

$$\mathcal{A}V_k = V_{k+1}\bar{H}_k$$

where

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

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

Let $v_1 = r_0/\beta$ where $\beta = \|r_0\|_2$. The residual $r_k$ associated with the approximate solution $x_k$ defined by (1.21) satisfies

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

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

$$\min_{y \in \mathbb{C}^k} \|\beta e_1 - \bar{H}_k y\|_2. \tag{1.23}$$

We denote by $y_k$ the solution of (1.23). Therefore, $x_k = x_0 + V_k y_k$ is an approximate solution of (1.18) for which the residual is minimized over $x_0 + \mathcal{K}(\mathcal{A}, r_0, k)$. The GMRES method owes its name to this minimization property that is its key feature as it ensures the decrease of the residual norm associated with the sequence of iterates.

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

We now briefly describe GMRES with right preconditioner. Let $\mathcal{M}$ be a square nonsingular $n \times n$ complex matrix, we define the right preconditioned linear system

$$\mathcal{A}\mathcal{M}t = b \tag{1.24}$$

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

The GMRES algorithm builds an approximation of the solution of (1.24) of the form

$$t_k = t_0 + V_k y_k \tag{1.25}$$

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

$$\mathcal{K}_k = \text{span}\left\{r_0, \mathcal{A}Mr_0, \ldots, (\mathcal{A}M)^{k-1}r_0\right\}$$

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

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

$$\mathcal{A}[Mv_1, \cdots, Mv_k] = \mathcal{AM}V_k = V_k H_k + h_{k+1,k} v_{k+1} e_k^H \quad (1.26)$$

where $e_k$ is the $k^{\text{th}}$ canonical basis vector. Equation (1.26) can be rewritten as

$$\mathcal{AM}V_k = V_{k+1}\bar{H}_k$$

where

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

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

Let $v_1 = r_0/\beta$ where $\beta = \|r_0\|_2$. The residual $r_k$ associated with the approximate solution defined by Equation (1.25) verifies

$$\begin{aligned} r_k &= b - \mathcal{AM}t_k = b - \mathcal{AM}(t_0 + V_k y_k) \\ &= r_0 - \mathcal{AM}V_k y_k = r_0 - V_{k+1}\bar{H}_k y_k \\ &= \beta v_1 - V_{k+1}\bar{H}_k y_k \\ &= V_{k+1}(\beta e_1 - \bar{H}_k y_k). \end{aligned} \quad (1.27)$$

Since $V_{k+1}$ is a matrix with orthonormal columns, the residual norm $\|r_k\|_2 = \left\|\beta e_1 - \bar{H}_k y_k\right\|_2$ is minimal when $y_k$ solves the linear least-squares problem (1.23). We will denote by $y_k$ the solution of (1.23). Therefore, $t_k = t_0 + V_k y_k$ is an approximate solution of (1.24) for which the residual is minimal over $\mathcal{K}_k$. We depict in Algorithm 1 the sketch of the Modified Gram-Schmidt (MGS) variant of the GMRES method with right preconditioner.

### 1.4.3 The symmetric positive definite problems

The Conjugate Gradient method was proposed in different versions in the early 50s in separate contributions by Lanczos [96] and Hestenes and Stiefel [85]. It becomes the method of choice for the solution of large sparse hermitian positive definite linear systems and is the starting point of the extensive work on the Krylov methods [130].

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

$$\mathcal{A}x = b \quad (1.28)$$

to be solved.

Let $x_0 \in \mathbb{C}^n$ be an initial guess for this linear system, $r_0 = b - \mathcal{A}x_0$ be its corresponding residual and $\mathcal{M}^{-1}$ be the preconditioner. The preconditioned conjugate gradient algorithm is classically described as depicted in Algorithm 2.

**Algorithm 1** Right preconditioned GMRES

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

**Algorithm 2** Preconditioned Conjugate Gradient

1: $k = 0$
2: $r_0 = b - \mathcal{A}x_0$
3: **for** $k = 0, 1, 2, \dots$ **do**
4: $\quad$Solve $\mathcal{M}z_k = r_k$
5: $\quad$**if** $k = 0$ **then**
6: $\quad\quad p_0 = z_0$
7: $\quad$**else**
8: $\quad\quad \beta_{k-1} = z_{(k-1)}^H r_{k-1}/z_{k-2}^H r_{k-2}$
9: $\quad\quad p_k = z_{k-1} + \beta_{k-1}p_{k-1}$
10: $\quad$**end if**
11: $\quad q_k = \mathcal{A}p_k$
12: $\quad \alpha_k = z_{k-1}^H r_{k-1}/p_k^H q_k$
13: $\quad x_k = x_{k-1} + \alpha_k p_k$
14: $\quad r_k = r_{k-1} - \alpha_k q_k$
15: $\quad$Exit if convergence is detected
16: **end for**

The conjugate gradient algorithm constructs the solution that makes its associated residual orthogonal to the Krylov space. A consequence of this geometric property is that it is also the minimum error solution in A-norm over the Krylov space $\mathcal{K}_k = \text{span}\left\{r_0, \mathcal{A}r_0, \dots, \mathcal{A}^{k-1}r_0\right\}$. It exists a rich literature dedicated to this method; for more details we, non-exhaustively, refer to [74, 108, 128] and the references therein.

We simply mention that the preconditioned conjugate gradient method written as depicted in Algorithm 2 enables us to still have short recurrence on the unpreconditioned solution.

### 1.4.4 Stopping criterion for convergence detection

The backward error analysis, introduced by Givens and Wilkinson [146], 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 (1.29) 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 [46, 87]. 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 [25].

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

$$\begin{aligned}\eta_{\mathcal{A},b}(x_k) &= \min_{\Delta\mathcal{A},\Delta b}\{\tau > 0 : \|\Delta\mathcal{A}\| \le \tau\|\mathcal{A}\|,\ \|\Delta b\| \le \tau\|b\| \\ &\qquad \text{and } (\mathcal{A}+\Delta\mathcal{A})x_k = b + \Delta b\} \\ &= \frac{\|\mathcal{A}x_k - b\|}{\|\mathcal{A}\|\|x_k\| + \|b\|} \qquad (1.29)\end{aligned}$$

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 $\Delta b$ 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

$$\begin{aligned}\eta_b(x_k) &= \min_{\Delta b}\{\tau > 0 : \|\Delta b\| \le \tau\|b\| \text{ and } \mathcal{A}x_k = b + \Delta b\} \\ &= \frac{\|\mathcal{A}x_k - b\|}{\|b\|}. \qquad (1.30)\end{aligned}$$

This latter criterion measures the norm of the smallest perturbations $\Delta b$ on $b$ (assuming that they are no perturbations on $A$) such that $x_k$ is the exact solution of $\mathcal{A}x_k = b + \Delta b$. Clearly we have $\eta_{\mathcal{A},b}(x_k) < \eta_b(x_k)$. It has been shown [59, 116] that GMRES with robust

orthogonalization schemes is backward stable with respect to a backward error similar to (1.29) with a different choice for the norms.

We mention that $\eta_{\mathcal{A},b}$ and $\eta_b$ are recommended in [25] when the concern related to the stopping criterion is discussed; the stopping criteria of the Krylov solvers we used for our numerical experiments are based on them.

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

1. $\mathcal{M}\mathcal{A}x = \mathcal{M}b$ for left preconditioning;
2. $\mathcal{A}\mathcal{M}y = b$ for right preconditioning;
3. $\mathcal{M}_2\mathcal{A}\mathcal{M}_1 y = \mathcal{M}_2 b$ for split preconditioning.

Consequently, the backward stability property holds for those preconditioned systems where the corresponding stopping criteria are depicted in Table 1.2. In particular, it can be

| | Left precond. | Right precond. | Split precond. |
|---|---|---|---|
| $\eta_{\mathcal{M},\mathcal{A},b}$ | $\frac{\lVert\mathcal{M}\mathcal{A}x-\mathcal{M}b\rVert}{\lVert\mathcal{M}\mathcal{A}\rVert\lVert x\rVert+\lVert\mathcal{M}b\rVert}$ | $\frac{\lVert\mathcal{A}\mathcal{M}t-b\rVert}{\lVert\mathcal{A}\mathcal{M}\rVert\lVert x\rVert+\lVert b\rVert}$ | $\frac{\lVert\mathcal{M}_2\mathcal{A}\mathcal{M}_1t-\mathcal{M}_2b\rVert}{\lVert\mathcal{M}_2\mathcal{A}\mathcal{M}_1\rVert\lVert t\rVert+\lVert\mathcal{M}_2b\rVert}$ |
| $\eta_{\mathcal{M},b}$ | $\frac{\lVert\mathcal{M}\mathcal{A}x-\mathcal{M}b\rVert}{\lVert\mathcal{M}b\rVert}$ | $\frac{\lVert\mathcal{A}\mathcal{M}t-b\rVert}{\lVert b\rVert}$ | $\frac{\lVert\mathcal{M}_2\mathcal{A}\mathcal{M}_1t-\mathcal{M}_2b\rVert}{\lVert\mathcal{M}_2b\rVert}$ |

Table 1.2: Backward error associated with preconditioned linear system in GMRES.

seen that for all but the right preconditioning and $\eta_{\mathcal{M},b}$, the backward error depends on the preconditioner. For right preconditioning, the backward error $\eta_b$ is the same for the preconditioned and unpreconditioned system because $\lVert\mathcal{A}\mathcal{M}t - b\rVert = \lVert\mathcal{A}x - b\rVert$. This is the main reason why for all our numerical experiments with GMRES we selected right preconditioning. A stopping criterion based on $\eta_b$ enables a fair comparison among the tested approaches as the iterations are stopped once the approximations have all the same quality with respect to this backward error criterion.

## 1.5 Sparse linear solvers on modern supercomputers

### 1.5.1 Manycore architectures

For many decades, the performance of CPUs had increased mostly thanks to higher clock frequencies – a consequence of higher degrees of Instruction Level Parallelism (ILP) – and deeper memory hierarchies. As these traditional techniques reached the point of diminishing returns and started imposing unsustainable levels of power consumption and heat dissipation, the microprocessors industry abruptly steered towards Thread Level Parallelism (TLP). As a consequence, the design and development of microprocessors started focusing

on adding more processing units (or *cores*) rather than increasing the single-threaded performance. A few years from then, multicore processors are nowadays ubiquitous, and the evolution of computers is driven by a run towards higher numbers of cores per chip.

This trend was largely anticipated by Graphical Processing Units (GPUs). Originally designed for processing images, GPUs were highly specialized computing devices meant to handle a particular class of applications with well defined characteristics, large computational requirements and substantial, fine-grained parallelism. These features naturally presented GPU devices as a compelling alternative to traditional microprocessors to the scientific computing community. First attempts to use GPUs for scientific applications can be dated back to the 90's [115]. GPU computing has been considered an academic exercise for long time and only recently, also thanks to the ever increasing interest of the scientific community, GPU devices started evolving into powerful programmable processors. As a result of this evolution, modern GPUs can substantially outperform high-end, multicore CPUs both in terms of data processing rate and memory bandwidth: theTesla K40 model produced by NVIDIA has a peak performance of 1.43 Tflops/s for double-precision operations (4.29 Tflop/s for single-precision) and a memory bandwidth of 288 GBytes/s, theoretically almost a factor of six times faster than the Intel Xeon Processor E5-2680 processor available at the same time. Moreover, if porting general purpose codes on GPUs required a considerable programming effort mostly due to the lack of tools and interfaces, Application Programming Interfaces (API) for GPUs such as CUDA or OpenCL have been rapidly evolving in the last few years: GPU programs can now be written in familiar programming languages (such as C or Fortran) according to a Single Program Multiple Data (SPMD) parallel programming model which opened the way for the collective effort that is commonly known under the name of *General Purpose GPU* (GPGPU) computing [114].

On the other side, CPU manufacturers started designing and producing accelerator boards with features that resemble those of GPUs: extremely high core-count, Thread Level Parallelism, memory bandwidth and limited power consumption. One example of this trend is represented by the Intel MIC (Many Integrated Cores) boards such as the Xeon Phi, commercialized in 2012, whose peak performance can exceed 1 Tflop/s. These boards share the same instruction set architecture as traditional x86 CPUs – a very desirable feature that should ease the development of applications. In addition, Intel also released a MIC version of some basic scientific computing libraries such as BLAS or LAPACK which readily provide considerable gains to applications that use them.

Accelerators have thus become more attractive alternatives to traditional CPUs and are considered to cost less money-per-flop and to consume less watts-per-flop. As a result, more production-quality codes are developed nowadays for these devices. A striking evidence of the success of accelerators in scientific computing is the MIC based computer Tianhe-2, installed at the National Super Computer Center in Guangzhou, China, which was ranked number one in the November 2015 Top500 list[2].

Although accelerators can achieve considerable performance on a specific class of applications, they can still hardly be considered general purpose computing devices. Furthermore, the efficient programming of heterogeneous systems equipped with accelerators is still a hard challenge. For example, only 60% of the theoretical peak performance can be achieved

[2]The Top500 (http://top500.org) ranking lists the 500 most powerful supercomputers in the world.

on Titan as compared to the 80% or more obtained on CPU based supercomputers of the same segment in the Top500 list.

Several software libraries for dense linear algebra have been produced. For instance, the MAGMA [2] project at University of Tennessee Knoxville has been extended in collaboration with the HiePACS and STORM Inria projects in order to handle heterogeneous nodes and clusters[3]. The most common dense linear algorithms are extremely rich in computation and exhibit a very regular pattern of access to data which makes them extremely good candidates for execution on accelerators. The most common sparse linear algebra algorithms are the methods for the solution of linear systems which, contrary to the dense linear algebra variants, usually have irregular, indirect memory access patterns that adversely interact with typical accelerator throughput optimizations. So, achieving a high effciency on accelerators for numerical sparse linear solvers is more challenging. These solution methods can be roughly classified in two families:

- iterative methods: in their basic, unpreconditioned version, they are very easily parallelizable and for this reason they have been the object of research on GPU scientific computing. However, the accelerator implementation of efficient preconditioners is extremely complicated, and the lack of such implementations renders iterative solvers for accelerators insufficiently reliable and robust at present. Furthermore, iterative methods are based on operations such as the sparse matrix-vector product characterized by a very low computation-to-communication ratio which can considerably limit their performance and scalability.
- direct methods: for algorithms belonging to this family, such as sparse matrix factorization methods, the computation is commonly casted in terms of operations on a sparse collection of dense matrices which makes them much denser in floating-point operations and opens up opportunities for massive multithreaded parallelization and porting on accelerators. Nonetheless, sparse matrix factorization methods have extremely complicated data access patterns which render their high-performance implementation on accelerator devices complicated.

### 1.5.2 Evolutionary parallel programming paradigms

From the 90's where distributed memory parllel computers have progressively replaced the vector processor based mainframes dedicated to intensive scientific computing, most of the parallel simulation codes have been developed using the message passing programing model. At that time a Message Passing Interface (MPI [117]) has been defined by the computational science community to enable the portability of the new designed codes. It was critical to ensure the return of the human effort invested in these code porting since this development effort was significant. During more than a decade the substainability of the Moore's law was ensured by adding more nodes on clusters that were mainly exploited by MPI based large scale computations. Due to the progresses of electronic integration and to reduce the cost of the network interconnects, clusters of symmetric multi-processors (SMP) first appear in the 2000's. The architecture of these computers exhibit a logical and physical memory

[3] https://project.inria.fr/chameleon/

hierarchy where the first level is composed by the memory of the SMP and the second level corresponds to the physically distributed memory. This memory structure naturally matches hybrid programming model with multithread run within the SMP nodes and MPI processes running between the SMPs. The multithreading is managed either explicitly through fine POSIX programming or via OpenMP directives. Such two levels of parallelism did not received much attention in particular by explicit simulation codes where flat MPI ensured a good control of data locality thanks to the explicit management of the data partitioning. However its benefit first revealed in the algorithms where the message passing exchanges had a strong impact on the design of the numerical algorithms, on the memory consumption or on the numerical behavior. A first example can be borrowed from sparse direct numerical linear algebra with the two sparse codes PaStiX [81] and WSMP [77]. The attraction of such hybrid programming increases significantly with the advent of multicores where communication cost of the flat MPI became prohibitive. The next evolution that impacted the programming paradigms was the introduction of accelerator such as GPGPU. Not only the memory accesses are non uniform but also the computation capability becomes heterogeneous. The evolution to take advantage of this new computing power has first been to explicitly off-load on GPGPU some of the numerical kernels from larger applications.

In the last decades, the evolution of the hardware technology has been accomodated by an evolutionary path of the programming methodologies where the complexity of underlying hardware was directly exposed to the application designers though a stack of libraries/languages such as MPI, POSIX and CUDA. The algorithm designers had to manage the low level data management while developping more sophisticated numerical schemes, which led to an complex interleaving of numerical and computer science applications making the resulting codes complex to manage and to upgrade.

### 1.5.3 Revolutionary programming paradigms

Computing platform hardware has dramatically evolved ever since the computer science began, always striving to provide new convenient accelerating features for achieving higher computational power. Each new accelerating hardware feature inevitably leaves programmers to decide whether to make their application dependent on that feature (and break compatibility) or not (and miss the potential benefit), or even to handle both cases (at the cost of extra management code in the application). This common problem is known as the *performance portability issue*. A discruptive approach to tackle the code development complexity associated with emerging heterogeneous manycore platforms is to consider *a task based programming paradigm*. Such an approach enables to have a high level expressivity of the algorithms while ensuring the performance portability thanks to a *runtime system*, a third party layer of the system stack.

The first purpose of runtime systems is thus to provide *abstraction*. Runtime systems offer a uniform programming interface for a specific subset of hardware (e.g., OpenGL or DirectX are well-established examples of runtime systems dedicated to hardware-accelerated graphics) or low-level software entities (e.g., POSIX-thread implementations). They are designed as thin user-level software layers that complement the basic, general purpose functions provided by the operating system calls. Applications then target these uniform

programming interfaces in a portable manner. Low-level, hardware dependent details are hidden inside runtime systems. The adaptation of runtime systems is commonly handled through drivers. The abstraction provided by runtime systems thus enables portability. Abstraction alone is however not enough to provide portability of performance, as it does nothing to leverage low-level-specific features to get increased performance.

Consequently, the second role of runtime systems is to *optimize* abstract application requests by dynamically mapping them onto low-level requests and resources as efficiently as possible. This mapping process makes use of scheduling algorithms and heuristics to decide the best actions to take for a given metric and the application state at a given point in its execution time. This allows applications to readily benefit from available underlying low-level capabilities to their full extent without breaking their portability. Thus, optimization together with abstraction allows runtime systems to offer portability of performance.

In the specific case of parallel work mapping, other approaches have occasionally been adopted instead of using runtimes. Many scientific applications and libraries, including linear system solvers, integrate their own, customized dynamic scheduling algorithms or even resort to static scheduling techniques, either for historical reasons, or to avoid the potential overhead of an extra runtime layer.

However, as multicore processors densify, as cache and memory hierarchies deepen, the resulting increase in complexity now makes the use of work-mapping runtime systems virtually unavoidable. Such work-mapping runtime systems take elementary task descriptions and dependencies as input and are responsible for dynamically scheduling the tasks on available computing units so as to minimize a given cost function (usually the execution time) under some pre-defined set of constraints.

Work-mapping runtime systems themselves are now facing new challenges with the recent move of the high performance community towards the use of specialized accelerating cores together with traditional general-purpose cores. They not only have to decide about the interest (or not) to use some specific hardware features, but also have to decide whether some entire application tasks should rather be performed on an accelerated core or is better left on a standard core.

In the case where specialized cores are located on an expansion card having its own memory (e.g., most existing GPUs), the input data of a task have to be copied from central memory to the card memory before the task can be run. The output results must also be copied back to the central memory once the task computation is complete. The cost of copying data between central memory and accelerator memory is not negligible. This cost, as well as data dependencies between tasks, must therefore also be taken into account by the scheduling algorithms when deciding whether to offload a given task, to avoid unnecessary data transfers. Transfers should also be done in advance and asynchronously so as to overlap communication with computation.

An original and effective runtime system is StarPU [19] developed by the STORM Inria's project. Within this framework, the algorithm is described as a sequential task flow with data dependencies expressed through read/write attributes provided for each task parameters. This task flow is internally translated by StarPU into a Directed Acyclic Graph (DAG) used to optimaly scheduled the task on the different computing units.

## 1.6 Positioning of the thesis

The work developped in the context of this thesis addresses the design and the implementation of an hybrid iterative/direct solver, namely MaPHyS. In Chapter 2, we first extend in a pure algebraic formalism the non-overlapping domain decomposition technique described in this introductory chapter. We describe the algebraic preconditioner that led to the first implementation of the MaPHyS solver used as starting point of this study. We detail in this chapter an evolutionary approach, where we first consider a 2-level parallel implementation based on parallel multithreaded libraries for the dense and sparse linear algebra calculation within each subdomain while communication between the subdomains are implemented using MPI. In particular, we detail how the interoperability of different multithreaded libraries has been mastered. On challenging large linear systems arising from 3D modeling, we illustrate that such a 2-level parallelism enables to compromise between the numerical and parallel efficiency of the hybrid solver. We consider test problems from our main matrix collections (Table 2.1) in Section 2.4 and test examples from geoscience applications in Section 2.5. We also compare the performance of our solver with a state-of-the-art hybrid solver PDSLin [149], that uses a full MPI based parallelism and that relies on an approximate global Schur complement preconditioner described in Section 2.7.

In Chapter 3 we consider a more disruptive task-based design. In a first step, we still rely on MPI communicating between subdomains, but subdomains are internally processed with task-based solvers instead of multithreade solvers. We propose a prototype extension of MaPHyS based on that design. In a second step, we tackle a full task-based paradigm where the architecture is fully abstracted, the algorithm being entirely expressed in terms of task graph. In that latter case, we have not implemented an entire hybrid solver. Instead, we have studied the attactivity of the approach on one of the core numerical kernels of hybrid solvers, the Conjugate Gradient method; that is the Krylov subspace method used by MaPHyS for symmetric positive linear systems. We present the high performance we achieve following such a design both on multi-GPU and multicore architectures and present preliminary results on heterogenous nodes.

Finally, we describe possible tracks for future research and development of MaPHyS.

Chapter 2

# A hierarchical hybrid sparse linear solver for multicore platforms

## 2.1 Introduction

The solution of large sparse linear systems of the form $\mathcal{A}x = b$ where $\mathcal{A}$ is a sparse matrix, $b$ is a vector and $x$ the unknown vector lies at the heart of many numerical simulations and appears often in the inner-most loops of intensive simulation codes. Over the past decade or so, several teams have been developing innovative numerical algorithms to exploit advanced high performance, large-scale parallel computers to solve these equations efficiently. There are two basic approaches for solving linear systems of equations: direct methods and iterative methods. Those two large classes of methods have somehow opposite features with respect to their numerical and parallel implementation efficiencies.

Direct methods based on the Gaussian elimination are the oldest method 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 efficient 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 [87]. 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 3-dimensional partial differential equations (PDEs). Furthermore, the linear systems involved in the numerical simulation of complex phenomena result from 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.

As explained in the previous chapter, iterative methods on the other hand, generate sequences of approximations to the solution either through fixed point schemes or via search in Krylov subspaces [128]. The best known representatives of these latter numerical techniques are the Conjugate Gradient [85] and the GMRES [129] methods. 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. In this work, we consider stopping criteria based on the backward error analysis [25, 59, 75] introduced in Section 1.2.

Our approach to 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 paralelization. The direct part provides its favorable numerical properties. Furthermore, this combination enables us to naturally exploit several levels of parallelism that logically match the hardware feature of multicore platforms as it will be described in details in this chapter. In particular, we 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 ideas are not new. They have been used to design domain decomposition techniques for the numerical solution of PDEs [107, 119, 135] as briefly introduced in Section 1.3. In our work, we consider domain decomposition techniques extended to general unstructured linear systems. More precisely, we consider numerical techniques based on a non-overlapping decomposition of the graph associated with the sparse matrices. The vertex separator, constructed using graph partitioning [52, 90], defines the interface variables that will be solved iteratively using a Schur complement approach, while the variables associated with the interior subgraphs will be handled by a sparse direct solver. Although the Schur complement system is usually more tractable than the original problem by an iterative technique, preconditioning treatment is still required. For that purpose, we developed parallel preconditioners and designed hierarchical parallel implementations.

This chapter is organized as follows. First, in Section 2.2 we describe the numerical technique implemented by the hybrid solver, MaPHyS, that is the focus of this work and give a brief overview of the other hybrid solvers in the literature. Section 2.3 is devoted to the algorithmic description of its parallel implementations. We first present the baseline parallel implementation that assigns one subgraph per processing unit. We then describe a more flexible implementation that enables us to decorrelate the number of subgraphs from the number of processing unit to enhance the numerical performance. In that latter implementation we can keep the number of subgraphs low while handling each subgraph with multiple processing units, introducing two levels of parallelism. Such an implementation enables us to exploit the natural parallelism of the subgraphs but also parallelism within each subgraph. A performance analysis on test examples from our main matrix collections presented in Section 2.4 as well as on geoscience challenging test cases in Section 2.5. We also assess the performance of MaPHyS with respect to the quality of the partitioning in Section 2.6. Achieved comparative performance analysis with the state-of-the-art PDSLin hybrid solver is provided in Section 2.7 before concluding (Section 2.8).

## 2.2 Parallel algebraic non-overlapping domain decomposition methods

In this section we describe the design of the hybrid solvers based on a non-overlapping domain decomposition. For the sake of simplicity, we assume that $\mathcal{A}$ has a symmetric pattern. First we present the main ideas used in MaPHyS (Section 2.2.1), followed by a brief overview of the method used in the other hybrid solvers.

### 2.2.1 Governing ideas

Let $\mathcal{A}x = b$ be the linear problem and $\mathcal{G} = \{V, E\}$ the adjacency graph associated with $\mathcal{A}$. In this graph, each vertex is associated with a row or column of the matrix $\mathcal{A}$ and it exists an edge between the vertices $i$ and $j$ if the entry $a_{i,j}$ is non zero. In the sequel, to facilitate the exposure and limit the notation we voluntarily mix a vertex of $\mathcal{G}$ with its index depending on the context of the description. The governing idea behind substructuring or Schur complement methods is to split the unknowns in two categories: interior and interface vertices. We assume that the vertices of the graph $\mathcal{G}$ are partitioned into $\mathcal{N}$ disconnected subgraphs $\mathcal{I}_1$, ..., $\mathcal{I}_\mathcal{N}$ separated by the global vertex separator $\Gamma$. We also decompose the vertex separator $\Gamma$ into non-disjoint subsets $\Gamma_i$, where $\Gamma_i$ is the set of vertices in $\Gamma$ that are connected to at least one vertex of $\mathcal{I}_i$. 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 domain decomposition in a finite element framework, $\Omega_i = \mathcal{I}_i \cup \Gamma_i$ will be referred to as a subdomain with internal unknowns $\mathcal{I}_i$ and interface unknowns $\Gamma_i$. If we denote $\mathcal{I} = \cup \mathcal{I}_i$ and order vertices in $\mathcal{I}$ first, we obtain the following block reordered linear system

$$\begin{pmatrix} \mathcal{A}_{\mathcal{I}\mathcal{I}} & \mathcal{A}_{\mathcal{I}\Gamma} \\ \mathcal{A}_{\Gamma\mathcal{I}} & \mathcal{A}_{\Gamma\Gamma} \end{pmatrix} \begin{pmatrix} x_\mathcal{I} \\ x_\Gamma \end{pmatrix} = \begin{pmatrix} b_\mathcal{I} \\ b_\Gamma \end{pmatrix} \tag{2.1}$$

where $x_\Gamma$ contains all unknowns associated with the separator and $x_\mathcal{I}$ contains the unknowns associated with the interiors. Because the interior vertices are only connected to either interior vertices in the same subgraph or with vertices in the interface, the matrix $\mathcal{A}_{\mathcal{I}\mathcal{I}}$ has a block diagonal structure, where each diagonal block corresponds to one subgraph $\mathcal{I}_i$. Eliminating $x_\mathcal{I}$ from the second block row of Equation (2.1) leads to the reduced system

$$\mathcal{S} x_\Gamma = f \tag{2.2}$$

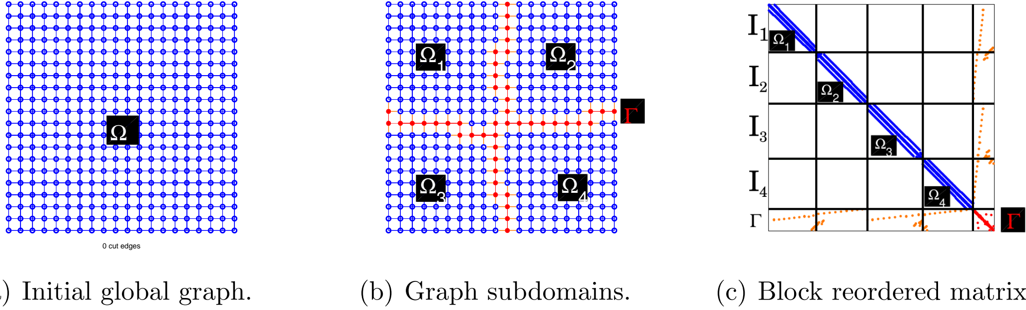
where

$$\mathcal{S} = \mathcal{A}_{\Gamma\Gamma} - \mathcal{A}_{\Gamma\mathcal{I}} \mathcal{A}_{\mathcal{I}\mathcal{I}}^{-1} \mathcal{A}_{\mathcal{I}\Gamma} \text{ and } f = b_\Gamma - \mathcal{A}_{\Gamma\mathcal{I}} \mathcal{A}_{\mathcal{I}\mathcal{I}}^{-1} b_\mathcal{I}. \tag{2.3}$$

The matrix $\mathcal{S}$ is referred to as the *Schur complement matrix*. This reformulation leads to a general strategy for solving (2.1). Specifically, an iterative method can be applied to solve (2.2). Once $x_\Gamma$ is known, $x_\mathcal{I}$ can be computed with one additional solve for the interior unknowns via

$$x_\mathcal{I} = \mathcal{A}_{\mathcal{I}\mathcal{I}}^{-1} \left( b_\mathcal{I} - \mathcal{A}_{\mathcal{I}\Gamma} x_\Gamma \right).$$

We illustrate in Figure 2.1(a) all these notations for a decomposition into 4 subdomains. The local interiors are disjoint and form a partition of the interior $\mathcal{I} = \sqcup \mathcal{I}_i$ (blue vertices in




(a) Initial global graph. (b) Graph subdomains. (c) Block reordered matrix.

Figure 2.1: Domain decomposition into four subdomains $\Omega_1, \ldots, \Omega_4$. The initial domain $\Omega$ may be algebraically represented with the graph $\mathcal{G}$ associated to the sparsity pattern of matrix $\mathcal{A}$ (a). The local interiors $\mathcal{I}_1, \ldots, \mathcal{I}_\mathcal{N}$ form a partition of the interior $\mathcal{I} = \sqcup \mathcal{I}_i$ (blue vertices in (b)). They interact with each others through the interface $\Gamma$ (red vertices in (b)). The block reordered matrix (c) has a block diagonal structure for the variables associated with the interior $\mathcal{A}_{\mathcal{I}\mathcal{I}}$.

Figure 2.1(b)). It is not necessarily the case for the boundaries. Indeed, two subdomains $\Omega_i$ and $\Omega_j$ may share part of their interface ($\Gamma_i \bigcap \Gamma_j \neq \emptyset$), such as $\Omega_1$ and $\Omega_2$ in Figure 2.1(b) which share eleven vertices. Altogether, the local boundaries form the overall interface $\Gamma = \cup \Gamma_i$ (red vertices in Figure 2.1(b)), which is not a disjoint union. Because interior vertices are only connected to vertices of their subset (either on the interior or on the boundary), matrix $\mathcal{A}_{\mathcal{I}\mathcal{I}}$ associated to the interior has a block diagonal structure, as shown in Figure 2.1(c). Each diagonal block $\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}$ corresponds to a local interior.

While the Schur complement system is significantly smaller and better conditioned than the original matrix $\mathcal{A}$, it is important to consider further preconditioning when employing a Krylov method. We introduce the general form of the preconditioner considered in MaPHyS. The preconditioner presented below was originally proposed in [43] and successfully applied to large problems in real life applications in [70, 78]. To describe the main preconditioner in MaPHyS, we define $\bar{\mathcal{S}}_i = \mathcal{R}_{\Gamma_i} \mathcal{S} \mathcal{R}_{\Gamma_i}^T$, that corresponds to the restriction of the Schur complement to the interface $\Gamma_i$. If $\mathcal{I}_i$ is a fully connected subgraph of $\mathcal{G}$, the matrix $\bar{\mathcal{S}}_i$ is dense.

With these notations the Additive Schwarz preconditioner reads

$$\mathcal{M}_{AS} = \sum_{i=1}^{\mathcal{N}} \mathcal{R}_{\Gamma_i}^T \bar{\mathcal{S}}_i^{-1} \mathcal{R}_{\Gamma_i}. \tag{2.4}$$

We notice that this preconditioner has a form similar to the Neumann-Neumann preconditioner [37, 57](See Section 1.3.3.2), but in the SPD case $\mathcal{M}_{AS}$ is always defined and SPD (as $\mathcal{S}$ is SPD [43]); which is not always the case for Neumann-Neumann.

If we considered a planar graph partitioned into horizontal strips (1D decomposition),

the resulting Schur complement matrix has a block tridiagonal structure as depicted in (2.5)

$$\mathcal{S} = \begin{pmatrix} \ddots & & & \\ & \boxed{\begin{matrix} \mathcal{S}_{k,k} & \mathcal{S}_{k,k+1} \\ \mathcal{S}_{k+1,k} & \mathcal{S}_{k+1,k+1} \end{matrix}} & & \\ & & \boxed{\begin{matrix} \mathcal{S}_{k+1,k+1} & \mathcal{S}_{k+1,k+2} \\ \mathcal{S}_{k+1,k+2} & \mathcal{S}_{k+2,k+2} \end{matrix}} & \\ & & & \ddots \end{pmatrix}. \tag{2.5}$$

For that particular structure of $\mathcal{S}$, the submatrices in boxes correspond to the $\bar{\mathcal{S}}_i$ local restriction of the Schur $\mathcal{S}$. 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 [42, 122] for the solution of general sparse linear systems. Because of this link, the preconditioner defined by (2.4) is referred to as algebraic additive Schwarz for the Schur complement.

### 2.2.2 Related work

With the need of solving ever larger sparse linear systems while maintaining numerical robustness, multiple variants for computing the preconditioner for the Schur complement of such hybrid solvers have been proposed. PDSLin [99], ShyLU [123] and Hips [64] first perform an exact[1] factorization of the interior of each subdomain concurrently. PDSLin and ShyLU then compute the preconditioner with a two-fold approach. First, an approximation $\widetilde{\mathcal{S}}$ of the (global) Schur complement $\mathcal{S}$ is computed. Second, this approximate Schur complement $\widetilde{\mathcal{S}}$ 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 $\widetilde{\mathcal{S}}$ (see details in Section 2.6), ShyLU [123] also implements a structure-based approach for discarding values named *probing* and that was first proposed to approximate interfaces in DDM [48]. Instead of following such a two-fold approach, Hips [64] forms the preconditioner by computing a global ILU factorization based on the multi-level scheme formulation from [82]. Finally, the object of this thesis, MaPHyS [72], computes an additive Schwarz preconditioner for the Schur complement as previously described in Section 2.2.1.

To ensure numerical robustness while exploiting all the processors of a platform, an important effort has been devoted to propose two levels of parallelism for these solvers. Relying on the SuperLU_DIST [100] distributed memory sparse direct solver, PDSLin implements a 2-level MPI (MPI+MPI) approach with finely tuned intra- and inter-subdomain load balancing [149]. A similar MPI+MPI approach has been assessed for additive Schwarz preconditioning in a prototype version of MaPHyS [73], relying on the Mumps [13, 14] and ScaLAPACK [31] sparse direct and dense distributed memory solvers, respectively. On the contrary, expecting a higher numerical robustness thanks to multi-level preconditioning, Hips associates multiple subdomains to a single process and distributes the subdomains to

[1] There are also options for computing Incomplete LU (ILU) factorizations of the interiors but the related descriptions are out the scope of this paper.

the processes in order to maintain load balancing [64]. Finally, especially tuned for modern multicore platforms, ShyLU implements a 2-level MPI+thread approach [123].

Whereas PDSLin and ShyLU can be virtually turned into to a pure direct method if no dropping is performed, and whereas Hips may expect robustness by relying on a multilevel scheme, additive Schwarz preconditioners are extremely local. As a result, their computation is potentially much more parallel (and scalable), but their application may lead to a dramatic increase of the number of iterations (or even to non convergence) if the number of subdomains becomes too large. The objective of the present study is to assess whether a 2-level parallel approach allows additive Schwarz preconditioning for Schur Complement methods to achieve an efficient trade-off between numerical robustness and performance on modern hierarchical multicore platforms. Following the paralelization scheme adopted in [123], we have designed a MPI+thread approach (Section 2.3) to cope with these hardware trends. Contrary to the MPI+MPI approach investigated in [73], such a paralelization allows for a better usage of the memory (*e.g.* symbolic data structures such as the elimination tree used within a direct solver are shared between threads of a same process) and a better exploitation of the CPU cores local to a node at each step of the parallel computation [3, 63, 97].

## 2.3 Design of parallel implementations of MaPHyS

MaPHyS is based on an algebraic domain decomposition whose primary motivation is to naturally exploit some parallelism between the computation performed on each subproblem of the decomposition. In this chapter, we describe the choices made on the data decomposition and the numerical algorithms that handled these data. We first describe the baseline (starting point of this thesis) approach in Section 2.3.1 where parallelism is only exploited between subdomains. Such an approach strongly constrains the number of cores to be equal to the number of subdomains. This 1 level parallel implementation only relies on the message passing paradigm and suffers from a lack of flexibility that strongly interleaves the numerical and parallel behaviors. In order to relax this constraint, we have designed a 2-level parallel implementation that exploits multithreading within the computation of each subdomain. Such a 2-level approach for parallel hybrid solver implementation has already been investigated [78,88] via complex MPI-MPI implementations; those implementations do not fully match the natural memory hierarchy of the current HPC platforms and motivate this contribution.

We first introduce the baseline 1-level parallelism implementation of MaPHyS in Section 2.3.1 and then explain how we have extended it to design a two level MPI+thread version of the solver in Section 2.3.2.

### 2.3.1 Baseline MPI implementation of the MaPHyS solver

Based on the decomposition of $\mathcal{G}$ introduced in Section 2.2.1, we can define a decomposition of the matrix $\mathcal{A}$ where each sub-matrix is associated with a subdomain and is allocated to one MPI process. Note that due to the overlap between local interfaces $\Gamma_i$, a special

attention has to be paid to the decomposition of $\mathcal{A}_{\Gamma\Gamma}$ as its entries are shared between different processes. In that respect the matrix entries of $\mathcal{A}_{\Gamma\Gamma}$ 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 $\mathcal{A}^w_{\Gamma_i\Gamma_i}$ that satisfies $\mathcal{A}_{\Gamma\Gamma} = \sum_{i=1}^{\mathcal{N}} \mathcal{R}^T_{\Gamma_i} \mathcal{A}^w_{\Gamma_i\Gamma_i} \mathcal{R}_{\Gamma_i}$, where $\mathcal{R}_{\Gamma_i} : \Gamma \to \Gamma_i$ is again the canonical point-wise restriction which maps full vectors defined on $\Gamma$ into vectors defined on $\Gamma_i$. For instance, the twelve red edges shared by subdomains $\Omega_1$ and $\Omega_2$ in Figure 2.1(b) may get a weight $\frac{1}{2}$ as they are shared by two subdomains. In matrix terms, a subdomain $\Omega_i$ may then be represented by the *local matrix* $\mathcal{A}_i$ defined by

$$\mathcal{A}_i = \begin{pmatrix} \mathcal{A}_{\mathcal{I}_i\mathcal{I}_i} & \mathcal{A}_{\mathcal{I}_i\Gamma_i} \\ \mathcal{A}_{\Gamma_i\mathcal{I}_i} & \mathcal{A}^w_{\Gamma_i\Gamma_i} \end{pmatrix}. \tag{2.6}$$

The global Schur complement matrix $\mathcal{S}$ from (2.2) can then be written as the sum of elementary matrices

$$\mathcal{S} = \sum_{i=1}^{\mathcal{N}} \mathcal{R}^T_{\Gamma_i} \mathcal{S}_i \mathcal{R}_{\Gamma_i} \tag{2.7}$$

where

$$\mathcal{S}_i = \mathcal{A}^w_{\Gamma_i\Gamma_i} - \mathcal{A}_{\Gamma_i\mathcal{I}_i} \mathcal{A}^{-1}_{\mathcal{I}_i\mathcal{I}_i} \mathcal{A}_{\mathcal{I}_i\Gamma_i} \tag{2.8}$$

is the *local Schur complement* associated to subdomain $\Omega_i$. This local expression allows for computing local Schur complements independently from each other.

The $\bar{\mathcal{S}}_i$'s that are involved in the definition of $\mathcal{M}_{AS}$ can actually be built within this data distribution from the $\mathcal{S}_i$'s. Let us simply describe this calculation on a simple example for a given subdomain $\Omega_i$. In Figure 2.2, we depict an internal subdomain $\Omega_i$ together with its

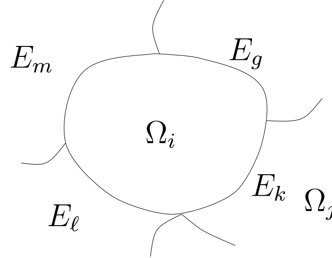



Figure 2.2: An internal subdomain.

interface $\Gamma_i = E_m \cup E_g \cup E_k \cup E_\ell$. The local Schur complement matrix associated with $\Omega_i$ is dense and has the following $4 \times 4$ block structure

$$\mathcal{S}_i = \begin{pmatrix} \mathcal{S}^{(i)}_{mm} & \mathcal{S}_{mg} & \mathcal{S}_{mk} & \mathcal{S}_{m\ell} \\ \mathcal{S}_{gm} & \mathcal{S}^{(i)}_{gg} & \mathcal{S}_{gk} & \mathcal{S}_{g\ell} \\ \mathcal{S}_{km} & \mathcal{S}_{kg} & \mathcal{S}^{(i)}_{kk} & \mathcal{S}_{k\ell} \\ \mathcal{S}_{\ell m} & \mathcal{S}_{\ell g} & \mathcal{S}_{\ell k} & \mathcal{S}^{(i)}_{\ell\ell} \end{pmatrix} \tag{2.9}$$

where each block accounts for the interactions between the unknowns on the edges of its interface. The matrix $\bar{\mathcal{S}}_i$ can be built from the local Schur complement $\mathcal{S}_i$ by assembling

its diagonal blocks thanks to a few neighbour to neighbour communications. For instance, the diagonal blocks of $\mathcal{S}_i$ associated with the edge interface $E_k$, depicted in Figure 2.2, 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

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

The original idea of hybrid methods based on DDM consists in subdividing the graph into subgraphs that are individually mapped to one process. This approach is referred to as the classical parallel implementation.

With all these components, the classical parallel implementation of MaPHyS can be decomposed into four main phases:

- *the partitioning step* consists of partitioning the adjacency graph $\mathcal{G}$ of $\mathcal{A}$ into several subdomains and distribute the $\mathcal{A}_i$ to different processes;
- *the factorization of the interiors* and the computation of the local Schur complement $\mathcal{S}_i$ using $\mathcal{A}_i$;
- *the setup of the preconditioner* by assembling diagonal blocks of $\mathcal{S}_i$ via a few neighbour to neighbour communications and factorization of $\bar{\mathcal{S}}_i$;
- *the solve step* where a parallel preconditioned Krylov method is performed on the reduced system (Equation 2.2) to compute $x_{\Gamma_i}$ followed by the back solve on the interior to compute $x_{\mathcal{I}_i}$.

Comparing to other hybrid solvers, MaPHyS has one major implementation difference. All other solvers are an extension of an already existing direct sparse solver. With this strategy, the code of the direct sparse solver can be optimized for the needs of the hybrid solver. MaPHyS, on the other hand, is implemented in a "black box" strategy. An external library is used for each individual step of the algorithm. A set of needed functionalities are defined for each step, so any library that provides these functionalities can be used. With this strategy, MaPHyS can use several different libraries for each step. Also it will automatically benefit from new developments of each one that is used.

In the rest of this section, a more detailed explanation for each step is presented and the libraries that MaPHyS is able to exploit for each one of them are presented.

#### 2.3.1.1 *Partitioning step*

The ultimate objective of the partitioning in MaPHyS is to construct subdomains with balanced sizes for the interiors (similar cardinality of the $\mathcal{I}_i$'s) and the interfaces (similar cardinality of the $\Gamma_i$'s). Such a partitioning constraint is rather uncommon and no graph

partitioner implements a heuristic to achieve this specific objective. Consequently MaPHyS currently relies on the nested dissection approach from the graph partitioners such as Scotch [118] and METIS [90]. For a given graph $\mathcal{G} = \{V, E\}$, this method splits $V$ into two balanced subgraphs $V_1^1, V_2^1$ with a minimal size separator $V_3^1$. The same algorithm is then applied recursively on the two subgraphs $V_1^\ell, V_2^\ell, l \geqslant 1$; the outcome of this recursive algorithm is a binary tree, where the leaves correspond to the MaPHyS's $\mathcal{I}_i$ subdomains and the other nodes are the separators generated at each level of the recursion, so that $\Gamma = \cup_\ell V_3^\ell$. To compute the interfaces $\Gamma_i$, we need to extract them from the hierarchy of separators computed by the recursive nested dissection algorithm. Notice that in general, we have no control on their individual size nor imbalance between them. We however present a strategy that has been implemented in Scotch in the scope of this thesis in order to limit this drawback (see Section 2.6). Another consequence of using a nested dissection method is that the number of subdomains is a power of two.

#### 2.3.1.2 *Factorization of the interiors*

Many parallel sparse direct numerical techniques have been developed such as multifrontal approaches [61, 62], supernodal approaches [58] and fan-both algorithms [18]. Among the available direct solvers, Mumps [13], Pardiso [133] and PaStiX [81] offer a unique feature, which is the possibility to compute both the factorization of the interior matrices $\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}$ and the Schur complements using efficient sparse calculation techniques:

$$\mathcal{S}_i = \mathcal{A}^w_{\Gamma_i\Gamma_i} - \mathcal{A}_{\Gamma_i\mathcal{I}_i}\mathcal{A}^{-1}_{\mathcal{I}_i\mathcal{I}_i}\mathcal{A}_{\mathcal{I}_i\Gamma_i} \tag{2.10}$$

Because of the close research interactions with these sparse packages and to illustrate the flexibility of the implementation, MaPHyS uses the two solvers MUltifrontal Massively Parallel sparse direct Solver (Mumps) and Parallel Sparse matriX package (PaStiX). The computation of the local Schur complement benefits from the general overall efficiency of the sparse direct solver. Basically the main feature for the Schur computation, from the algorithmic point of view, can be expressed as a partial right looking factorization where instead of doing the factorization of the entire matrix $\mathcal{A}_i$, the factorization associated with the indices of $\mathcal{A}^w_{\Gamma_i\Gamma_i}$ is disabled.

From a software point of view, the user must specify the list of indices associated with the nodes on $\mathcal{A}_{\Gamma_i\Gamma_i}$. The code then provides a factorization of the $\mathcal{A}_{\Gamma_i\Gamma_i}$ matrix and the explicit Schur complement matrix $\mathcal{S}_i$, that is returned as a dense matrix. The partial factorization that builds the Schur complement matrix is used to solve linear systems associated with the matrix $\mathcal{A}_{\Gamma_i\Gamma_i}$.

#### 2.3.1.3 *Setup of the preconditioner*

The *setup of the preconditioner* mainly consists of two phases: the assembly and the factorization of the assembled local Schur complement $\bar{\mathcal{S}}_i$ concurrently on each process.

**Assembly phase.** During the assembly phase, each process communicates with its neighbours constructing the local $\bar{\mathcal{S}}_i$. This step only requires a few point-to-point communications between neighbours and is briefly described by Algorithm 3.

**Algorithm 3** Assembling the local Schur complement

1: $\bar{\mathcal{S}}_i \leftarrow \mathcal{S}_i$
2: **for** $k = 1, nb_neighbour$ **do**
3: Buffered SEND part of $\mathcal{S}_i$ to neighbour k
4: **end for**
5: **for** $k = 1, nb_neighbour$ **do**
6: Receive RECV part of $\mathcal{S}_i$ from neighbour k: $buffer_{temp} \leftarrow$RECV()
7: Update $\bar{\mathcal{S}}_i \leftarrow \bar{\mathcal{S}}_i + buffer_{temp}$
8: **end for**

**Factorization of the Schur.** The next phase consists of the factorization of $\bar{\mathcal{S}}_i$ that is a dense matrix. Its factorization (as well as the backward/forward substitution at each iteration of the Krylov solver) might be computationally expensive. A possible alternative to get a cheaper preconditioner is to consider a sparse approximation for $\bar{\mathcal{S}}_i$ in (2.4), which results in a saving of memory to store the preconditioner and saving of computation to factorize and apply it. This approximation $\widetilde{\bar{\mathcal{S}}}_i$ can be constructed by dropping the elements of $\bar{\mathcal{S}}_i$ that are smaller than a given threshold. More precisely, the following dropping strategy, that preserves the symmetry for symmetric problems, can be applied:

$$\widetilde{\bar{\mathcal{S}}}_{\ell j} = \left\{ \begin{array}{ll} 0 & \text{if } \; |\bar{s}_{\ell j}| \leq \theta(|\bar{s}_{\ell\ell}| + |\bar{s}_{jj}|) \\ \bar{s}_{\ell j} & \text{otherwise} \end{array} \right. \tag{2.11}$$

where $\bar{s}_{\ell j}$ denotes the entries of $\bar{\mathcal{S}}_i$. The resulting preconditioner based on these sparse approximations reads

$$\mathcal{M}_{AS_{sp}} = \sum_{i=1}^{\mathcal{N}} \mathcal{R}_{\Gamma_i}^T \widetilde{\bar{\mathcal{S}}}_i^{-1} \mathcal{R}_{\Gamma_i}. \tag{2.12}$$

In equation 2.4, we will refer to a dense preconditioner ($\mathcal{M}_{AS_d}$), if no dropping is performed during the assembly phase on the local assembled Schur complement. The sparse preconditioner ($\mathcal{M}_{AS_{sp}}$), given in Equation (2.12), corresponds to a preconditioner where dropping is performed on the local assembled Schur complement. The dense preconditioner consists of assembling the local Schur complement on each process computed by the direct sparse solver, and then factorizing them concurrently. MaPHyS relies on the LAPACK standard and can use any library implementing this standard. In the work proposed in the rest of this chapter, we use the Intel Math Kernel Library (Mkl) for that purpose. For the sparse preconditioner, it consists in assembling the local Schur complement, sparsifying and factorizing the local assembled Schur complement ($\bar{\mathcal{S}}_i$) using the Mumps or PaStiX sparse direct solvers.

#### 2.3.1.4 *Solve step*

Two main phases are implemented by the *solve step*: the iterative solution of the Schur complement system defined on the interface unknowns (1) and the back-solve on the interiors (2).

**d1) Iterative solution.** The efficient implementation of a Krylov iterative method depends on two factors: optimized implementation of computational kernels and stopping criterion for convergence detection. In these methods three major computational kernel are crucial: matrix-vector product, the application of the preconditioner to a vector and the dot product kernel. Below our implementation of these kernels is explained followed by the definition of the stopping criterion used in our study.

**Matrix-vector product: $y = \mathcal{S}x$.** It can be performed in two ways, explicitly using a BLAS-2 routine or implicitly using sparse matrix-vector calculations. Both versions perform only local point-to-point communications. The explicit computation is described by Algorithm 4, whereas the implicit one is given by Algorithm 5.

**Algorithm 4** Explicit matrix-vector product

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

**Algorithm 5** Implicit matrix-vector product

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

**Application of the preconditioner:** $y[i] \leftarrow \mathcal{M}_i^{-1} x_i$ where $\mathcal{M}_i$ is equal to $\widetilde{\bar{\mathcal{S}}}_i$ or $\bar{\mathcal{S}}_i$ depending on the selected (sparse or dense) preconditioner.

Because the preconditioners have a form similar to the Schur complement, its parallel application to a vector is implemented similarly. This step described in Algorithm 6 can be performed using either MKL kernels in the dense preconditioner case or MUMPS or PASTIX sparse direct solver in the sparse preconditioner case.

**Algorithm 6** Application of the preconditioner

1: In parallel each process performs the triangular solve $y_i \leftarrow \mathcal{M}_i^{-1} x_i$
2: Update data: exchanges of information between the neighbouring subdomains
   Each process assembles $y \leftarrow \sum_{i=1}^{nb_neighbour} \mathcal{R}_{\Gamma[i]} y_i$,

**The dot product:** $y_i = y_i^T x_i$. The dot product calculation is simply a local dot-product computed by each process followed by a global reduction to assemble the complete result as described in Algorithm 7.

**Algorithm 7** Parallel dot product

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

**Stopping criterion:** The *normwise backward error* presented in Section 1.4.4 is used for our study. As the iterative method is applied on the reduced system introduced in Equation 2.2, one option consists in using scaled residual norm associated to that system:

$$\varepsilon_f = \frac{\|\mathcal{S}x_\Gamma - f\|}{\|f\|} \tag{2.13}$$

The results presented in Section 2.4, Section 2.5 and Section 2.6 are based on this stopping criterion.

However, this criterion is related to the reduced system but not necessarily to the global system. Indeed, assuming the backsolve on the interior is performed in exact arithmetic, we have: $\frac{\|\mathcal{S}x_\Gamma - f\|}{\|f\|} = \frac{\|\mathcal{A}x - b\|}{\|f\|}$. Because $\|f\|$ depends on the number of subdomains, this criterion may thus not be appropriate for comparisons between executions whose number of subdomains differs. This is important in particular when we compare MAPHYS and PDSLIN (Section 2.7) whose respective optimum tuning often correspond to different numbers of subdomains. In order to be on the same foot of equality in that case, we rely on the alternative stopping criterion:

$$\varepsilon_b = \frac{\|\mathcal{S}x_\Gamma - f\|}{\|b\|} \tag{2.14}$$

This latter stopping criterion indeed reflects the error on the global system because, assuming the backsolve on the interior is performed in exact arithmetic, we have: $\frac{\|\mathcal{S}x_\Gamma - f\|}{\|b\|} = \frac{\|\mathcal{A}x - b\|}{\|b\|}$.

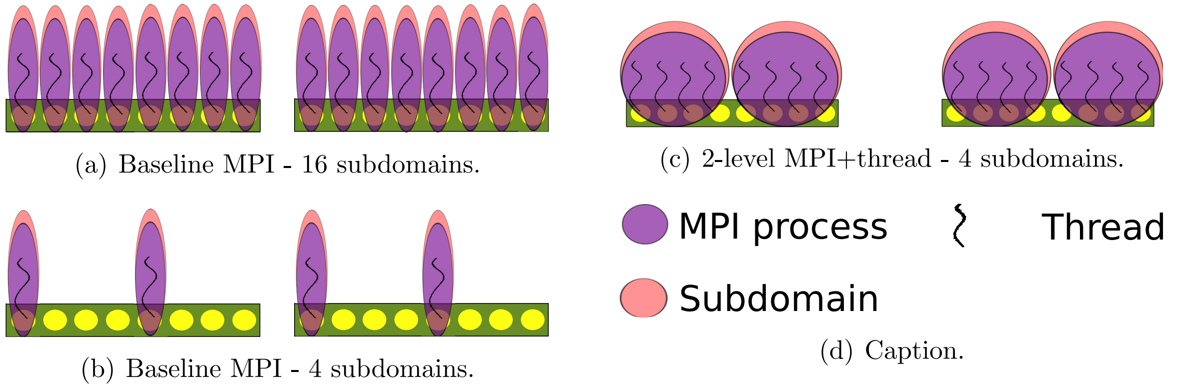
**d2) Back-solve on the interiors** Once $x_{\Gamma_i}$ is obtained with the above iterative solution step, $x_{\Gamma_i}$ can be calculated with a back-solve. This is done by calling the solve step of the sparse direct solver to compute $x_{\mathcal{I}_i} = \mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}^{-1}(b_{\mathcal{I}_i} - \mathcal{A}_{\mathcal{I}_i\Gamma_i}x_{\Gamma_i})$.

### 2.3.2 Design of a 2-level MPI+thread extension of MaPHyS

The baseline MPI model presented above presents some strong limitations on modern multicore supercomputers. Indeed, modern platforms tend to have an increased number of cores. Yet, the baseline model assigns exactly one subdomain per core being used. As a consequence, if one wants to exploit all available computational cores, he has to decompose the matrix in as many subdomains. For instance, if two nodes of eight cores each are being used, 16 subdomains have to be used in order to exploit all the 16 cores (see Figure 2.3(a)). As shown later on this thesis (Section 2.4), this approach can:

- deteriorate or even prevent the convergence of the method at scale;
- leads to excessive memory consumption, possibly leading the method to run out-of-memory.

Solving those issues with the baseline MPI MaPHyS implementation would impose to assign fewer subdomains per node, hence fewer cores. For instance, on the same example platform, imposing four subdomains would limit us to use four cores only (see Figure 2.3(b)). To overcome these limitations while using all available cores, we have designed a 2-level




(a) Baseline MPI - 16 subdomains.

(b) Baseline MPI - 4 subdomains.

(c) 2-level MPI+thread - 4 subdomains.

(d) Caption.

Figure 2.3: Baseline MPI (a, b) versus MPI+thread (c) models on a 16 cores platforms composed of two nodes of eight cores each. Imposing four subdomains limits the baseline MPI model to exploit four cores (b) but not the MPI+thread model (c).

MPI+thread method. With such a method, each process associated to a subdomain can use multiple cores thanks to multithreading. In our example, each subdomain would thus be processed by four cores using four threads per process (see Figure 2.3(c)).

In this section we present the design of our MPI+thread extension of MaPHyS. Because MaPHyS is modular, we first need to select and possibly tune multithreaded versions of the external software packages MaPHyS relies on (Section 2.3.2.1). We then need to make

sure that those libraries cooperate correctly in order to maximize the overall performance of our hybrid solver (Section 2.3.2.2). Among other important issues, the physical placement of the threads on the processing units is critical. We present the three different binding strategies that have been studied in Section 2.3.2.3.

#### 2.3.2.1 Multithreaded building block operations

We focus in this section on the description of multithreaded version of the external software packages that are used by MaPHyS. The 2-level parallel implementation will be effective for our hybrid solver if its three main computational phases can be efficiently performed in parallel: *factorization of the interiors*, *setup of the preconditioner* and *solve step*. We do not consider the parallel implementation of the *partitioning step* for two reasons. First, with the current state of MaPHyS, the domain decomposition is performed sequentially in a pre-processing phase, which performs the decomposition and sends the needed data on the corresponding processes. Furthermore, the current software version of the partitioners Scotch and METIS are not multithreaded. Let us quickly recall the main numerical kernels of the three computational steps of our solver and for each of them describe the multithreaded strategy.

For the *factorization of the interiors* with the current state of our package we are able to use two sparse direct solvers, PaStiX and Mumps. These packages are able to perform Cholesky, $LU$ or $LDL^T$ factorizations of symmetric definite, symmetric and general sparse matrices respectively. Both solvers are state of the art software, implemented with a large number of numerical and technical functionalities. The major difference between the two solvers is that Mumps is based on the multifrontal approach while PaStiX is a supernodal solver. With the recent development of the Mumps group, activities towards a multithreaded version have been accomplished [97], but when this study was developed, only the PaStiX package offered the possibility of using multithreading. In this study only PaStiX is considered and a more detailed description is given below. Two different versions of the *setup of the preconditioner* are possible in our solver, sparse and dense. For each one, a different library is used. For the sparse version, the PaStiX solver is used, while for the dense version, the LAPACK routines of the Mkl libraries are used. For the iterative solver, the BLAS operation are provided by the Mkl library and the application of the preconditioner is performed by the Mkl library or by the back-solve operation of the sparse direct solver that is used. Finally the back-solve on the interiors is performed by the sparse direct solver. All in all, we use the PaStiX sparse direct solver and the Mkl library.

**Multithreaded** Mkl The Mkl library provides highly optimized BLAS and LAPACK routines for Intel processors. The multithreaded version of the Mkl library is implemented on top of the Intel OpenMP runtime system.

**Multithreaded sparse direct solver** Sparse direct solvers usually consist of three distinct steps. First, the analysis step performs a reordering of the variables (by calling a third-party library such as Scotch or METIS) and a symbolic factorization in order to

compute data structures that cope with expected fill-in. Second, the matrix is decomposed in factors in the so-called numerical factorization or factorization step for short. In MaPHyS these first two steps occur in sequence during the *factorization of the interiors* (and separately during the computation of a sparse preconditioner). Third, the solve step computes the solution from the right-hand side and the factors. This third step occurs during the *back-solve of the interiors* (and separately during the application of the preconditioner if a spare preconditioner is used).

PaStiX is based on the supernodal technique. Once the symbolic factorization has been performed, the elimination tree is available. Basically the supernodal method consists of regrouping several nodes (columns) of the elimination tree in a larger set called "supernode". From the matrix point of view, the supernodal approach can be viewed as grouping several contiguous columns in a single column-block. Instead of using inefficient kernels with low fetch/compute ratio, well suited BLAS-3 operations can then be used and exploit much more efficiently modern processing units. The supernodal factorization occuring in PaStiX consists of performing three operations for each column-block of the matrix, as depicted in Figure 2.4 for the first column-block. First, the factorization of the diagonal block is computed (red block in Figure 2.4). Then a triangular solve is performed on the off-diagonal part of the column-block (yellow blocks in Figure 2.4). For each off-diagonal sub-block of the factorized column-block, an update to the facing column-block that owns the same rows is applied (green blocks in Figure 2.4). This is repeated for each column-block until the whole matrix is factorized. In PaStiX each operation is performed by one thread and the parallelism is expressed by performing several independent matrix operations simultaneously.

The multithreaded strategy of PaStiX was implemented for the factorization of $\mathcal{A}_{\Gamma_i\Gamma_i}$ but not for the calculation of the Schur ($\mathcal{S}_i$) that was performed sequentially. For the purpose of this thesis, we relieved this bottleneck by enabling concurrent updates of $\mathcal{S}_i$ as long as the updates are performed on disjoint matrix blocks, the 3 green blocks in the red bottom-right block in Figure 2.4). In that respect, we considered a 2D block decomposition of the square dense Schur complement matrix. The concurrency is protected by a simple deadlock-free algorithm consisting of performing active waiting until all the blocks accessed by an update are free for being updated. This additional software development designed to remove the final bottleneck in the Schur complement capability of PaStiX has been eventually integrated in the public domain distribution of PaStiX.

#### 2.3.2.2 Coexistence of different multithreaded libraries

The main issue of interoperability between PaStiX and Mkl is related to the different ways these libraries manage the parallelism. As explained above, PaStiX expresses the parallelism through individual tasks associated with different supernodes; those tasks are handled by single threads. The supernode calculation is performed by sequential (mono-threaded) BLAS-3 routines from the Mkl library. However, the multithreading exploited in the other steps of MaPHyS such as the *setup of the preconditioner* and the Krylov iterations is based on calls to multithreaded Mkl routines. Consequently, both mono and multithreaded Mkl subroutines need to be used in different contexts by MaPHyS when PaStiX or Mkl library is used respectively.

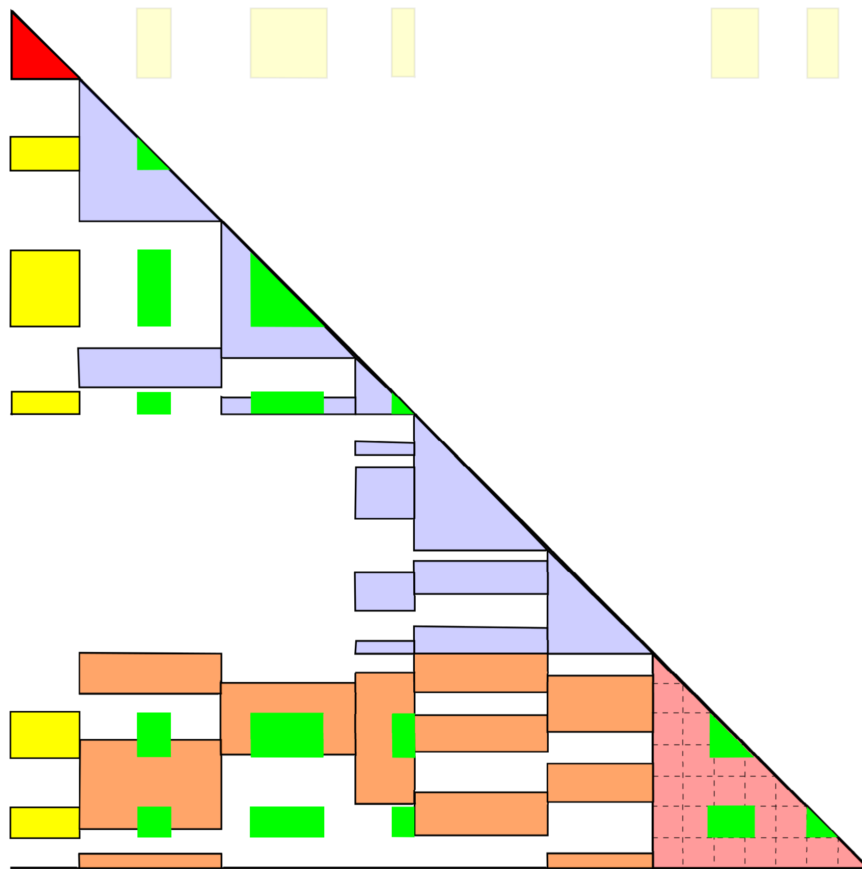


Figure 2.4: Operations related to the factorization of the first supernode (red block): triangular solve on the off-diagonal column-block (yellow blocks) and update to the facing column-blocks that owns the same rows (green blocks).

Because the Mkl library relies on OpenMP, there are two different ways to control multithreading: through the OpenMP functionalities or directly using functions defined in the Mkl library such as *mkl_set_num_threads*(*nb_threads*) or *omp_set_num_threads*(*nb_threads*). The Mkl functions override the functions of OpenMP. We point out below the adaptation that needs to be performed for the main three steps of MaPHyS.

First, during the *factorization of the interiors*PaStiX is used. So, during this step, we are using the mono-threaded version of Mkl. For the *setup of the preconditioner*, depending on wich version od the preconditioner is used, the dense one or the sparse one, multithreade Mkl or PaStiX on top of a mono-threaded Mkl are used respectively. In a similar manner, during the iterative method, Mkl or PaStiX is used for applying the preconditioner depending on the version of the preconditioner. For the rest of the routines within the iterative part, the multithreaded version of Mkl is used. Finaly, for the back-solve on the interiors, PaStiX is used with the mono-threaded Mkl.

#### 2.3.2.3 Binding strategies

Three binding variants have been initially explored. The first strategy simply consists of not enabling any binding and let the operating system handle the placement of processes and threads (Figure 2.5(a)). The second strategy consists of binding processes to a subset of cores and let the operating system handle the placement of threads (Figure 2.5(b)). The third strategy consists of binding both processes and threads within processes (Figure 2.5(c)).

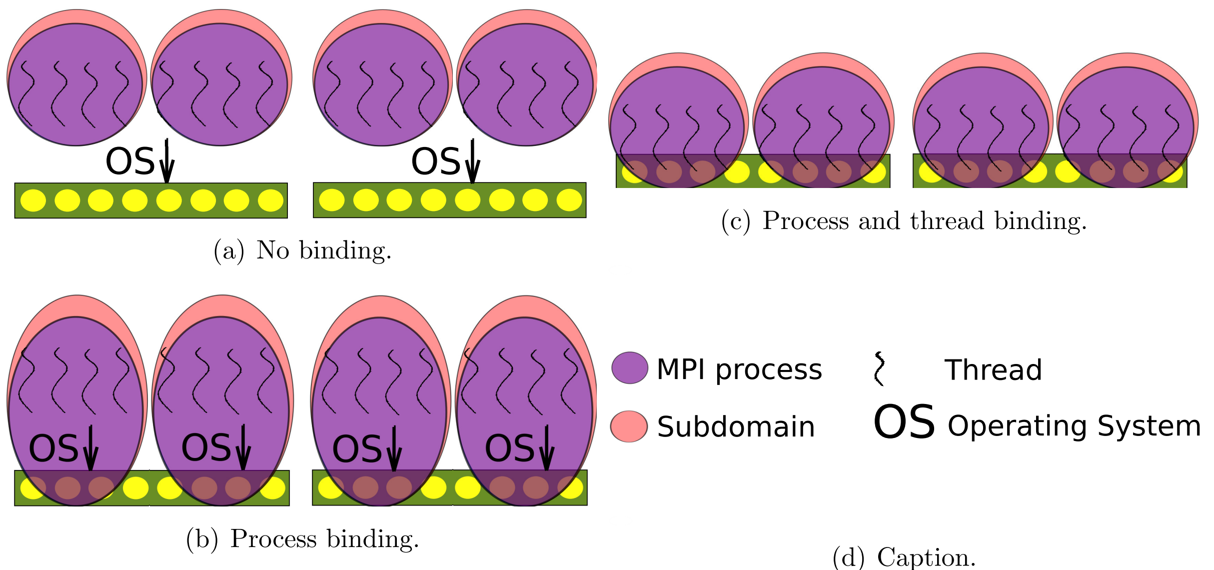



Figure 2.5: Binding strategies.

The mechanism for binding processes and threads may differ depending on the considered target operating system and architecture. To perform a consistent and portable binding, we rely on the Portable Hardware Locality (hwloc) software package [38]. The hwloc library

indeed provides a portable interface that is an abstraction of all the low level technical details (operating system and architecture). It also offers a consistent numbering of the physical cores, ensuring that cores that share cache memory will be numbered next to each other. This is not always the case with the raw information provided by the operating system. The numbering of hwloc is thus better suited for designing portable binding algorithms.

Modern multithreaded BLAS libraries commonly perform placement of processes and threads. It is for instance the case of the MKL library on which we rely in this thesis. In order to control the binding by our own, we therefore prevent MKL from doing it. This is achieved by setting the environment variable $KMP_AFFINITY$ to "disable".

We further discuss the impact of binding on performance in Section 2.4.2 where we show that the second strategy consisting of binding processes to a subset of cores and let the operating system handle the placement of threads (Figure 2.5(b)) is consistently optimum. For this reason, this strategy is used in the rest of this chapter.

## 2.4 Performance analysis

In this section, a large set of experiments is presented to analyze the parallel and numerical performance of the proposed MPI+thread extension of MaPHyS. After selecting the most appropriate binding strategy (Section 2.4.3), we study the impact of multithreading on performance (Section 2.4.3). In that context, the number of subdomains is set to the number of nodes and we only vary the number of threads per subdomain. We then study the flexibility of our 2-level parallel implementation to achieve the best trade-off between the numerical and parallel behaviors (Section 2.4.4). For those experiments, we vary the number of subdomains and the number of threads per subdomain, so that the total number of cores remains constant.

### 2.4.1 Experimental setup

| Matrix | `Matrix211` | `Audi_kw` | `Nachos` | `Haltere` | `Tdr455k` | `Nachos4M` | `Amande` |
|---|---|---|---|---|---|---|---|
| $n$ | 0.8M | 0.9M | 1.1M | 1.3M | 2.7M | 4.1M | 7.0M |
| $nnz$ | 129M | 392M | 40M | 10M | 113M | 256M | 58M |
| Symmetry | non-symmetric | SPD | non-symmetric | symmetric | non-symmetric | non-symmetric | symmetric |
| Arithmetic | real | real | complex | complex | complex | complex | complex |

Table 2.1: Our main matrix collection used in this chapter.

The experiments presented in this chapter were conducted on two platforms: `PlaFRIM`, located at Inria Bordeaux-Sud-Ouest and the `Hopper` machine from NERSC, located at the Lawrence Berkeley National Laboratory. The `PlaFRIM` platform is composed of 68 nodes. Each node is equipped with two Quad-core Nehalem Intel Xeon X5550 with a total of eight cores per node. Each node has a total of 24Gb of RAM memory. The nodes are

interconnected in a fat tree fashion using an Infiniband QDR network delivering a 40Gb/s bandwidth. The `Hopper` platform is composed by 6384 nodes, each being two twelve-core AMD magny-cours 2.1-GHz processors. Each node has 24Gb of RAM. `Hopper`'s compute nodes are connected via a custom high-bandwidth, low-latency network provided by Cray.

We display in Table 2.1 the main characteristics of the test matrices considered in this section. They arise from different application domains ranging from structural mechanic analysis to electromagnetics calculation using both classical and discontinuous finite element modeling.

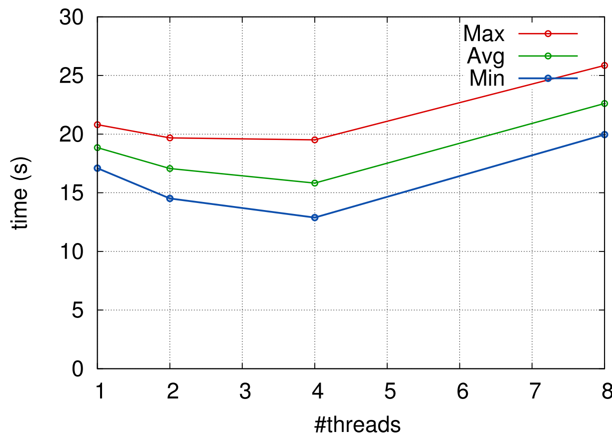
We use the convergence criterion proposed in Equation 2.13 which we set to $\varepsilon_f = 10^{-10}$ when no stated otherwise. The right preconditioned GMRES restart parameter is set to 500 and the maximum number of iterations is set to 7000. We use the version 4.0.1 of METIS to perform the domain decompositions and rely on the MKL library version 11.1.3 and the modified PASTIX version 5.2 as discussed in Section 2.3.2.

### 2.4.2 Impact of the thread binding strategies

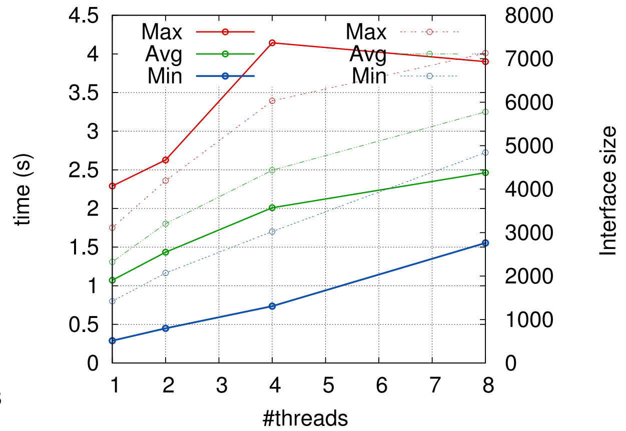
As discussed in Section 2.3.2, three binding strategies have been designed consisting either of fully delegating the placement management to the operating system (Figure 2.5(a)), binding processes to a subset of cores and let the operating system handle the placement of threads (Figure 2.5(b)) or binding both processes and threads within processes (Figure 2.5(c)), respectively. Intensive experiments (not reported here) have been conducted on all matrices from Table 2.1 showing that the second strategy consisting of binding processes to a subset of cores and allowing the operating system handle the placement of threads (Figure 2.5(b)) is consistently optimum.

Indeed, this configuration outperforms the case where threads are also binded by the application for two reasons. First, binding threads at the application level prevents MKL from doing internal optimizations related to multithreading. Second, every time the number of threads used by the MKL library is changed, the binding needs to be performed again. This effects turns out to be especially significant in the iterative method when the sparse preconditioner is used, because it has to be done twice at each iteration for switching between calls to the dense operations performed by MKL and the call to the sparse direct solver for applying the preconditioner.

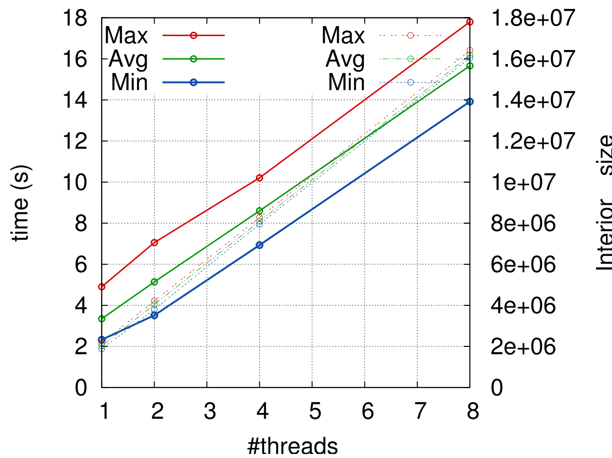
It also outperforms the case where the placement is fully delegated to the operating system. We report here a detailed analysis on a particular example, considering matrix `Matrix211` processed on 8 nodes of `PlaFRIM` with a dense preconditioner. We compare the behavior of MAPHYS when no binding is performed (Figure 2.6) to the case where processes are binded (Figure 2.7). When only one process per node is used (eight threads per domain), both versions are equivalent and therefore achieve the same performance (right-most part of the plots). However, when multiple processes compete on the same node (*i.e.* when one, two or four threads per domain are used), the performance gap may be significant, especially because of the *solve step*.



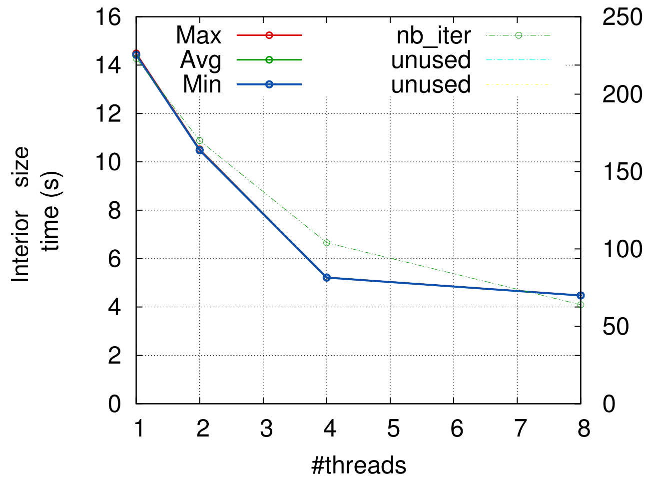

(a) *All computantional steps.*




(c) *Setup of the preconditioner.*

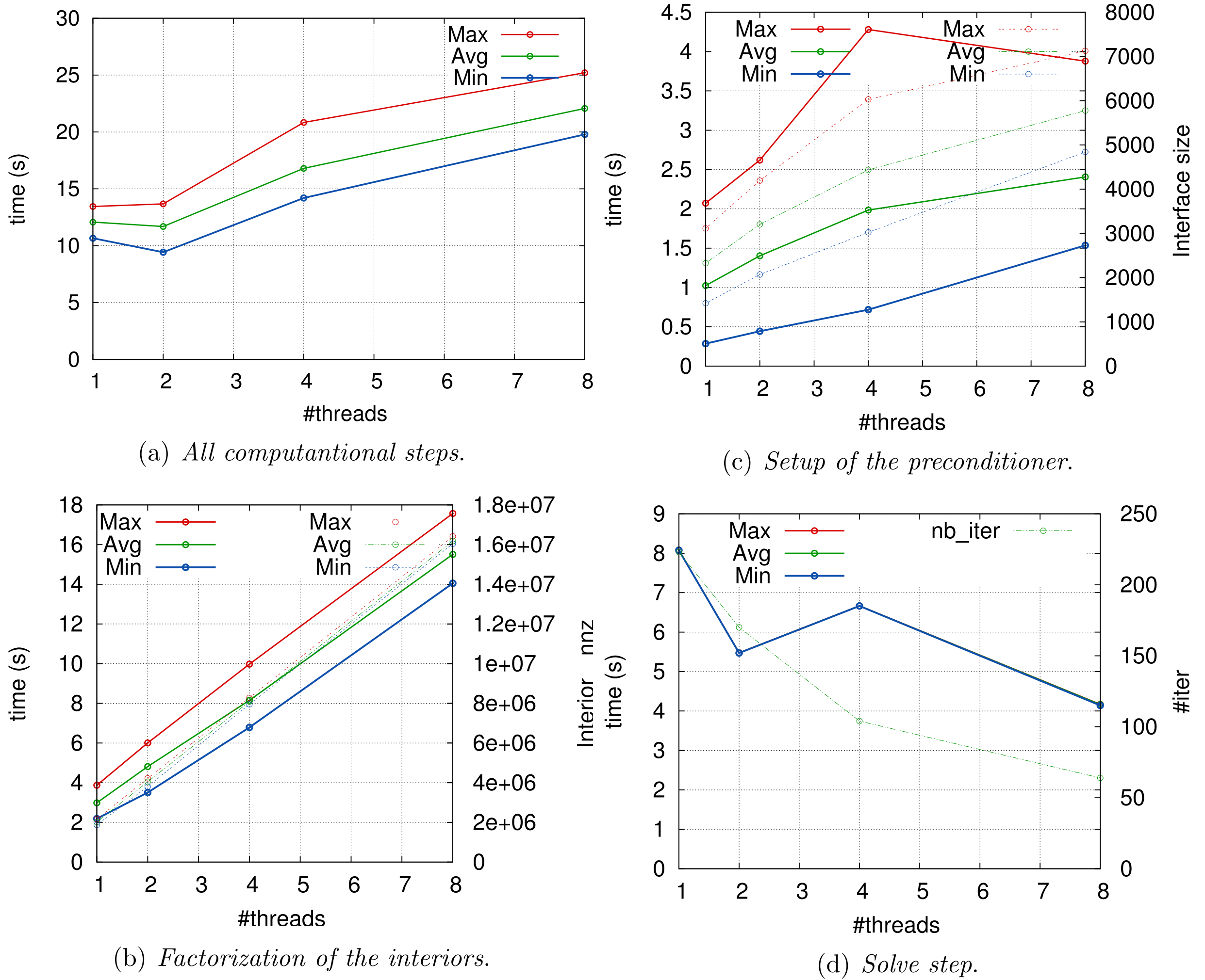



(b) *Factorization of the interiors.*




(d) *Solve step.*

Figure 2.6: The maximum, average and minimum elapsed time per subdomain of MaPHyS when binding management is fully delegated to the operating system (configuration of Figure 2.5(a)) for matrix `Matrix211` on eight nodes of `PlaFRIM` with dense preconditioner. On each node, all eight cores are consistently used but the number of threads per subdomain (1, 2, 4 or 8, x-axis) is inversely proportional to the number of subdomains (8, 4, 2, 1).




(a) *All computantional steps.*

(c) *Setup of the preconditioner.*

(b) *Factorization of the interiors.*

(d) *Solve step.*

Figure 2.7: The maximum, average and minimum elapsed time per subdomain of MaPHyS when binding processes are binded to a subset of cores (configuration of Figure 2.5(b)) for matrix `Matrix211` on 8 nodes of `PlaFRIM` with dense preconditioner. On each node, all eight cores are consistently used but the number of threads per subdomain (1, 2, 4 or 8, x-axis) is inversely proportional to the number of subdomains (8, 4, 2, 1).

### 2.4.3 Multithreading performance

In this section, we investigate the performance of the multithreaded implementation on multicore nodes. First, we are interested in the performance behavior of MaPHyS for the cases described in Table 2.2. For those experiments, the number of subdomains in only four and we use the dense variant of the preconditioner. Processes (hence subdomains) have dedicated nodes. When the number of threads increases, more and more cores can thus be exploited (see Figure 2.8). The parallel performance, measured as a speed-up with respect to the single threaded reference is plotted in Figure 2.9 using bar charts with the corresponding thread number on the top of the bars. For each matrix, we vary the number of threads from one to eight and compute the speed-ups for the three numerical steps of MaPHyS (steps 2.3.1.2, 2.3.1.3 and 2.3.1.4 in Section 2.3.1) as well as the overall speed-up (combination of steps 2.3.1.2, 2.3.1.3 and 2.3.1.4 ).

| Matrix | `Matrix211` | `Audi_kw` | `Nachos` | `Haltere` | `Tdr455k` | `Amande` |
|---|---|---|---|---|---|---|
| Interior's size (avg) | 197K | 233K | 275K | 320K | 684K | 1742K |
| Local Schur's size (avg) | 7K | 6K | 10K | 4K | 2K | 12K |
| Schur's size | 14K | 39K | 11K | 7K | 4K | 24K |
| #iter | 36 | 24 | 28 | 9 | 5 | 16 |

Table 2.2: Sizes of the interiors ($\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}$), size of the local Schur complement ($\mathcal{S}_i$) and total size of the Schur complement ($\mathcal{S}$) are given for each matrix when four subdomains are used. The number of iterations (#iter) needed with four subdomains and dense preconditioner is also given.

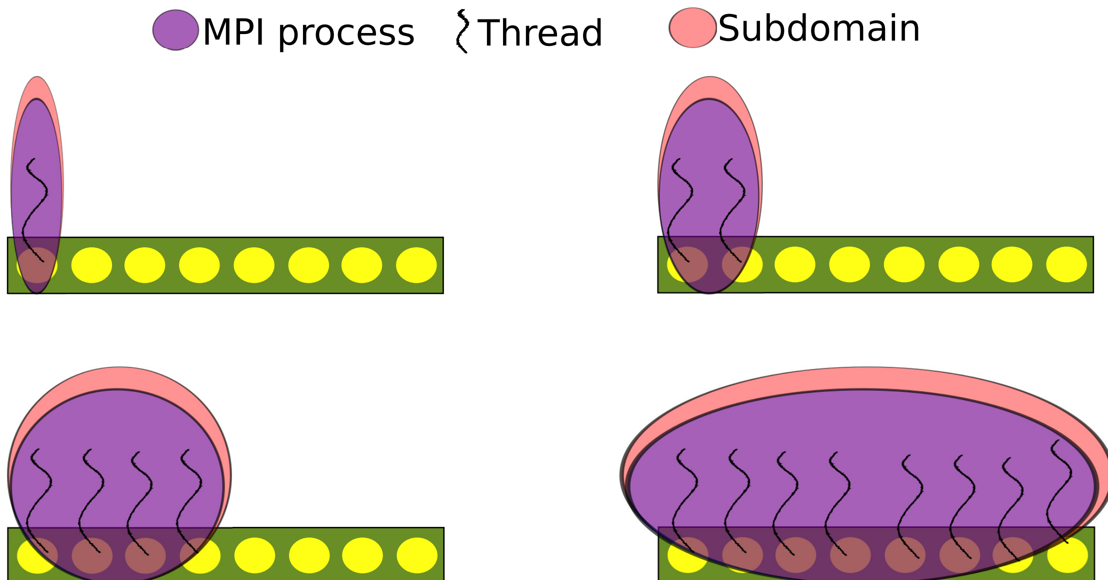
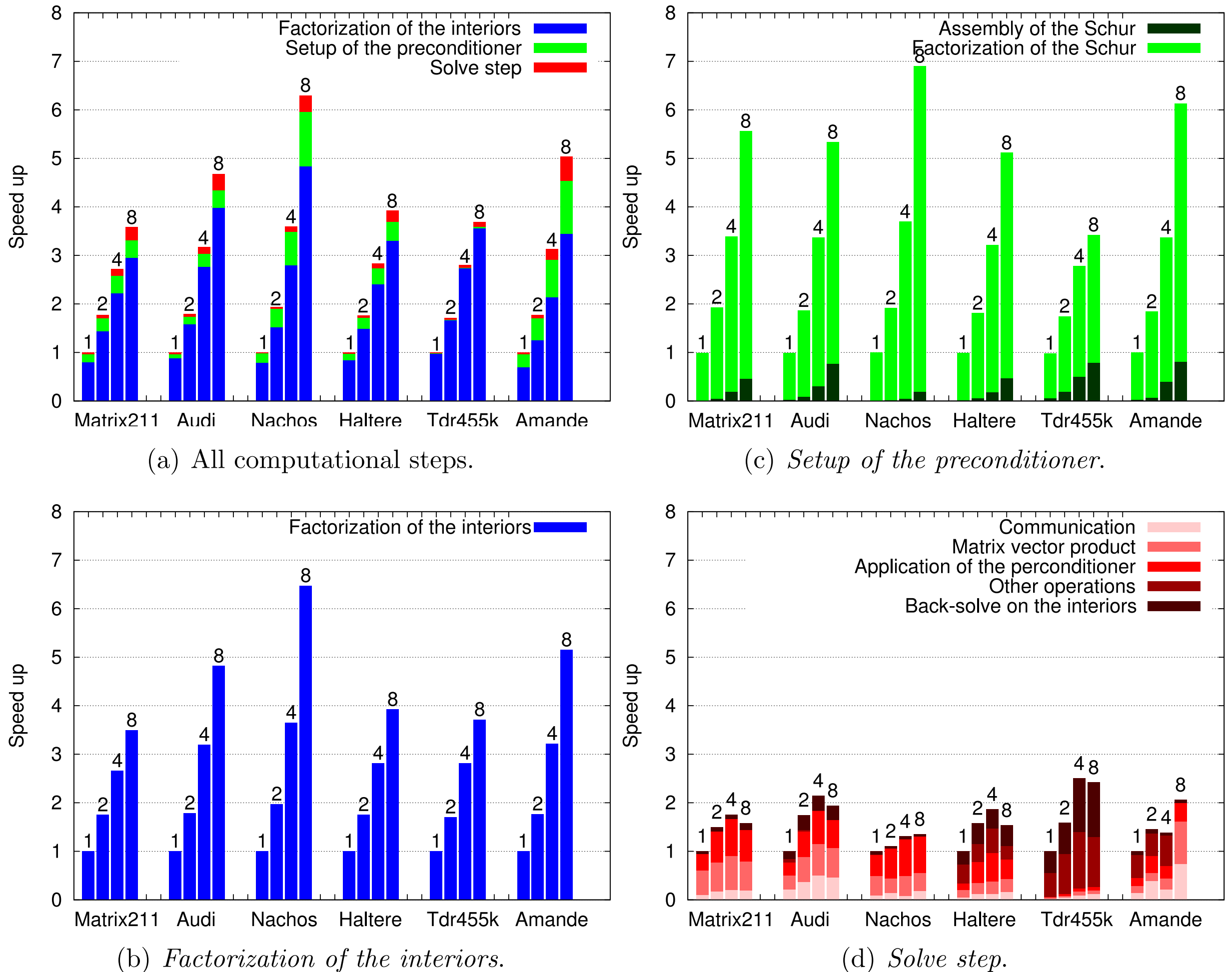



Figure 2.8: Using 1, 2, 4 or 8 threads and cores per process (set up in Section 2.4.3).

It can be seen that significant speed-ups are achieved for all test cases. An average speed-up of 4.53 is achieved for the overall execution time of our matrix collection. In

this configuration, the overall execution time is consistently dominated by the *factorization of the interiors*. An explanation for this is the fact that only four subdomains are used. Most of the unknowns are associated with the interior resulting to relatively small Schur complements. As a consequence, the *factorization of the interiors* takes more time to be performed compare to the *setup of the preconditioner*. Using only four subdomains and a dense preconditioner results with a preconditioner with good numerical quality, thus only a small number of iterations is needed in order to achieve the target accuracy.




(a) All computational steps.

(c) *Setup of the preconditioner.*

(b) *Factorization of the interiors.*

(d) *Solve step.*

Figure 2.9: Overall speed-up with using 1, 2, 4 or 8 threads (and cores) with respect to the time when 1 thread (and core) per process, for the matrices from Table 2.1. The dense preconditioner is used. Four nodes are used and one process per node is assigned. The number of threads (1, 2, 4 or 8) for each histogram is given on the top of the bars.

Since the overall time is dominated by the *factorization of the interiors*, it has a behavior similar to the overall speed-up. An average speed-up of 4.6 is obtained for the *factorization of the interiors* (Figure 2.9(b)) when 8 threads are used. The size of the interiors is large enough (Table 2.2) for the multithreaded direct sparse solver. During the *setup of the preconditioner* with this configuration, the dense solver is used for the factorization of $\bar{\mathcal{S}}_i$. The size of the local Schur complements is large enough (at least a few thousands unknowns). As a result, the direct dense solver is able to exploit efficiently the multicore machines

(Figure 2.9(c)). For the *solve step*, an average speed up of only 1.94 (out of 8) is achieved (Figure 2.9(d)). Two main reasons explain this behavior. First, the iterative method is expensive in terms of communication, which do not scale with the multithreaded version. Additionally, the operations that are done during the iterative method have a low level of computational intensity. These operations do not exploit efficiently the multicore machines. In addition to what has been presented, for the `Haltere` and `Tdr455k` matrices the back-solve on the interior represents a large portion of the *solve step*. This is a consequence of the fact that only 9 and 5 iterations are needed for the `Haltere` and `Tdr455k` matrix respectively (Table 2.2). When the number of iterations increases, the portion that corresponds to the back-solve decreases. When more than a few dozen of iterations are used, the back-solve becomes inconsiderable compared to the time needed for the *solve step*.

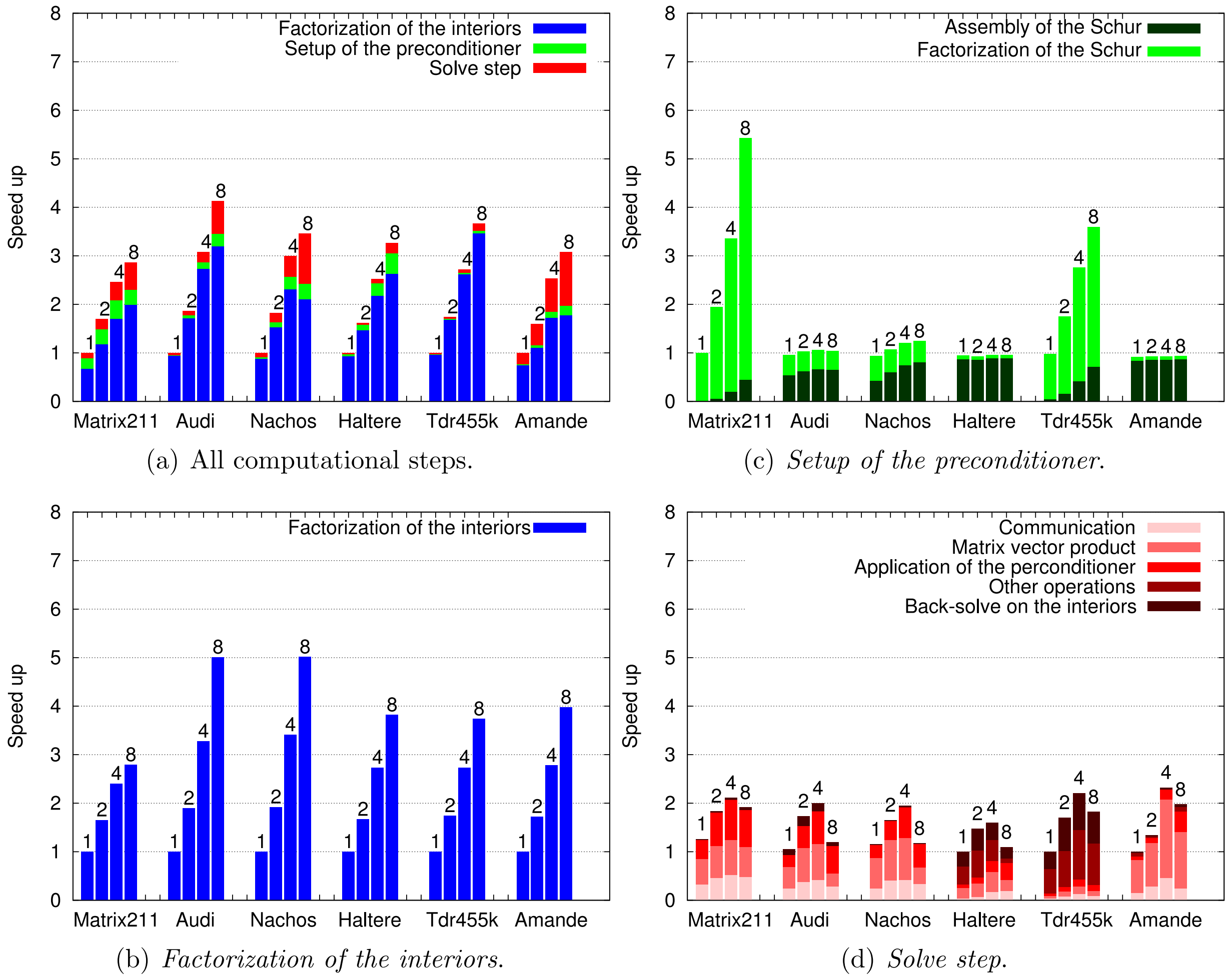
In the above study we have shown that MaPHyS is able to efficiently exploit the multicore platforms with a dense preconditioner and four nodes. To further explore the capabilities of the 2-level parallel implementation we consider experiments at a scale related to the problem size and relying on a preconditioning strategy tuned for each matrix.The characteristics of this second set of experiments are reported in Table 2.3 while the observed speed-ups are displayed in Figure 2.10.

| Matrix | | `Matrix211` | `Audi_kw` | `Nachos` | `Haltere` | `Tdr455k` | `Amande` |
|---|---|---|---|---|---|---|---|
| #nodes | | 8 | 4 | 16 | 4 | 8 | 32 |
| Preconditioner | | dense | $10^{-3}$ | $10^{-2}$ | $10^{-3}$ | dense | $10^{-2}$ |
| Interior's size (avg) | | 97K | 233K | 67K | 320K | 341K | 216K |
| Local Schur's | size (avg) | 6K | 6K | 6K | 3K | 2K | 5K |
| | kept entries (avg) | — | 2.75% | 2.5% | 2% | — | 1% |
| Schur's size | | 23K | 11K | 49K | 8K | 10K | 85K |
| #iter | | 64 | 49 | 53 | 11 | 9 | 94 |

Table 2.3: The configuration (#nodes and preconditioner) that are studied in Section 2.4.3 and 2.4.4 are presented. The sizes of the interior's ($\mathcal{I}_i$), the sizes of the local Schur ($\mathcal{S}_i$), the total size of the Schur complement ($\mathcal{S}$) and the number of iterations are given for each matrix.

Similar to the experiments reported in Figure 2.9, we display the speed-ups using bar charts when the number of threads is varied while each subdomains is mapped to one node. We still observe significant but lower speed-ups than the ones presented previously in Figure 2.9. Indeed, although the overall time is dominated by the *factorization of the interiors*, the number of subdomains increases with the number of nodes yielding smaller interiors (Table 2.3) and resulting in a *factorization of the interiors* that is harder to parallelize efficiently with the use of multiple threads (Figure 2.10(b)).

Concerning the preconditioner step, in the case where the sparsified preconditioner is used, the involved matrices almost do not benefit from the multithreading (Figure 2.10(c)). The main reason is that sparsifing considerably decreases the amount of computation (factorization of the local assembled Schur $\bar{\mathcal{S}}_i$), which thus becomes dominated by the assembly




(a) All computational steps.

(c) *Setup of the preconditioner.*

(b) *Factorization of the interiors.*

(d) *Solve step.*

Figure 2.10: Overall speed-up when using 1, 2, 4 or 8 threads (and cores) with respect to the time when 1 thread (and core) per process for the matrices from Table 2.3. The configuration (#nodes and preconditioner) is given in Table 2.3 for each matrix and the test were performed on the `PlaFRIM` platform. The number of threads (1, 2, 4 or 8) for each histogram is given on the top of each bar.

step. Because the assembly step consists only of communications, it is not affected by multithreading. The second reason is that the local Schur complement become too small for a multithreaded sparse direct solver. For example, for the `Audi_kw` matrix, the average size of $\bar{\mathcal{S}}_i$ is 5,884. After applying the dropping with a threshold of $10^{-3}$, approximately 2.75 percent of the entries are kept, resulting in a sparsified $\bar{\mathcal{S}}_i$ that contains around 692K of non zero values. This is not a large enough computational load for the sparse direct solver to exploit efficiently the multicore processors. As a consequence, the overall *setup of the preconditioner* is hard to accelerate with multithreading when a sparse preconditioner is used. On the other hand, thanks to the sparsification, the *setup of the preconditioner* turns out to be only a very limited proportion of the total execution time (see Figure 2.10(a)). For instance, when eight threads are used on four nodes with the `Audi_kw` matrix, the dense factorization is performed in 2.55 seconds, while the sparse factorization takes only 0.95 seconds. Regarding the *solve step*, it follows the same comportment for all the matrices for the same reasons as explained above.

### 2.4.4 Numerical and parallel flexibility of the 2-level implementation

In the previous section, we mainly focused on the parallel efficiency of the 2-level implementation when the number of cores is increased while keeping constant the number of nodes. We are now investigating the capability of the code to fully exploit the computing resources of a hierarchical multicore platform. To highlight the flexibility we fix the number of cores used for the solution of a given problem and we vary the number of subdomains and the number of threads per subdomain so that $nb_nodes \times nb_cores = nb_threads_per_process \times nb_domains$ (see Figure 2.11). Those iso-computing resource experiments show that our 2-level approach enables us to find the best trade-off between the numerical behavior of the solver (which depends on the number of subdomains) and its parallel performance (which depends on the number of threads per subdomain). For those experiments we consider the same test examples as in Table 2.3.

With this configuration, when the number of threads per process increases, the number of subdomains decreases; this tends to reduce the number of iterations but enlarges the elapsed time of the *factorization of the interiors*. For instance, for the `Audi_kw` matrix, the first histogram presents a run with 32 subdomains with one thread per process, while the last one, when 8 threads are used, corresponds to a run with 4 subdomains. Compared to the results presented in the previous section, for each run not only the number of cores assigned per process changes, but also the numerical problem solved. With an increased number of subdomains, the size of the domain interior is decreasing (Figure 2.12(a)), while the size of the total Schur complement ($\mathcal{S}$) is increasing (Figure 2.12(b)). With a larger Schur complement and the larger number of subdomains, the numerical difficulty of the iterative part tends to increase, increasing the number iterations and possibly the overall solution time (Figure 2.12(d)). With our 2-level parallel approach, we are able to explore the trade-off between larger subdomains and numerical efficiency of the iterative method in order to exploit the multicore machines in a most optimal way. The results are presented in Figure 2.13.

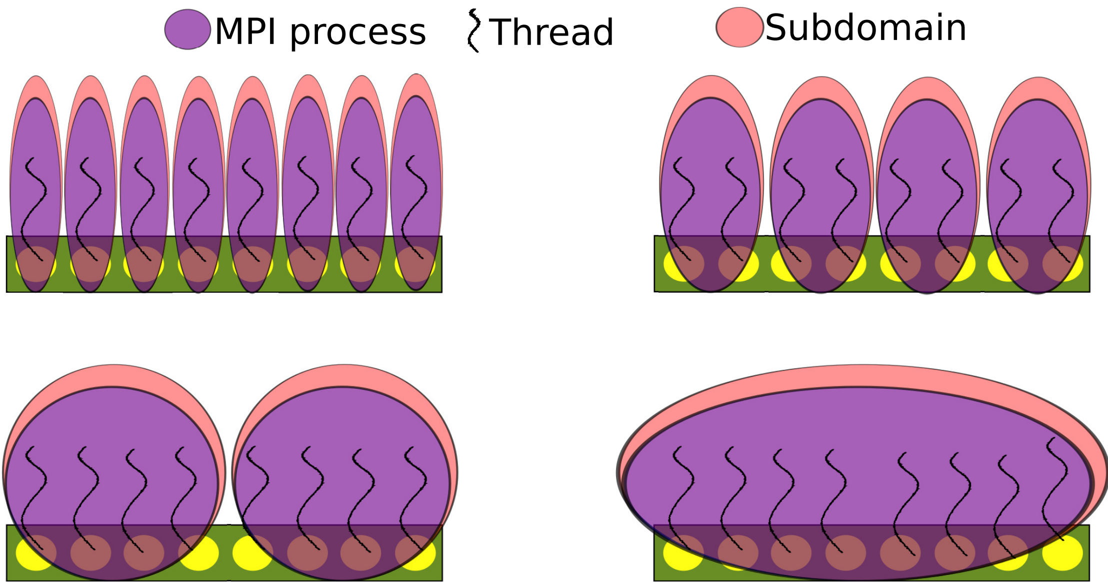
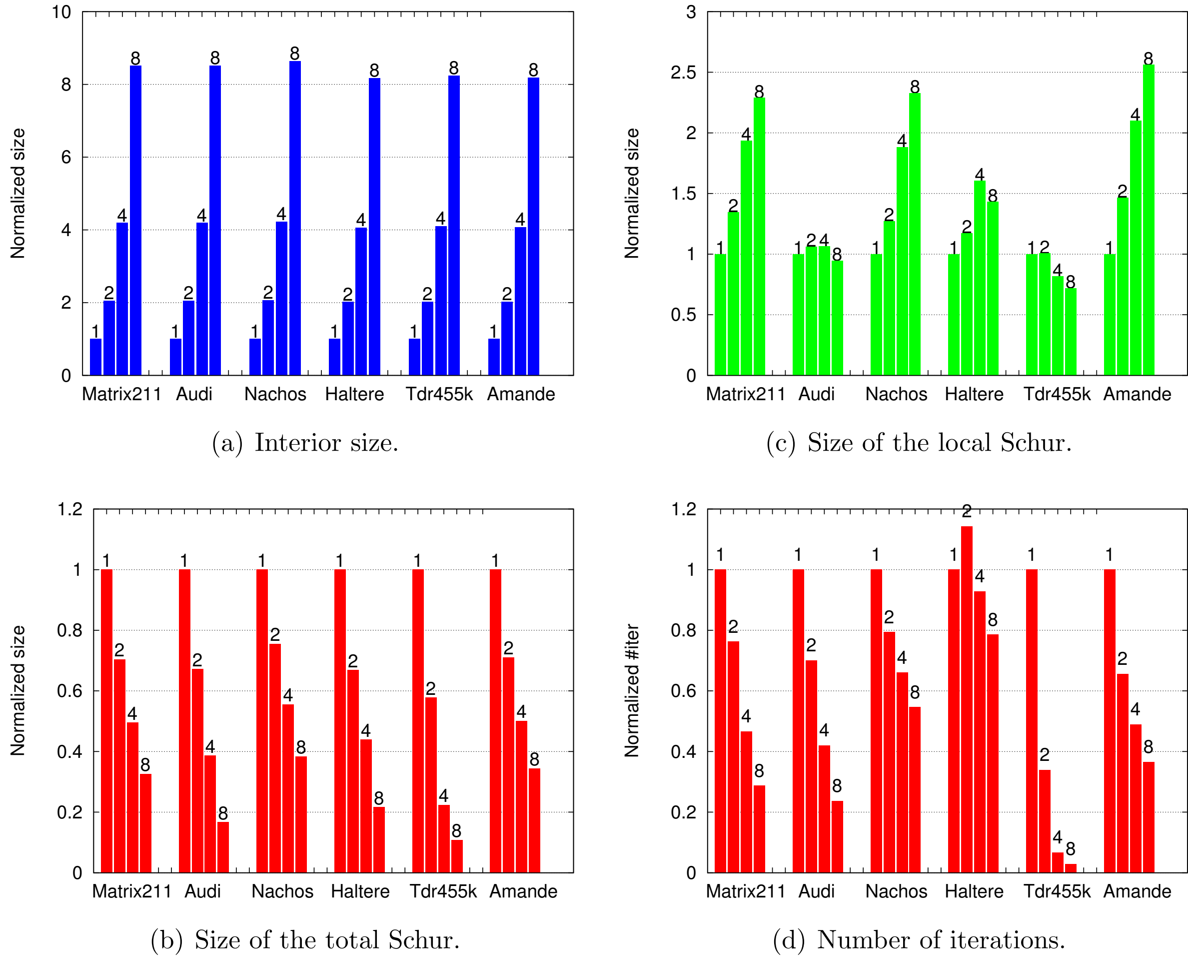



Figure 2.11: Numerical and parallel flexibility of the 2-level implementation for exploiting all cores of a node with 1, 2, 4 or 8 subdomains per node (set up in Section 2.4.4).

For the `Haltere` matrix the best performance is obtained when one thread per process is used. More specifically, each step yields the best performance when one thread per process is used. Considering the *solve step*, increasing the number of subdomains does not strongly deteriorate the quality of the preconditioner for this matrix. For instance, in this case, when the number of subdomains varies from 4 to 32, the number of iterations increases from 11 to only 14 respectively (Figure 2.12(d)). On that example, the best performance is obtained when the number of subdomains is the largest. Additionally, for this matrix with all configurations, the overall time is dominated by the *factorization of the interiors*. The *factorization of the interiors* yields best performance when the number of subdomains is the largest (Figure 2.13(b)). The main reason for this behavior is the fact that when the number of subdomains doubles, the size of each interior is approximately divided by two (see Figure 2.12(a)) and the number of threads are doubled. Since the complexity in terms of calculation for 3D problems for the sparse direct solver is squared (Figure 2.13(d)), larger number of domain yields better performance.

For the `Nachos` matrix, the best configuration is also when one thread per process is used. Contrary to the the `Haltere` matrix, the *solve step* is expensive for this matrix. Nevertheless, with the increasing of the number of subdomains, the number of iterations does not decrease drastically (Figure 2.12(d)). Since the solve step does not efficiently exploit the multithreading (Figure 2.10(d)), similar number of iterations will be executed in approximately the same amount of time (Figure 2.13(d)). Due to this, the overall execution time is driven by the *factorization of the interiors*, and as explained above, the factorization of the interior yields the best performance when one thread per process is used.

The `Amande` and `Matrix211` matrices both exhibit similar behaviors. The optimal performance is obtained when two threads per process are used. Compared to the `Nachos` matrix, the number of iterations decreases more rapidly. As a consequence the *solve step* becomes less expensive in terms of time consumption when the number of subdomains decreases




(a) Interior size.

(c) Size of the local Schur.

(b) Size of the total Schur.

(d) Number of iterations.

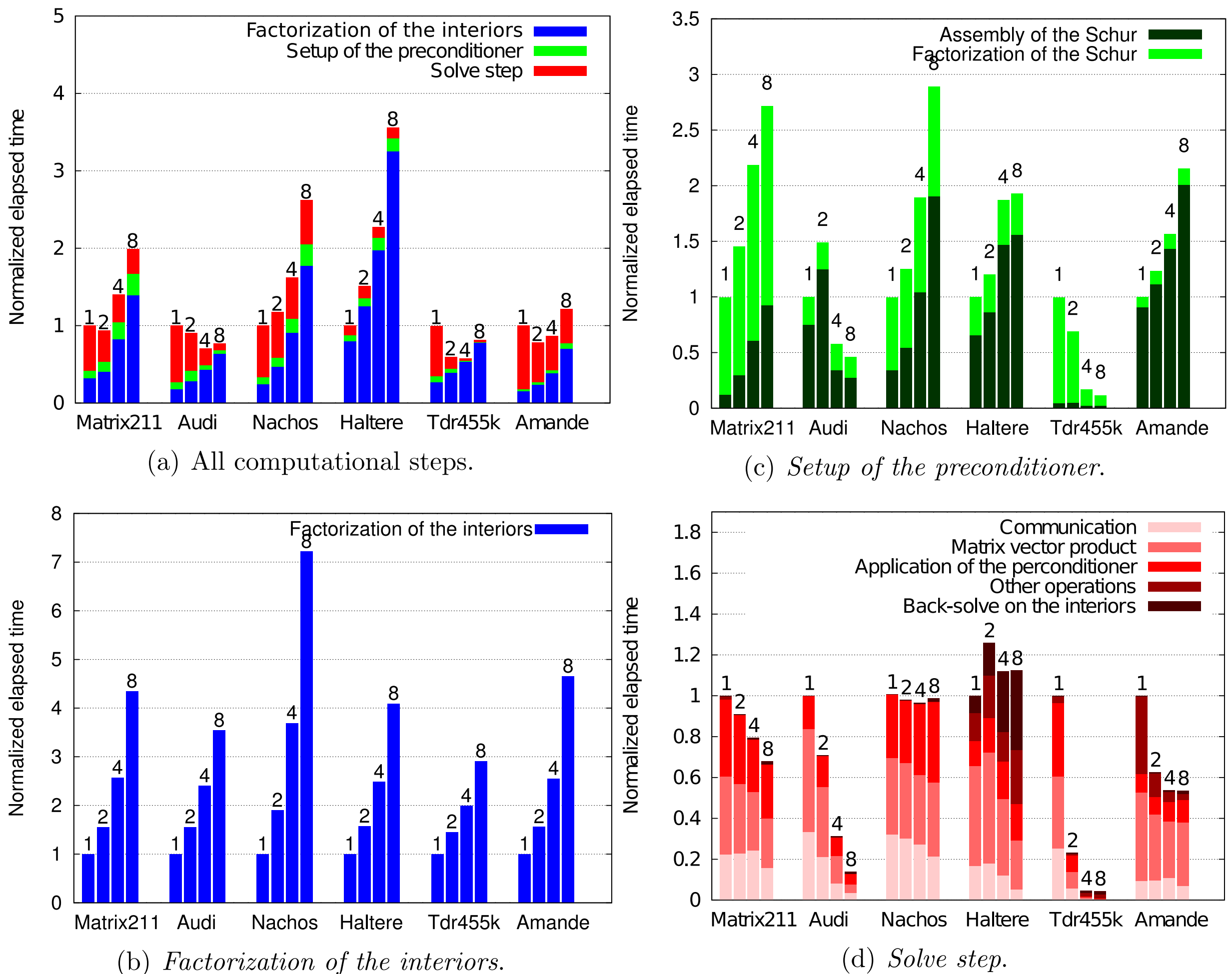
Figure 2.12: The average sizes of the interior (2.12(a)), the local Schur complement (2.12(c)), the total Schur complement (2.12(b)) and the number of iterations (2.12(d)) for each matrix are presented when all available cores are used and the following statement is satisfied : $nb_nodes \times nb_cores = nb_threads_per_process \times nb_subdomains$. The number of nodes and the preconditioner are given in Table 2.3 and the number of threads used per process is given on top of each histogram bar. All histograms are normalized with respect with the histogram when one thread per process is used.

| | | #subdomains | | | | |
|---|---|---|---|---|---|---|
| | | 8 | 16 | 32 | 64 | 128 |
| Preconditioner | dense | 9 | 21 | 107 | 315 | 6323 |
| | $10^{-4}$ | 31 | 108 | 502 | – | – |
| | $10^{-3}$ | 168 | 1002 | – | – | – |
| | $10^{-2}$ | 6496 | – | – | – | – |

Table 2.4: Number of iterations for different configurations (#subdomains and preconditioner) for the `Tdr455k` matrix with the GMRES algorithm. ”–” means that the algorithm fails to achieve convergence in 7000 iterations with restart of 500.

(Figure 2.13(d)). Moreover, both matrices are issued from difficult numerical problems, resulting in a large number of iterations during the solve step. For instance, when one thread per process is used, 223 and 468 iterations are needed for the `Matrix211` and `Amande` matrices, respectively. Therefore, the benefits of reducing the cost of the solve step have larger consequences on the overall computational time (Figure 2.13(d)).

For the two remaining matrices, `Audi_kw` and `Tdr455k`, this phenomena is even stronger. Additionally, when the number of subdomains increses, the size of the local Schur complement is slightly decreased (Figure 2.12(c)). Since the dense preconditioner is used for the `Tdr455k` matrix, the *setup of the preconditioner* yields better performance with the multi-threaded version for this matrix. For these matrices increasing the number of subdomains has the most significant influence on the number of iterations. Therefore the solve step yields the best performance when eight threads per process are used.




(a) All computational steps.

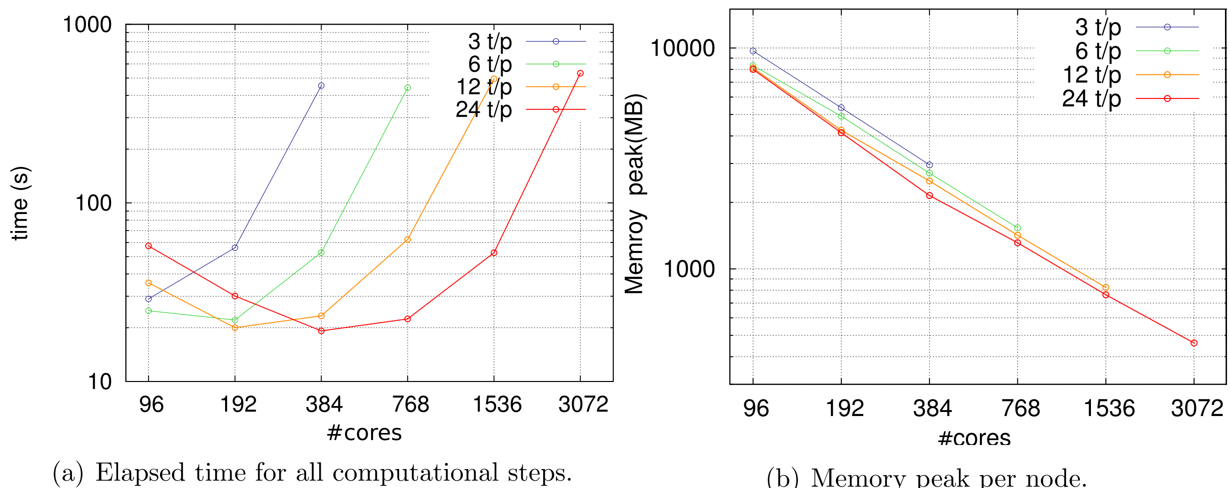
(b) *Factorization of the interiors.*

(c) *Setup of the preconditioner.*

(d) *Solve step.*

Figure 2.13: In these experiments all available cores are used and the following statement is satisfied : $nb_nodes \times nb_cores = nb_threads_per_process \times nb_subdomains$. The number of nodes and the preconditioner are given in Table 2.3 and the number of threads used per process is given on top of each histogram bar. Each histogram is normalized in comparison to the time obtained when one thread per subdomain is used. The test were performed on the `PlaFRIM` platform.

Furthermore, the `Tdr455k` matrix is the most difficult matrix to solve in our collection for our algorithm. The results presented in Figure 2.13 correspond to a dense preconditioner. Performing dropping on the local Schur complement strongly decreases the quality of the preconditioner (Table 2.4). Additionally, decomposing the matrix in larger number of subdomains increases strongly the number of iterations for each preconditioner, failing to converge in most cases. This matrix has strong numerical barriers for our algorithm. With the multithreaded version, we are able to break the numerical barrier by assigning larger number of cores per subdomain. On the `Hopper` platform we are able to efficiently exploit as much as 384 cores (see Figure 2.14(a)) and in the same time lower the memory peak per node (Figure 2.14(b)).



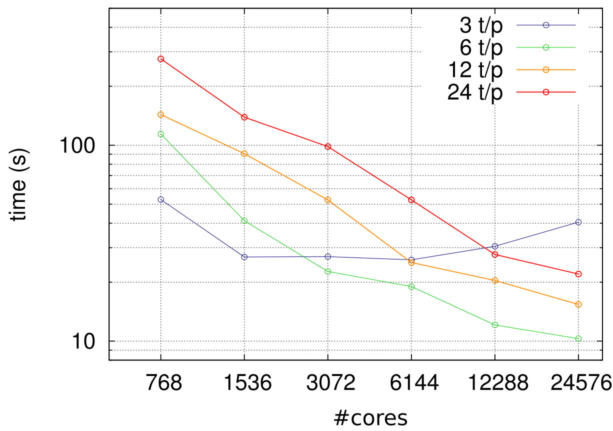

(a) Elapsed time for all computational steps.

(b) Memory peak per node.

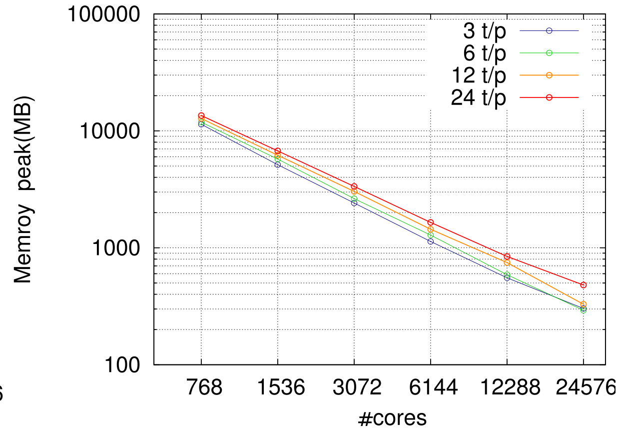
Figure 2.14: The maximum time 2.14(a) and memory peak per node 2.14(b) when the `Tdr455k` matrix is used with dense preconditioner. All the available cores per node are used and the statement is satisfied: $nb_nodes \times nb_cores = nb_threads_per_process \times nb_subdomains$. In the legend $\frac{t}{p}$ refers to number of threads $(t)$ per process $(p)$.

In order to push the limit of our algorithm, we have furthermore performed tests on the `Nachos4M` matrix on the `Hopper` platform. In these series of experiments we have increased the number of nodes up to one thousand (see Figure 2.15). With the increased number of nodes, the number of subdomains is also increased up to $2^{13}$ when three threads per process are used. When the number of nodes is increased, the solve step dominates more and more the overall execution time (the red portion in each histogram in Figure 2.15(b)).

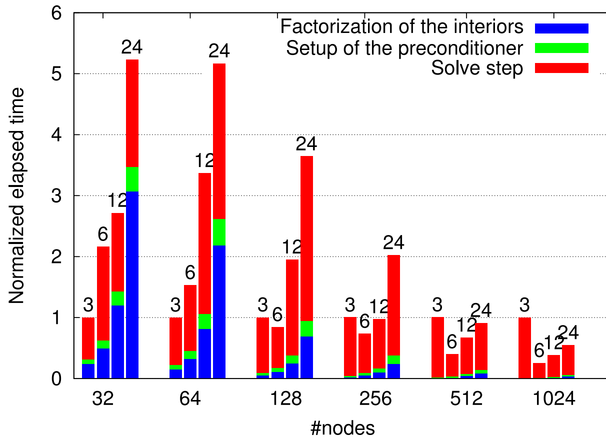
Although not illustrated up-to now, depending on the targeted accuracy, the best choice of the number of subdomains can vary for a given number of cores to be used. In Figure 2.16 we display the convergence history as a function of the elapsed time and iteration number for the `Audi_kw` matrix (Figures 2.16(a) and 2.16(b) respectively). Lower the number of subdomains, larger the time needed for *factorization of the interiors* and *setup of the preconditioner* is, and faster is the convergence. However, for a moderated target accuracy, this setup cost is not worth to invest. It might be more effective to enlarge the number of subdomains to shrink the setup time and having slower convergence rate but overall faster solution. For instance, for a $10^{-4}$ accuracy, the best choice is 64 subdomains (one thread



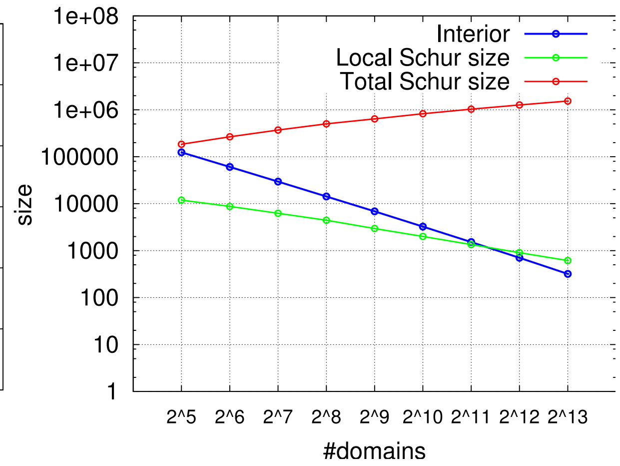

(a) Elapsed time for all computational steps.




(c) Memory peak per node.



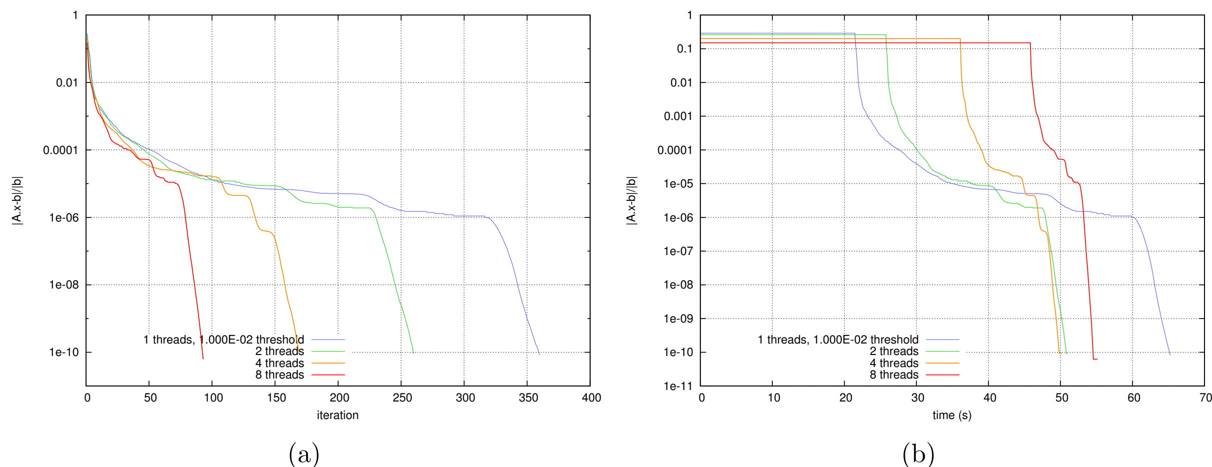

(b) All computational steps.




(d) Sizes.

Figure 2.15: The maximum time 2.15(a) and memory peak per node 2.15(c) for the `Nachos4M` matrix with sparse preconditioner when a dropping is applied to the preconditioner with threshold of $10^{-02}$. The detailed histogram for all computational steps is given in 2.15(b) and the sizes for the interior, the local Schur complement and the total size of the Schur complement are given in 2.15(d). The tests were performed on the `Hopper` platform. All the available cores per node are used and the statement is satisfied: $nb_nodes \times nb_cores = nb_threads_per_process \times nb_subdomains$. In the legend $\frac{t}{p}$ refers to number of threads ($t$) per process ($p$).

each) while for a $10^{-9}$ accuracy the best choice is 16 subdomains (i.e., 4 threads).




(a)

(b)

Figure 2.16: Convergence history with respect with the elapsed time for the `Audi_kw` matrix when four nodes are used in respect with the iteration (Figure 2.16(a)) and in respect with the time (Figure 2.16(b)). Dropping with a threshold of $10^{-2}$ is applied to the preconditioner.

## 2.5 Case study from geoscience applications

In this section we study on the numerical and parallel performance of MaPHyS for the solution of large linear systems arising from the discretization of 3D elastodynamic equations in the frequency domain. The elastodynamic equation, briefly introduced in Section 1.1 and fully described in [26], are discretized using a discontinuous Galerkin (DG) method defined on a fully unstructured 3D mesh similar to the one depicted in Figure 2.17. Different polynomial degrees are considered for the finite element approximation, namely P2, P3 and P4. For the experiments with P2 and P3, the mesh size is adjusted so that the two discretization schemes lead to linear systems of a similar dimension (i.e., a coarser mesh is used for the $P3$ approximation). The features in terms of size and non zeros for these complex symmetric matrices are displayed in Table 2.5. It can be observed that the higher the polynomial degree, the larger the number of non zeros per row yielding a less and less sparse matrix. Furthermore, with matrices arising from a DG discretization exhibit a special pattern with dense blocks (whose dimensions depend on the polynomial degree) and a connectivity between the blocks related to the mesh connectivity. Because MaPHyS is designed to be a general purpose hybrid solver, we have made no attempt to exploit this *a priori* additional information. All the calculations have been performed in double precision complex arithmetic using a restart of 500 for GMRES with right preconditioner (i.e., right preconditioned GMRES(500)) and a stopping convergence criterion threshold $\varepsilon_b = 10^{-04}$ (Equation 2.13).

For all the experiments performed in this section, we consider a right-hand side corresponding to a source located at the center of the cube. The solution computed by MaPHyS

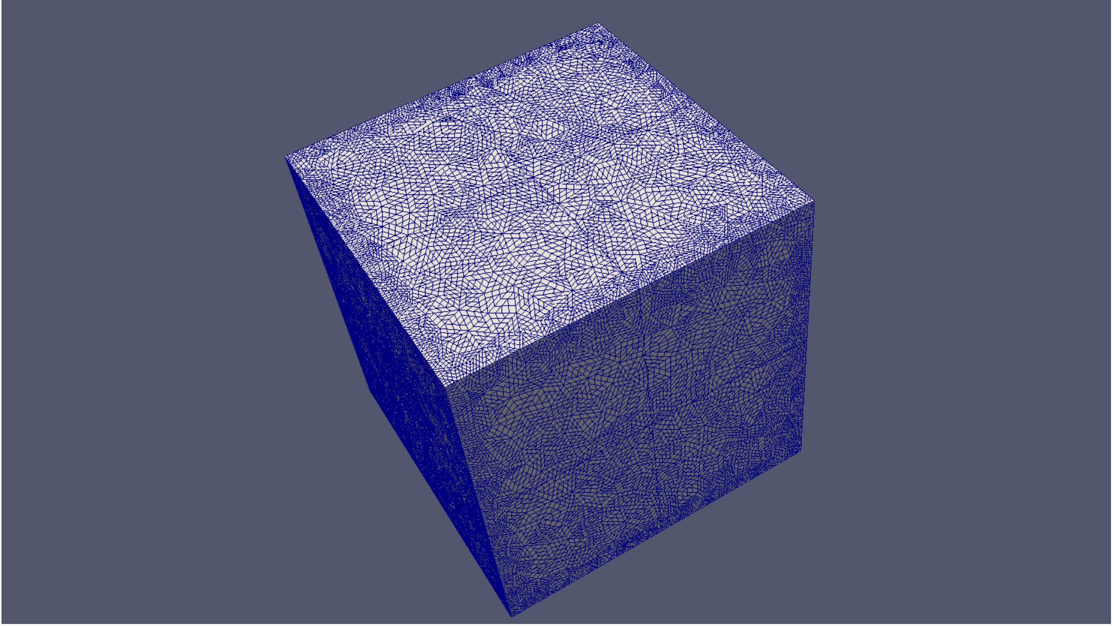


Figure 2.17: A mesh used for the elastodynamic simulations.

| Matrix | `Matrix_P2` | `Matrix_P3` | `Matrix_P4` |
|---|---|---|---|
| $n$ | 1.222M | 1.244M | 2.177M |
| $nnz$ | 89M | 178M | 546M |
| Symmetry | symmetric | symmetric | symmetric |
| Arithmetic | complex | complex | complex |

Table 2.5: Main features of the discontinuous Galerkin discretization matrices.

is displayed in Figure 2.18; the medium is homogeneous, consequently the wave has a spherical structure.

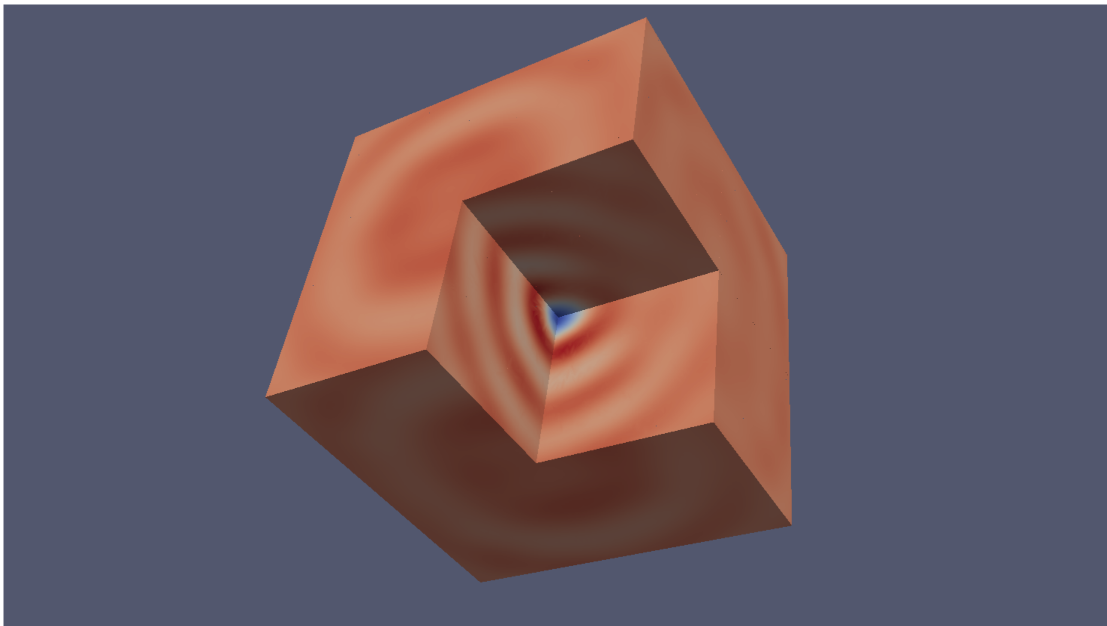


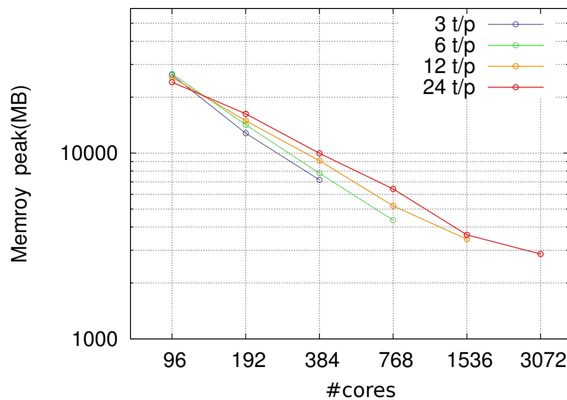
Figure 2.18: Computed solution, the source is at the center of the cube.

For the `Matrix_P2` matrix, we report the number of iterations when the number of subdomains varies in Table 2.6. It can be seen that the convergence deteriorates progressively and MaPHyS does not converge anymore with 256 subdomains. The same trend is observed when a sparse preconditioner is applied and amplified for the largest (i.e., $\tau = 10^{-3}$) threshold parameter that leads to the sparser and numerically poorer preconditioner (less than 5 % of the assembled local Schur complement matrices are kept to build the preconditioner). For those experiments, we display in Figure 2.19 the maximum memory consumption when the number of subdomains and cores per subdomain varies. Similarly to what was observed for the `Nachos` matrix, also arising from a 3D DG discretization, for a given number of cores, the larger the number of subdomains (i.e., the lower the number of threads), the lower the memory consumption is. This can be explained by the statistics given in Figure 2.19(b), where it can be seen that, even though the size of the global Schur complement increases with the number of subdomains, the size of the local interior decreases, so do the sizes of the associated local factors, as well as the size of the local Schur complement matrices.

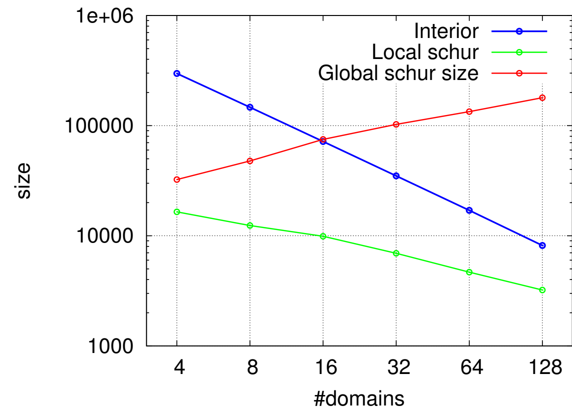
Following the same experimental approach presented in Section 2.4.4, we vary the number of threads per subdomain so that for a given number of cores we can tune the balance between the iterative and direct computation. For the dense variant of the preconditioner, we display the total elapsed time-to-solution as the function of the number of cores for various number of threads per subdomain (value on top of the bars in Figure 2.20(a)). To evaluate the possible benefit of using fewer subdomains with a larger number of threads per subdomain we report the normalized elapsed time in Figure 2.20(b), with respect to the elapsed time when three thread per process are used. In particular on 96 cores, one can see that using 24 threads per subdomain does reduce the number of iterations (see Table 2.6); consequently the minimum time spent in the iterative part (i.e., red bar associated with

| | | #subdomains | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 4 | 8 | 16 | 32 | 64 | 128 | 256 |
| Preconditioner | dense | 33 | 57 | 89 | 182 | 451 | 2970 | – |
| | sparse $10^{-4}$ | 33 | 58 | 90 | 184 | 448 | 2955 | – |
| | | (6.5%) | (8.6%) | (11.6%) | (13.3%) | (15.5%) | (15.7%) | – |
| | sparse $10^{-3}$ | 38 | 64 | 106 | 237 | 721 | 4441 | – |
| | | (1.0%) | (2%) | (2.3%) | (3.2%) | (4.5%) | (4.8%) | – |

Table 2.6: Number of iterations for different configurations (#subdomains and preconditioner) for the `Matrix_P2` matrix with the GMRES algorithm. The average percentage of the kept entries on the local assembled Schur complement ($\tilde{\tilde{\mathcal{S}}}_i$) is given for each sparse preconditioner. "–" means that the algorithm fails to achieve convergence in 7000 iterations with a restart of 500.



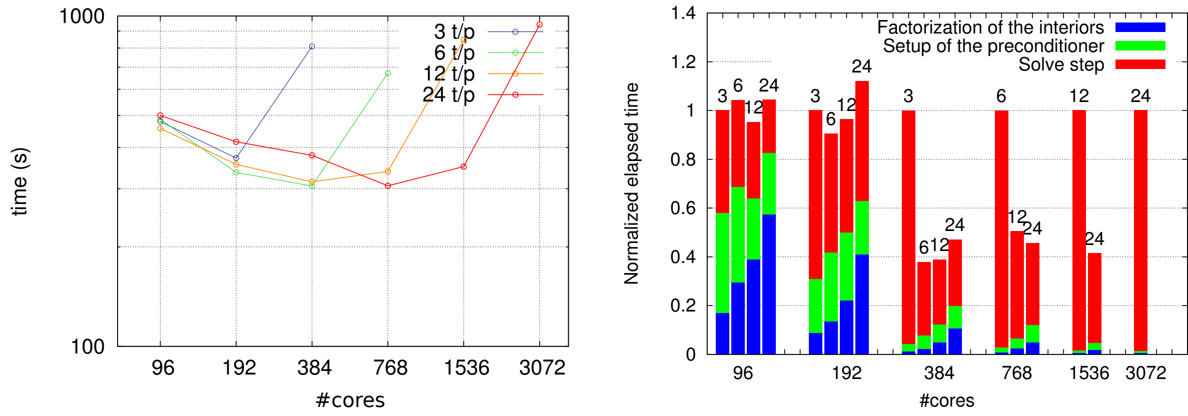

(a) Memory peak per node.




(b) Sizes of the interior, local and global Schur.

Figure 2.19: The memory peak per node for the `Matrix_P2` matrix is given in 2.19(a). All the available cores per node are used and the statement is satisfied: $nb_nodes \times nb_cores = nb_threads_per_process \times nb_subdomains$. The sizes for the interior ($\mathcal{I}_i$), the local Schur complement ($\mathcal{S}_i$) and the size of the global Schur complement ($\mathcal{S}$) are given in 2.19(b).

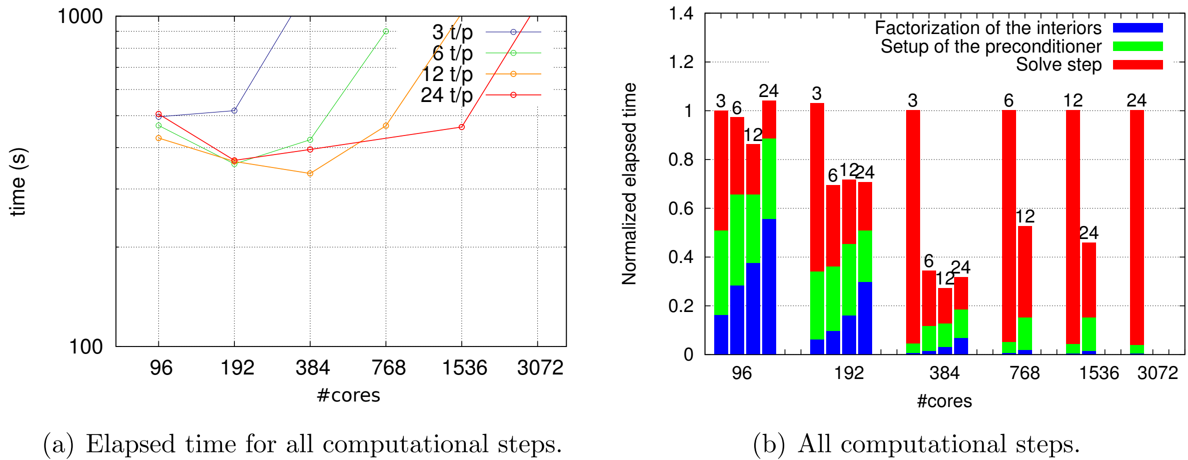
solve step) is achieved at a price of a much higher time spent in the direct calculation (i.e., blue bar). These chart bar highlight the numerical difficulties encountered in the iterative part to get the solution as the *solve step* dominates when the number of cores increases. Similar trends can be observed in Figure 2.21 and 2.22, when dropping is applied to sparsify the preconditioner. On that example, when the sparsification policy with the $10^{-3}$ threshold is applied, it is the slowest in terms of iterations, but it is the fastest in terms of time.




(a) Elapsed time for all computational steps.

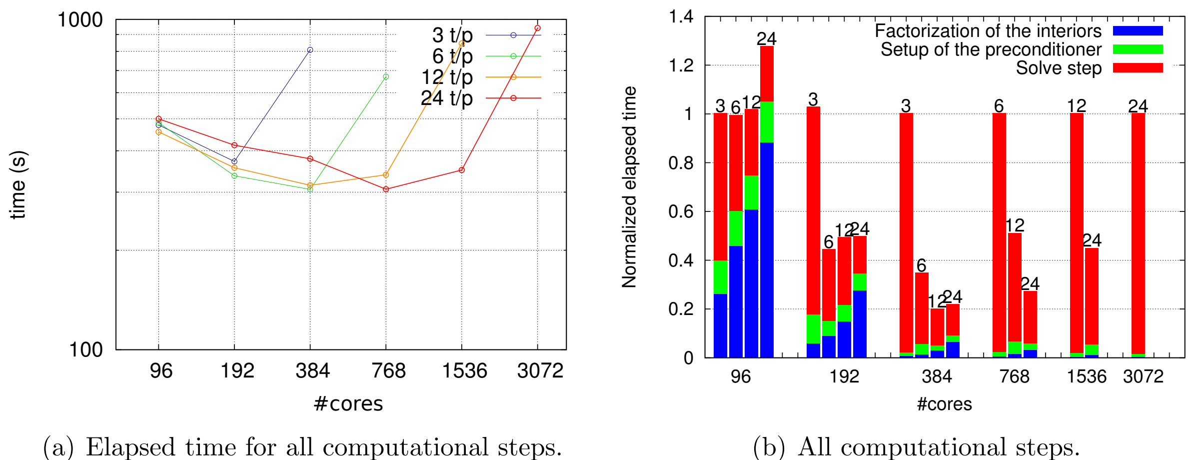
(b) All computational steps.

Figure 2.20: The elapsed time for the `Matrix_P2` matrix when no dropping is applied to the preconditioner is given in 2.20(a). The detailed histogram for all computational steps is given in 2.20(b). All the available cores per node are used and the statement is satisfied: $nb_nodes \times nb_cores = nb_threads_per_process \times nb_subdomains$. These tests were executed on the `Hopper` platform. In the legend $\frac{t}{p}$ refers to number of threads ($t$) per process ($p$).




(a) Elapsed time for all computational steps.
(b) All computational steps.

Figure 2.21: The elapsed time for the `Matrix_P2` matrix when dropping is applied to the preconditioner with a threshold of $10^{-4}$ is given in 2.21(a). The detailed histogram for all computational steps is given in 2.21(b). All the available cores per node are used and the statement is satisfied: $nb_nodes \times nb_cores = nb_threads_per_process \times nb_subdomains$. These tests were executed on the `Hopper` platform. In the legend $\frac{t}{p}$ refers to number of threads ($t$) per process ($p$).



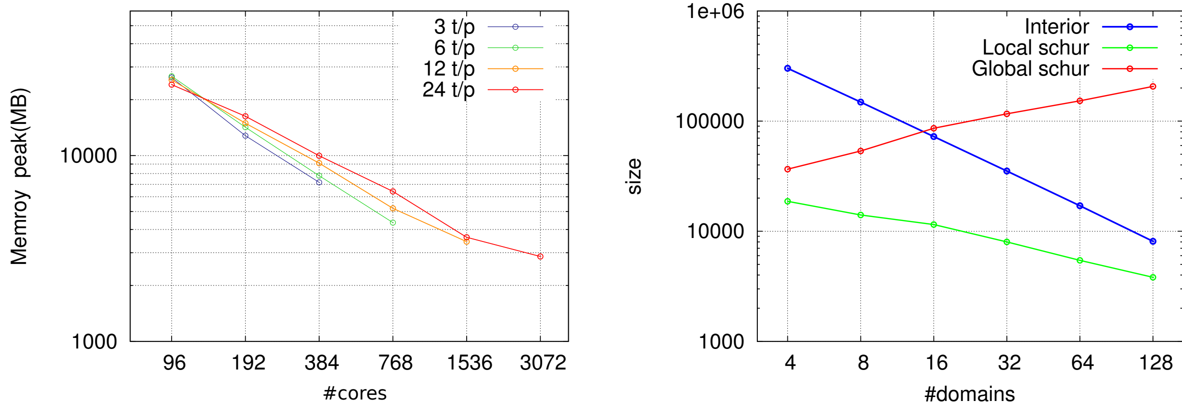

(a) Elapsed time for all computational steps.
(b) All computational steps.

Figure 2.22: The elapsed time for the `Matrix_P2` matrix when dropping is applied to the preconditioner with a threshold of $10^{-3}$ is given in 2.22(a). The detailed histogram for all computational steps is given in 2.22(b). All the available cores per node are used and the statement is satisfied: $nb_nodes \times nb_cores = nb_threads_per_process \times nb_subdomains$. These tests were executed on the `Hopper` platform. In the legend $\frac{t}{p}$ refers to number of threads ($t$) per process ($p$).

Table 2.7 gathers the number of iterations for the `Matrix_P3` problem when the number of subdomains varies for different preconditioners. The calculation with 4 subdomains could not be ran because it exceeds the memory capacity of a node on `Hopper`. Compared to the `Matrix_P2` problem, which has a similar number of unknowns and provides a similar "quality" of the geophysics solution, it can be seen that the number of iterations is significantly larger for `Matrix_P3` (except for 128 subdomains for the dense preconditioner). Direct consequences of this higher numerical difficulty can be observed in Figure 2.24 and 2.25, where the *solve step* dominate and the sparsification strategy never outperforms the dense variant.

| | | #subdomains | | | | | |
|---|---|---|---|---|---|---|---|
| | | 8 | 16 | 32 | 64 | 128 | 256 |
| Preconditioner | dense | 120 | 170 | 317 | 693 | 2593 | – |
| | sparse $10^{-4}$ | 125 | 180 | 363 | 883 | 3478 | – |
| | | (4.9%) | (5.9%) | (6.7%) | (8.0%) | (9.0%) | – |
| | sparse $10^{-3}$ | 274 | 396 | 774 | – | – | – |
| | | (2.0%) | (2.0%) | (2.3%) | – | – | – |

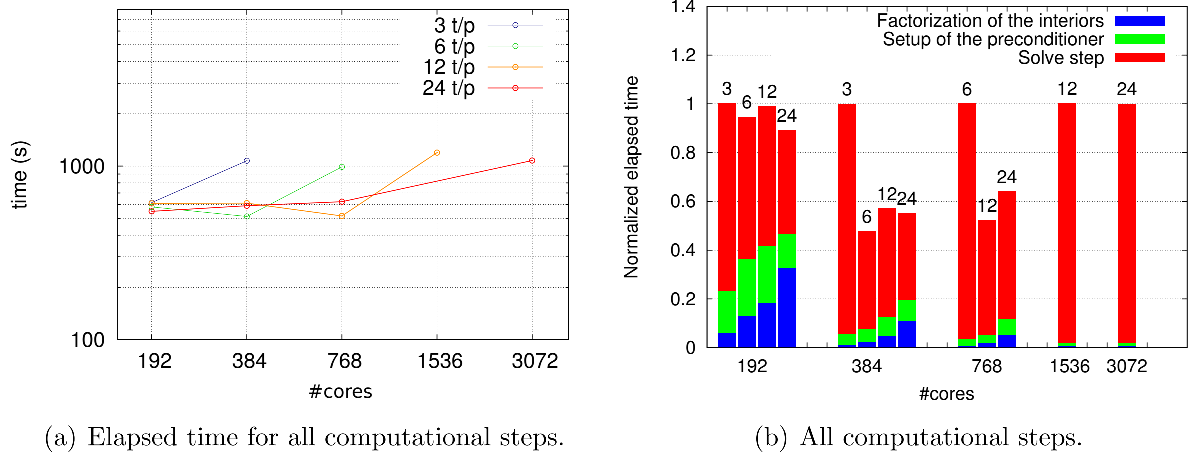
Table 2.7: Number of iterations for different configurations (#subdomains and preconditioner) for the `Matrix_P3` matrix with the GMRES algorithm. The average percentage of the kept entries on the local assembled Schur complement ($\widetilde{\widetilde{\mathcal{S}}}_i$) is given for each sparse preconditioner. "–" means that the algorithm fails to achieve convergence in 7000 iterations with a restart of 500.




(a) Memory peak per node.

(b) Sizes of the interior, local and global Schur.

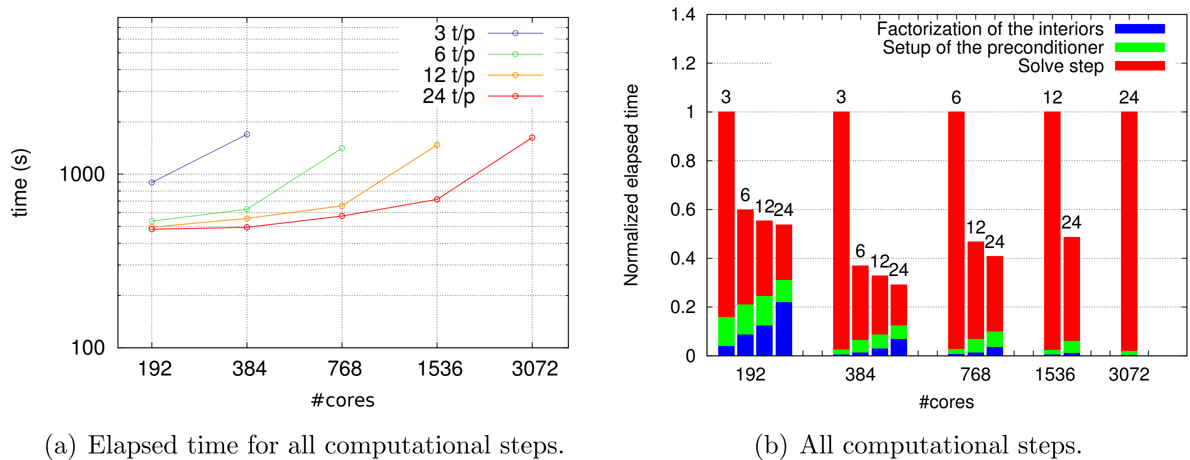
Figure 2.23: The maximum memory peak per node for the `Matrix_P3` matrix is given in 2.23(a). All the available cores per node are used and the statement is satisfied: $nb_nodes \times nb_cores = nb_threads_per_process \times nb_subdomains$. The sizes for the interior ($\mathcal{I}_i$), the local Schur complement ($\mathcal{S}_i$) and the size of the global Schur complement ($\mathcal{S}$) are given in 2.23(b). In the legend $\frac{t}{p}$ refers to number of threads ($t$) per process ($p$).




(a) Elapsed time for all computational steps.

(b) All computational steps.

Figure 2.24: The elapsed time for the `Matrix_P3` matrix when no dropping is applied to the preconditioner is given in 2.24(a). The detailed histogram for all computational steps is given in 2.24(b). All the available cores per node are used and the statement is satisfied: $nb_nodes \times nb_cores = nb_threads_per_process \times nb_subdomains$. These tests were executed on the `Hopper` platform. In the legend $\frac{t}{p}$ refers to number of threads ($t$) per process ($p$).




(a) Elapsed time for all computational steps.

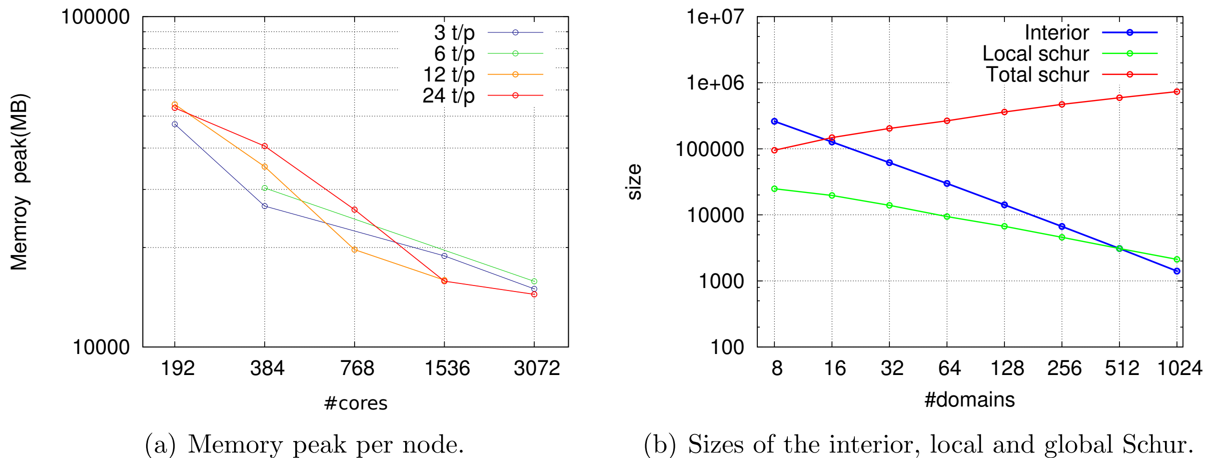
(b) All computational steps.

Figure 2.25: The elapsed time for the `Matrix_P3` matrix when dropping is applied to the preconditioner with a threshold of $10^{-4}$ is given in 2.25(a). The detailed histogram for all computational steps is given in 2.25(b). All the available cores per node are used and the statement is satisfied: $nb_nodes \times nb_cores = nb_threads_per_process \times nb_subdomains$. These tests were executed on the `Hopper` platform.

The `Matrix_P4` matrix is much larger than the former ones, but it does not translate in the difficulty to compute the solution; the number of iterations are lower that for the two previous examples. Contrarily to the two other examples, the number of iterations does not monotonically increase with the number of subdomains but even decreases after 256 subdomains. Similar behavior was reported in [78] for seismic simulations based on simple modeling (acoustic equation instead of elastodynamic here). Some additional investigations would deserve to be performed to better understand this behavior and it will be the topic of future research.

| | #subdomains | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 |
| Dense preconditioner | 55 | 77 | 145 | 350 | 974 | – | 1956 | 943 |

Table 2.8: Number of iterations for different configurations (#subdomains and preconditioner) for the `Matrix_P4` matrix with the GMRES algorithm. "–" means that the algorithm fails to achieve convergence in 7000 iterations with a restart of 500.

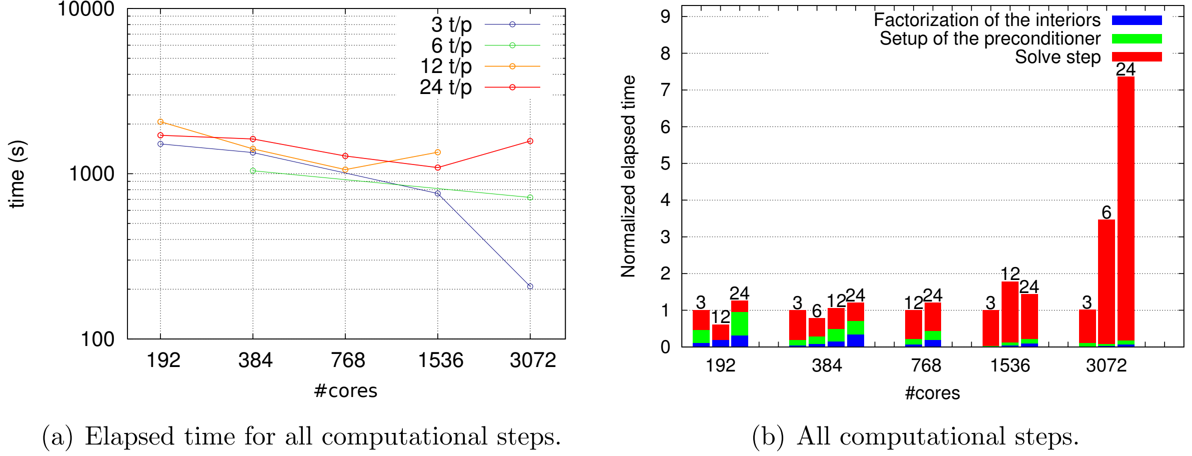



(a) Memory peak per node.

(b) Sizes of the interior, local and global Schur.

Figure 2.26: The maximum memory peak per node for the `Matrix_P4` matrix is given in 2.26(a). All the available cores per node are used and the statement is satisfied: $nb_nodes \times nb_cores = nb_threads_per_process \times nb_subdomains$. The sizes for the interior ($\mathcal{I}_i$), the local Schur complement ($\mathcal{S}_i$) and the size of the global Schur complement ($\mathcal{S}$) are given in 2.26(b). In the legend $\frac{t}{p}$ refers to number of threads ($t$) per process ($p$).

In this section we have reported on a few experiments performed on large sparse linear systems solution arising from the discretization of the 3D elastodynamic equation on 3D unstructured meshes and DG discretization. We have shown that the MPI+thread implementation of MaPHyS provides the user with the capability to trade-off between memory consumption, numerical robustness and time to the solution for a given number of cores dedicated to the simulation.




(a) Elapsed time for all computational steps.

(b) All computational steps.

Figure 2.27: The elapsed time for the `Matrix_P4` matrix when no dropping is applied to the preconditioner is given in 2.27(a). The detailed histogram for all computational steps is given in 2.27(b). All the available cores per node are used and the statement is satisfied: $nb_nodes \times nb_cores = nb_threads_per_process \times nb_subdomains$. These tests were executed on the `Hopper` platform. In the legend $\frac{t}{p}$ refers to number of threads ($t$) per process ($p$).

## 2.6 Performance sensitivity with respect to the partitioning quality

In the context of MaPHyS, the objectives in terms of load balancing for the decomposition domain tool (see Section 2.3.1) would be to decompose the adjacency graph associated to the initial sparse matrix into a set of disjoint subgraphs (subdomains) with both interior sets and interface sets (local Schur complement) of similar cardinality, while minimizing also the size of the global Schur complement for the complete system of equations.

To illustrate this point, let us denote $n_i$ and $s_i$ the number of nodes in the interior and in the interface for subdomain $i$ respectively. If we consider the familly of graphs corresponding to sparse matrices arising from 3D finite element discretizations and if we perform a nested dissection ordering on the subdomain interior nodes, the overall computation cost associated with subdomain $i$ grows asymptotically as

$$\mathcal{O}(n_i^2 + n_i^{4/3} s_i + n_i^{2/3} s_i^2) \ + \ \mathcal{O}(s_i^3) \ + \ \mathcal{O}(nb_iterations \times s_i^2)$$

where these three asymptotic terms correspond to the operation counts of the *factorization of the interiors* step [102], the *setup of the preconditioner* step (if we consider a dense preconditioner ) and the *solve step*, respectively. For 3D meshes with a *good aspect ratio* [109], which covers a very large application class for high performance numerical simulation, the size $s_i$ grows as $\mathcal{O}(n_i^{2/3})$. This clearly shows the impact of the size $s_i$ of the local interfaces, first to limit the computation cost associated to each subdomain $i$ which grows as $\mathcal{O}(n_i^2)$ for the two first asymptotic terms of the above formula, the third one depending more directly from the number of iterations in the *solve step*, and second to achieve a good balancing of the computation if the subdomains are distributed among the processors of the

compute platform. If we use a sparse preconditioner, the computation cost of the second asymptotic term will be smaller but the number of iterations in the third asymptotic term can be more important to achieve a similar convergence rate.

Finding a good domain decomposition with a sufficient quality in terms of interior and interface sizes is in fact a multi-objective optimization problem to be solved. In this section, we investigate the performance of the MaPHyS solver when the governing parameters of a classical graph partitioner is used. Contrary to the experiments presented so far where METIS was used to perform the domain decomposition, we rely on the Scotch [52] library in this section. This choise was made because Scotch's user API allows for easily tuning of the partitioning strategy. Scotch performs a recursive bipartioning heuristic which consists in finding at each step a well balanced bipartitioning of the current sugraph while minimizing the size of the separator. The objective is here to study the impact of relaxing the constraint on the balancing of the subgraph sizes (by adapting the value of the imbalance parameter of Scotch) on the ability to obtain smaller interfaces for the subdomains.

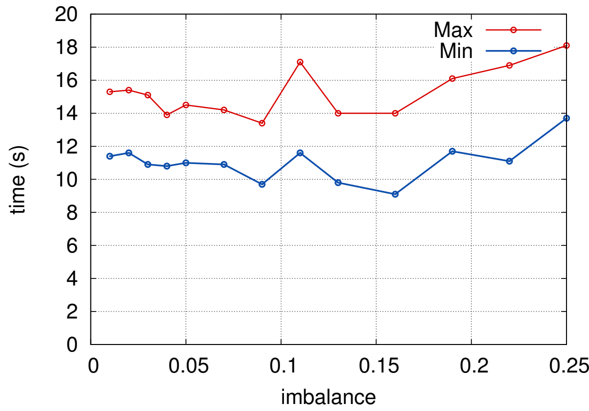
To illustrate this impact, we consider a set of test matrices (see Table 2.9 which gives the experimental parameters of MaPHyS runs for each matrix) and we vary the maximum imbalance parameter of Scotch that relaxes the constraint to have a "quasi-identical" size for the interior of the subdomains. More precisely, this imbalance parameter will vary from 1% to 25%. The tests are performed on the `PlaFRIM` platform.

| Matrix | `Matrix211` | `Audi_kw` | `Nachos` | `Haltere` | `Tdr455k` | `Amande` |
|---|---|---|---|---|---|---|
| #nodes | 8 | 4 | 16 | 4 | 8 | 32 |
| Preconditioner | dense | $10^{-3}$ | $10^{-2}$ | $10^{-3}$ | dense | $10^{-2}$ |
| #subdomains | 32 | 8 | 128 | 32 | 16 | 128 |
| #threads per process | 2 | 4 | 1 | 1 | 4 | 2 |

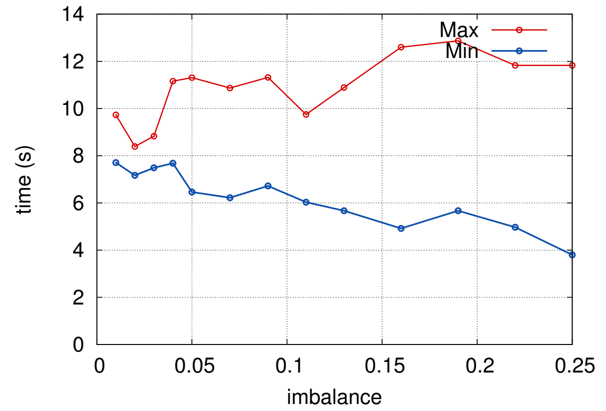
Table 2.9: Number of nodes, preconditioner type, number of subdomains, number of threads per process for each test matrix (see Figure 2.13). For each matrix, the optimal configuration (#threads per process vs #subdomains) is used.

First, we present in Figure 2.28 for each test matrix the impact of the imbalance factor on the total elapsed time when considering all the computational steps. In red (respectively in blue) are given the time values for the subdomain which has the biggest (respectively the smallest) computational cost and from a parallel elapsed time point of view, the maximum values are of course the most relevant. We can notice that the impact of the imbalance parameter is clear as there is a value for which the maximum time is "minimized" and this minimum is often, as expected, in the leftmost part of the curves (less than 10% of imbalance for all the cases).

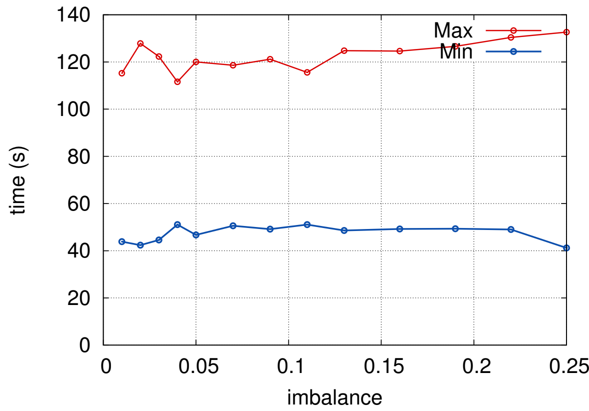
Then, we give for three test matrices (`Tdr455k`, `Audi_kw` and `Nachos`) the details of the impact of the imbalance parameter on each computational step : computation time for the *factorization of the interiors* step, computation time for the *setup of the preconditioner* step, computation time and number of iterations for the *solve step*. Finally, we display the sizes of the interiors (solid lines) and of interfaces (dotted lines). As above, except for the number of iterations which is in green, the maximum values are in red and the minimum values are



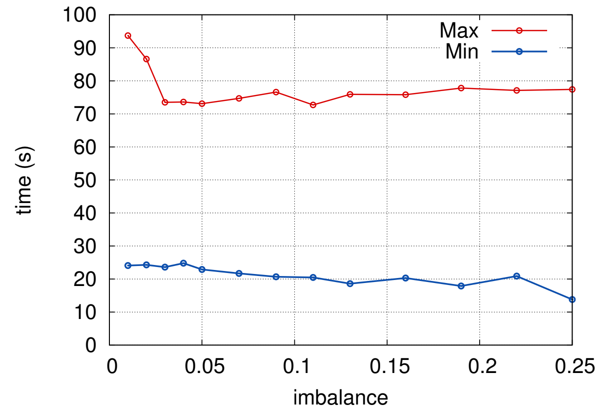

(a) Matrix211.



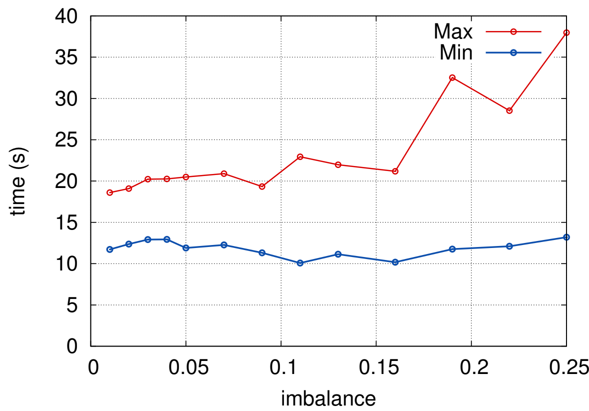

(d) Haltere.



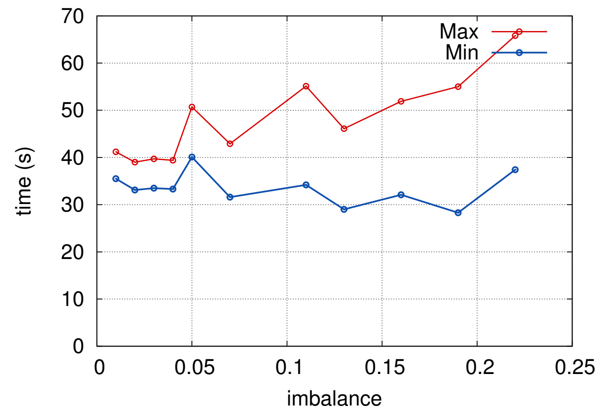

(b) Audi_kw.




(e) Tdr455k.




(c) Nachos.




(f) Amande.

Figure 2.28: Impact of the imbalance factor when using the SCOTCH partitioner on the total elapsed time for all computational steps. The experiments were performed on the PlaFRIM platform. The configuration for each matrix is given in Table 2.9.

in blue when considering the whole set of subdomains.

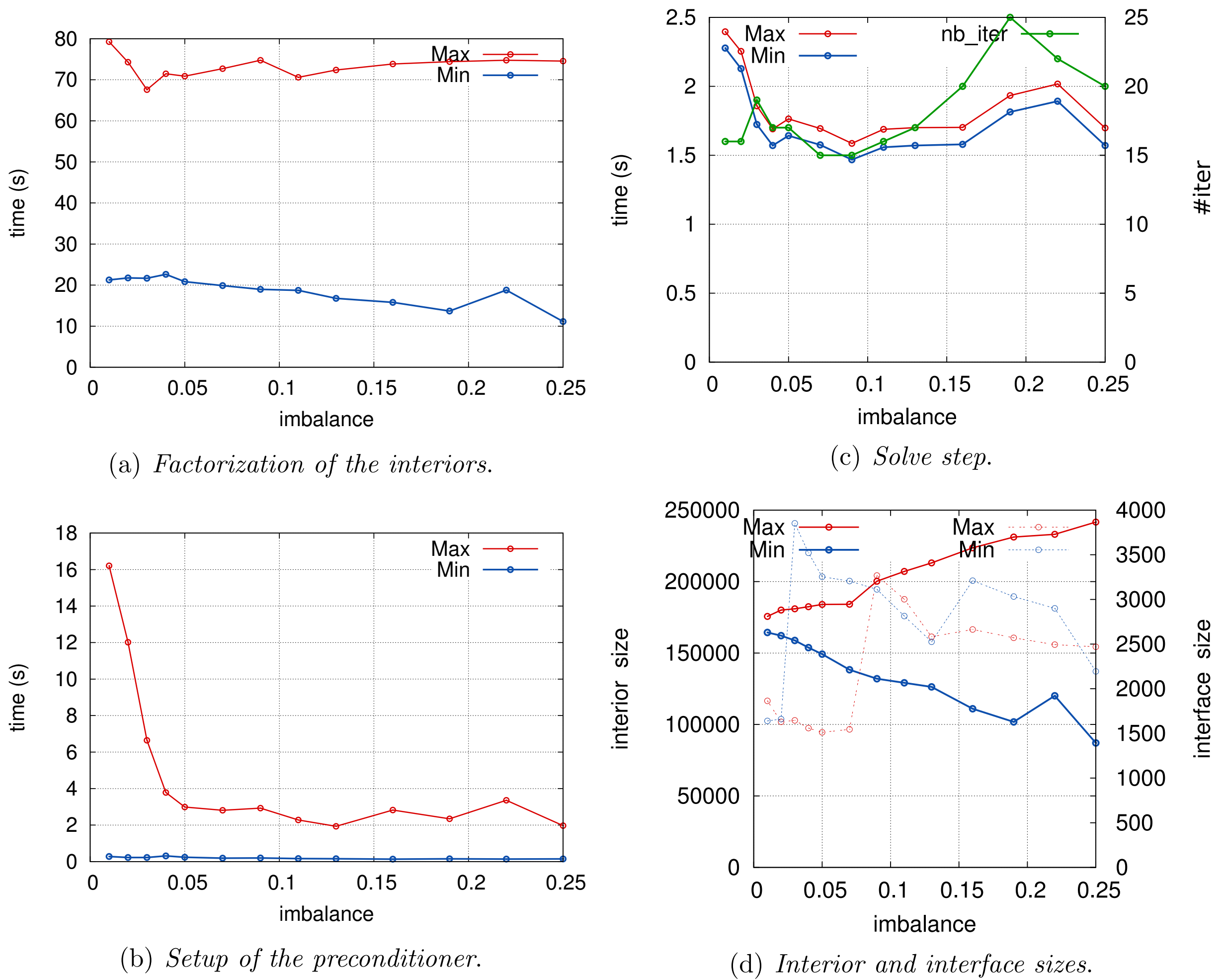
Let us consider the Tdr455k test case (see Figure 2.29). The first observation on the total elapsed time is that, as the imbalance parameter is increasing, it enlarges the difference between the largest and smallest subdomain sizes. Furthermore, as expected, increasing this parameter does not lead to a significant decrease of the maximum interface size that is nearly divided by a factor close to two (see Figure 2.29(d)).

From a parallel elapsed time point of view, the maximum values are the most interesting and we will only comment on them. Because the factorization time associated with the interior does not depend only on the size of the subdomain, but also on the size of the interface (see the first term of the asymptotic computational cost formula), the only increase of the size of the interior when the parameter is enlarged does not translate directly in a increase of the factorization time (see Figure 2.29(a)).

On the contrary for the setup of the dense preconditioner, the factorization cost is cubic with the interface size (see the second term of the asymptotic computational cost formula). Consequently, the decrease of the interface size translates into a significant decrease of this factorization time (see Figure 2.29(b)).

Of course, changing the interfaces changes the numerical convergence behavior of the iterative part as shown in the green line in Figure 2.29(c), where the number of iterations varies between 16 and 25. However, in the same time the size of the local interface decreases. The solution time, that takes into account for local matrix-vector products and applications of the preconditioner, decreases although the number of iterations increases. For the same reasons, a similar trend can be observed for the total elasped time displayed in Figure 2.28(e) where around 20 % of the elapsed time can be saved. For this matrix, the optimal performance are performed with 5% of imbalance.

Similar conclusions can be claimed for Audi_kw and Nachos test matrices. Those experiments show that each ingredient of the hybrid solver plays a role on the computational performance of the numerical scheme and that a good trade-off between the subdomain interior and interface sizes must be found to ensure good parallel performance.



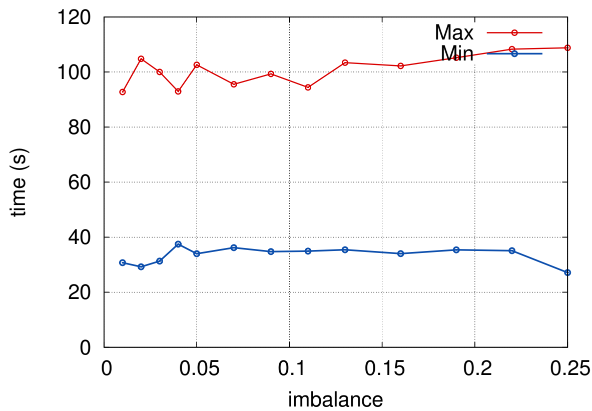

(a) *Factorization of the interiors.*

(c) *Solve step.*

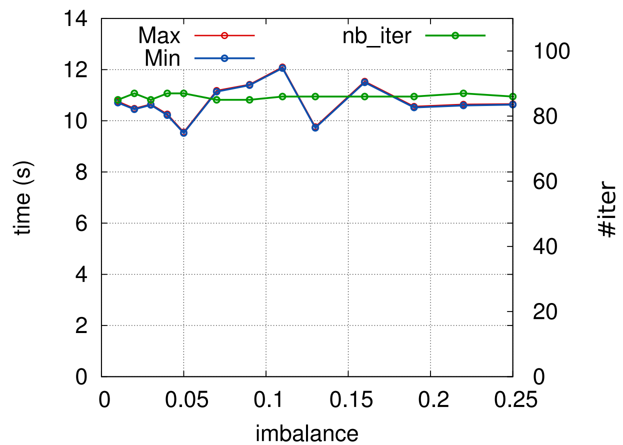
(b) *Setup of the preconditioner.*

(d) *Interior and interface sizes.*

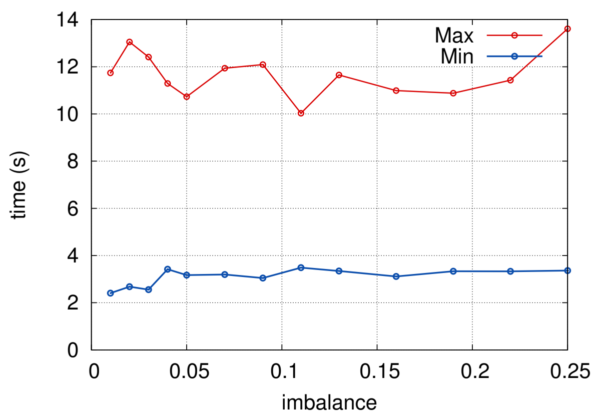
Figure 2.29: Impact of the imbalance factor on the *factorization of the interiors* 2.29(a), the *setup of the preconditioner* 2.29(b), the *solve step* 2.29(c) and on the *interior and interface sizes* 2.29(d) for the `Tdr455k` matrix. The experiments were performed on the `PlaFRIM` platform. The configuration for each matrix is given in Table 2.9.



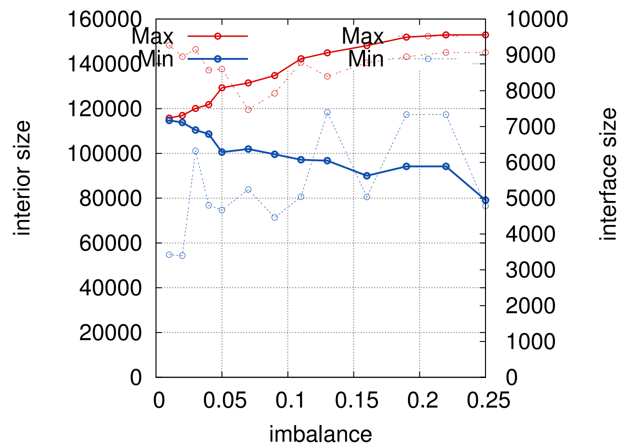

(a) *Factorization of the interiors.*




(c) *Solve step.*

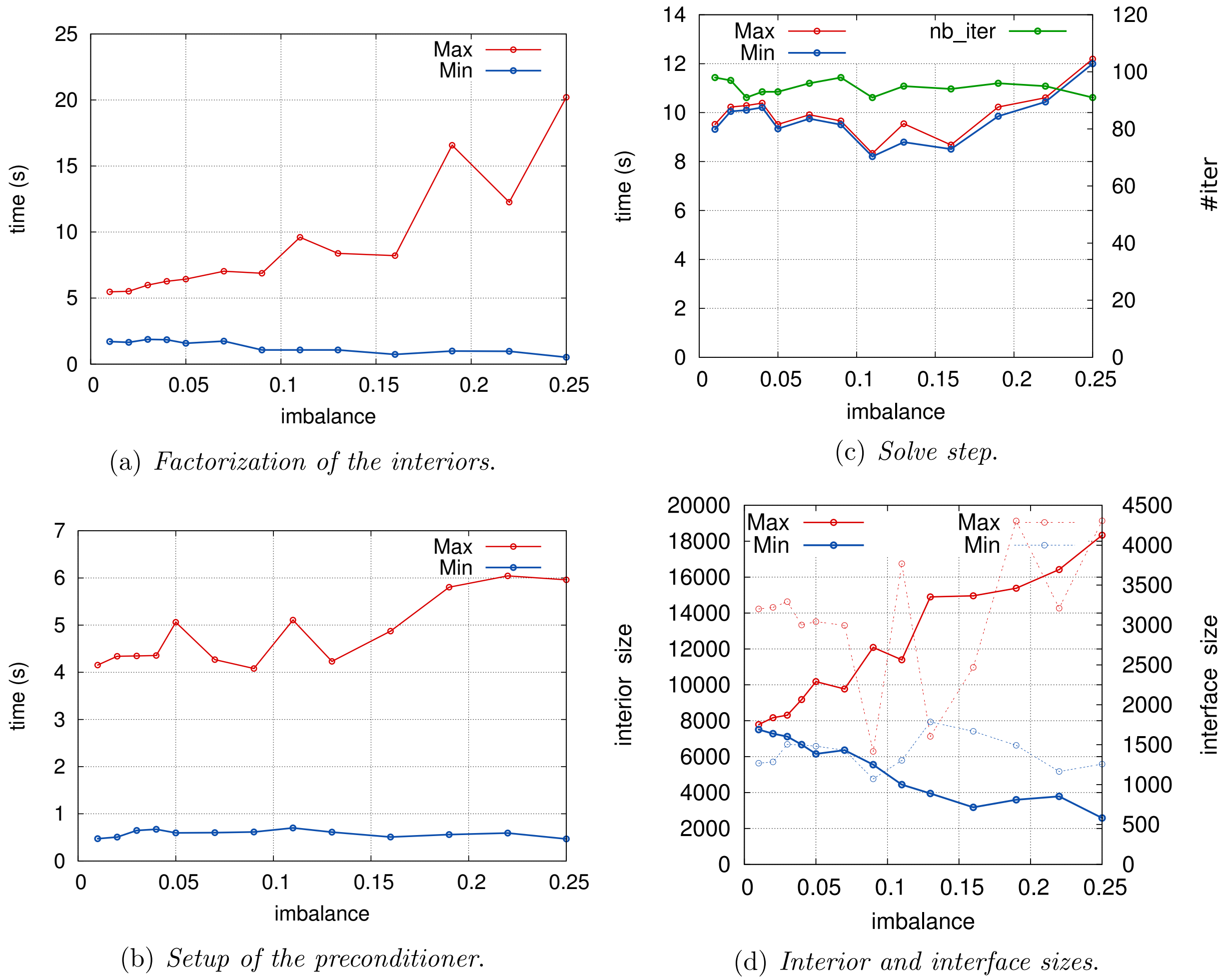



(b) *Setup of the preconditioner.*




(d) *Interior and interface sizes.*

Figure 2.30: Impact of the imbalance factor on the *factorization of the interiors* 2.30(a), the *setup of the preconditioner* 2.30(b), the *solve step* 2.30(c) and on the *interior and interface sizes* 2.30(d) for the `Audi_kw` matrix. The experiments were performed on the `PlaFRIM` platform. The configuration for each matrix is given in Table 2.9.




(a) *Factorization of the interiors.*

(b) *Setup of the preconditioner.*

(c) *Solve step.*

(d) *Interior and interface sizes.*

Figure 2.31: Impact of the imbalance factor on the *factorization of the interiors* 2.31(a), the *setup of the preconditioner* 2.31(b), the *solve step* 2.31(c) and on the *interior and interface sizes* 2.31(d) for the `Nachos` matrix. The experiments were performed on the `PlaFRIM` platform. The configuration for each matrix is given in Table 2.9.

## 2.7 Comparison of MaPHyS with the PDSLin hybrid-solver

In this section we present some performance comparison between MaPHyS and PDSLin [149] a hybrid solver developed at LBNL that is closely related to SuperLU_DIST [123]. In order to analyse and comment on these results we need to further detail the governing ideas and main steps implemented in PDSLin. PDSLin roots are in sparse direct techniques while MaPHyS come from classical domain decomposition ideas adapted to an algebraic formalism. The two approaches solve iteratively a Schur complement system that arises from the following block reordering of the linear system to be solved

$$\begin{pmatrix} \mathcal{A}_{\mathcal{I}\mathcal{I}} & \mathcal{A}_{\mathcal{I}\Gamma} \\ \mathcal{A}_{\Gamma\mathcal{I}} & \mathcal{A}_{\Gamma\Gamma} \end{pmatrix} \begin{pmatrix} x_{\mathcal{I}} \\ x_{\Gamma} \end{pmatrix} = \begin{pmatrix} b_{\mathcal{I}} \\ b_{\Gamma} \end{pmatrix},$$

where $x_\Gamma$ contains all unknowns associated with subdomain interfaces and $x_{\mathcal{I}}$ contains the remaining unknowns associated with subdomain interiors. The unknowns $x_{\mathcal{I}}$ are computed using a sparse direct solver while $x_\Gamma$ is calculated using an iterative scheme. The matrix $\mathcal{A}_{\mathcal{I}\mathcal{I}}$ is block diagonal where each block $\mathcal{A}_{\mathcal{I}_\ell\mathcal{I}_\ell}$ corresponds to a subdomain. The coefficient matrix can then be written

$$\left(\begin{array}{cccc|c} \mathcal{A}_{\mathcal{I}_1\mathcal{I}_1} & & & & \mathcal{A}_{\mathcal{I}_1\Gamma} \\ & \mathcal{A}_{\mathcal{I}_2\mathcal{I}_2} & & & \mathcal{A}_{\mathcal{I}_1\Gamma} \\ & & \ddots & & \vdots \\ & & & \mathcal{A}_{\mathcal{I}_\mathcal{N}\mathcal{I}_\mathcal{N}} & \mathcal{A}_{\mathcal{I}_\mathcal{N}\Gamma} \\ \hline \mathcal{A}_{\Gamma\mathcal{I}_1} & \mathcal{A}_{\Gamma\mathcal{I}_2} & \dots & \mathcal{A}_{\Gamma\mathcal{I}_\mathcal{N}} & \mathcal{A}_{\Gamma\Gamma} \end{array}\right).$$

Given a LU decomposition of $\mathcal{A}_{\mathcal{I}_\ell\mathcal{I}_\ell} = L_\ell U_\ell$, the Schur complement matrix $\mathcal{S}$ can be computed as follows

$$\begin{aligned} \mathcal{S} &= \mathcal{A}_{\Gamma\Gamma} - \sum_{\ell=1}^{\mathcal{N}} \mathcal{A}_{\Gamma\mathcal{I}_\ell}\mathcal{A}^{-1}_{\mathcal{I}_\ell\mathcal{I}_\ell}\mathcal{A}_{\mathcal{I}_\ell\Gamma} \\ &= \mathcal{A}_{\Gamma\Gamma} - \sum_{\ell=1}^{\mathcal{N}} (U_\ell^{-T}\mathcal{A}^T_{\Gamma\mathcal{I}_\ell})^T(L_\ell^{-1}\mathcal{A}_{\mathcal{I}_\ell\Gamma}) \\ &= \mathcal{A}_{\Gamma\Gamma} - \sum_{\ell=1}^{\mathcal{N}} W_\ell G_\ell, \end{aligned}$$

where $W_\ell$ ($G_\ell$) are the off-diagonal blocks associated with the interface rows (resp. interface columns) of the LU factorization of $\mathcal{A}$. A large amount of fill may occur in $W_\ell$ and $G_\ell$. To reduce the memory and computational costs, their approximations $\tilde{W}_\ell$ and $\tilde{G}_\ell$ are computed by discarding nonzeros with magnitudes less than a prescribed drop tolerance, and an approximate update matrix $\tilde{T}_\ell = \tilde{G}_\ell\tilde{W}_\ell$ is computed to form $\hat{\mathcal{S}} = \mathcal{A}_{\Gamma\Gamma} - \sum_{\ell=1}^{\mathcal{N}} \tilde{T}_\ell$. To further reduce the costs, small nonzeros are discarded from $\hat{\mathcal{S}}$ to form its approximation $\tilde{\mathcal{S}}$. $\tilde{\mathcal{S}}$ is eventually factorized to form the implicit preconditioner of PDSLin. A summary of the PDSLin implementation is as follows

1. Concurrent parallel SuperLU_DIST instances compute $\mathcal{A}_{\mathcal{I}_\ell\mathcal{I}_\ell} = L_\ell U_\ell$, each SuperLU_DIST instance is run by $n_{g\ell}$ MPI processes.

2. Sparsification of $W_\ell$ and $G_\ell$, to form $\tilde{W}_\ell$ and $\tilde{G}_\ell$ possibly using the thresholds $\tau_G$ and $\tau_W$.

3. Parallel computation of $\hat{\mathcal{S}} = \mathcal{A}_{\Gamma\Gamma} - \sum_{\ell=1}^{\mathcal{N}} \tilde{T}_\ell$ and its sparsification based on a threshold $\tau_{\hat{\mathcal{S}}}$ to compute $\tilde{\mathcal{S}}$.

4. Parallel factorization of $\tilde{\mathcal{S}}$ using an additional instance of SuperLU_DIST to form the implicit preconditioner of PDSLin.

5. Parallel iterative solution using a Krylov subspace method, GMRES for our experiments, using $n_{g_s}$ MPI processes.

The sophisticated 2-level MPI implementation of PDSLin is described in details in [149], we refer to this article for a more exhaustive presentation.

All the parallel experiments were conducted on the `Hopper` platform using the same partitionning tool. Both solvers used the same stopping criterion given in Equation 2.14, $\varepsilon_b = 10^{-10}$. We compare the two solvers varying the number of cores for the matrices `Matrix211` and `Tdr455k`. For a given number of cores, we select the set of control parameters for the two solvers that minimize the parallel time to solution. We highlight the fact that MaPHyS uses all the cores for each of its computational steps, while PDSLin only exploits all of them in its first algorithmic step that is the factorizations of the local subdmains matrices and calculation of $\tilde{\mathcal{S}}$. In PDSLin only a subset of all the cores are used for the iterative solution of the Schur complement system. While the optimal number of subdomains might differ from one solver to the other, we made sure (for the sake of comparison) that the unknowns associated with the smallest Schur complement are part of the unknowns of the largest Schur complement (i.e., the interface vertices of the smallest partition are a subset of the vertices of the partition with the largest number of subdomains). These "optimal" sets of control parameters are listed in Table 2.10 for the two solvers. For MaPHyS we give the number of threads per subdomain and we only used the dense variant of the local Schur complement for the preconditioner. For PDSLin, we display the various thresholds $(\tau_G, \tau_W, \tau_{\hat{\mathcal{S}}})$ as well as the number of MPI-processes in the communicators generated for the factorization of the local subdomains matrices, $n_{g_\ell}$; and for the iterative solution, $n_{g_S}$. As it can be seen, borrowing to the root principles of the two solvers, PDSLin performs the best with fewer subdomains than MaPHyS. For larger number of subdomains, the parallel calculation of the implicit preconditioner in PDSLin based on a global approximation of the Schur complement becomes expensive although its numerical scalability is still very effective.

Varying the number of cores from 192 to 1,536 for the solution of the `Matrix211` problem, we display bar charts in Figure 2.32 the parallel elapsed time for MaPHyS (left bar) and for PDSLin (right bar). Each bar is decomposed in three parts, the blue one represents the time for the *factorization of the interiors* $\mathcal{A}_{\mathcal{I}_\ell\mathcal{I}_\ell}$, the green one corresponds to the time for the *setup of the preconditioner* preconditionner (assembly and factorization) and the red part is the time spent in the iterative solution of the Schur complement system plus the

| | | | Matrix211 | | | | | Tdr455k | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| #cores | | | 96 | 192 | 384 | 768 | 1536 | 192 | 384 | 768 | 1536 |
| #subdomains | MaPHyS | | 16 | 16 | 16 | 32 | 64 | 64 | 128 | 256 | 256 |
| | PDSLin | | 8 | 8 | 8 | 8 | 16 | 32 | 16 | 16 | 16 |
| # cores | MaPHyS threads | | 6 | 12 | 24 | 24 | 24 | 3 | 3 | 3 | 6 |
| | PDSLin | $n_{g_\ell}$ | 12 | 24 | 48 | 96 | 96 | 6 | 24 | 48 | 96 |
| | | $n_{g_S}$ | 192 | 192 | 192 | 192 | 192 | 96 | 48 | 48 | 48 |
| preconditioner | MaPHyS | | dense | dense | dense | dense | dense | dense | dense | dense | dense |
| | PDSLin | $\tau_G$ | $10^{-7}$ | $10^{-7}$ | $10^{-7}$ | $10^{-7}$ | $10^{-7}$ | $10^{-5}$ | $10^{-5}$ | $10^{-5}$ | $10^{-5}$ |
| | | $\tau_W$ | dense | dense | dense | dense | dense | $10^{-5}$ | $10^{-5}$ | $10^{-5}$ | $10^{-5}$ |
| | | $\tau_{\hat{S}}$ | $10^{-5}$ | $10^{-5}$ | $10^{-5}$ | $10^{-5}$ | $10^{-5}$ | $10^{-4}$ | $10^{-4}$ | $10^{-4}$ | $10^{-4}$ |
| #iterations | MaPHyS | | 9 | 9 | 9 | 21 | 107 | 160 | 118 | 118 | 118 |
| | PDSLin | | 21 | 21 | 21 | 21 | 35 | 60 | 50 | 50 | 50 |

Table 2.10: For each matrix the values of the governing control parameters for MaPHyS and PDSLin.

back-solve of the interiors (*solve step*). For the two solvers, increasing the number of cores always translates in a smaller time to solution, although the overall speed-ups are not very good. One can see that MaPHyS does perform slightly better than PDSLin. Indeed, although its time spent in the iterative solution phase takes always longer than PDSLin, MaPHyS spends less time to set up the preconditioner and that time is dominant. Because the best performance of MaPHyS are obtained using more sudomains than PDSLin for a given number of cores, the time spent in the factorization of $\mathcal{A}_{\mathcal{I}_\ell \mathcal{I}_\ell}$ is always lower since the subdomains are smaller. Note that in the case of a sequence of multiple righ-hand sides (not studied here), the solution step need to be processed multiple times and would thus become dominant; in that case PDSLin would eventually outperform MaPHyS and the most efficient configuration would be PDSLin using $1,536$ cores.

Similar results are depicted in Figure 2.33 for the `Tdr455k` matrix. While increasing the number of cores translates in smaller time to the solution for PDSLin, the solution time with MaPHyS increases when more than the optimal number of cores, 384, are used. The increase in the solution time is mainly due to the lack of robustness of MaPHyS preconditioner when the number of subdomains is increased; this leads to a significant increase of the iterative part of the solver. If a sequence of linear systems had to be solved, MaPHyS would outperform PDSLin up to 384 cores and PDSLin would become the fastest when using more cores. The most efficient configuration would be MaPHyS on 384 cores for both a single and a sequence of right-hand sides.

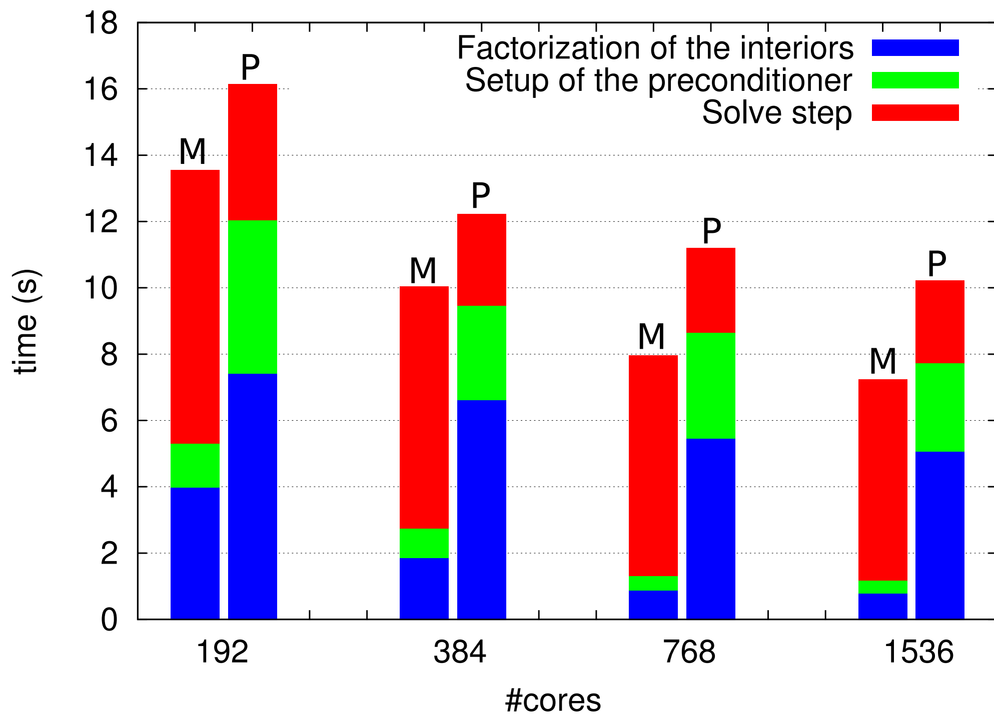



Figure 2.32: Histogram representing the elapsed time for the `Matrix211` matrix for MaPHyS and PDSLin. For each couple of bars, the elapsed time for MaPHyS (M) and PDSLin (P) are given in the first and second bar respectively. For each execution, the configuration is given in Table 2.10.

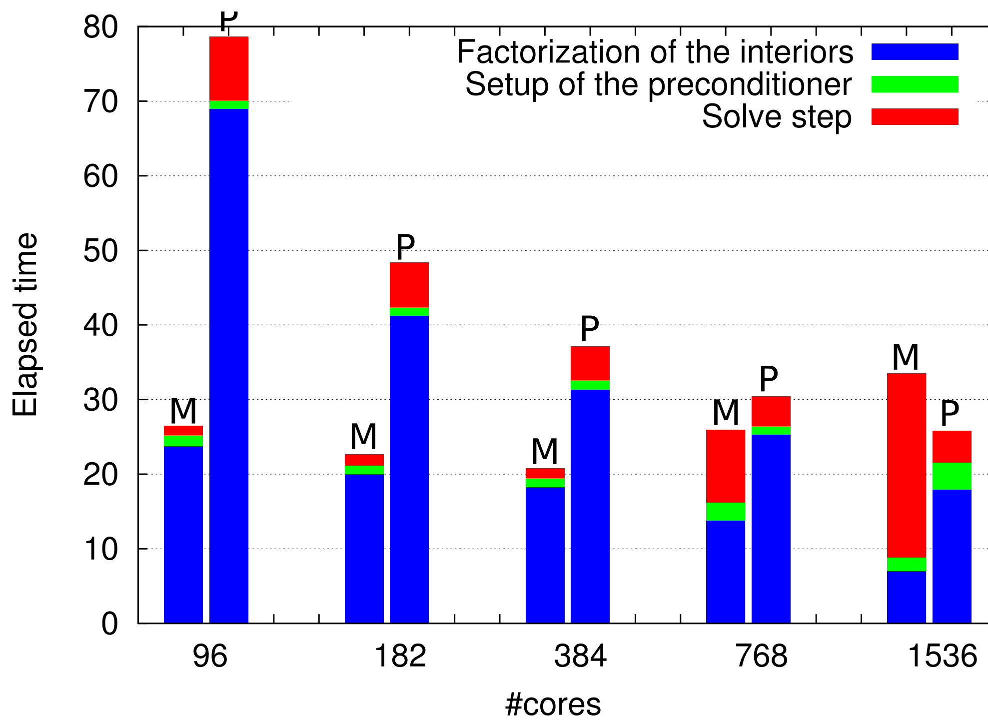



Figure 2.33: Histogram representing the elapsed time for the `Tdr455k` matrix for MaPHyS and PDSLin. For each couple of bars, the elapsed time for MaPHyS (M) and PDSLin (P) are given in the first and second bar respectively. For each execution, the configuration is given in Table 2.10.

## 2.8 Conclusion

In this chapter we have described a 2-level parallel implementation of the hybrid solver MaPHyS and shown how multithreaded libraries, namely Mkl and PaStiX, have been composed to design an efficient parallel implementation. The resulting MPI+thread parallel implementation does match the hierarchical structure of modern multicore parallel platforms. Not only it enables us to better exploit the computer architectures features but it also introduces an additional numerical flexibility to balance the numerical and parallel performances of the MaPHyS solver. Thanks to the new implementation we demonstrated that large computing platforms, up to a few tens of thousand cores, can be exploited to solve 3D linear systems that were not tractable before, neither with the baseline MaPHyS nor a direct sparse solver. For the numerical experiments we considered matrices classically used to benchmark parallel linear solvers as well as matrices arising from 3D geoscience simulations that are of interest for our industrial partner Total. Finally, jointly with colleagues from Lawrence Berkeley National Lab. and the ICL team of the University of Tennessee at Knoxville we conducted a preliminary comparison study with the state of the art parallel hybrid solver, PDSLin that also relies on a 2-level implementation (MPI+MPI). The MaPHyS numerical approach favors the parallelism, through a locally applied preconditioned, while PDSLin favors the numerical robustness, via a global preconditioning technique. These features of the two solvers have been illustrated by our experiments that provide clues on what are the situations where each solver performs better.

In this chapter we have considered classical programming tools, MPI and threads, to design our 2-level parallel hybrid solver. The next chapter is devoted to a more disruptive programming approach based on a task graph description of the algorithm that can be efficiently mapped and scheduled on multicore and manycore platforms by a modern runtime systems. Although the performance of the complete solver is not fully assessed we demonstrate the feasibility of the approach on a complex code and highlight the performance of one of its key component that is the Krylov subspace solver.

# Chapter 3

# Towards task-based hybrid sparse linear solvers

## 3.1 Introduction

In the last decade, the architectural complexity of High Performance Computing (HPC) platforms has strongly increased. In the previous chapter we have proposed a modular, yet relatively low-level, design for exploiting hierarchical supercomputers composed of multiple modern multicore nodes. We have shown that the proposed 2-level MPI+thread approach could successfully achieve an efficient trade-off between the numerical behavior of the method and the usage of the computational resources up to tens of thousands of cores. In spite of high performance it could achieve, such a low-level design suffers from two major bottlenecks for fully exploiting today's platforms.

**The first limitation** is that the design is tightly coupled with the target architecture, *i.e.*, multicore processor case. One solution for relieving this bottleneck would consist in relying on the modular software architecture and select appropriate libraries depending on the target architecture leading to a collection of MPI+X, MPI+Y, ... solution to support X, Y, ... architectures respectively. Alternatively, the architecture may be abstracted relying on task-based programming and delegating the orchestration of the execution of the tasks within computational nodes to a runtime system as illustrated in Figure 3.1. With such MPI+task approach, it is not only elegant to support X, Y, ... architectures in a consistent fashion, but also possible to exploit heterogeneous { X + Y} architectures. We discuss such an alternative in Section 3.3 and we propose a preliminary and partial MPI+task sparse hybrid solver to illustrate our discussion. To do so, we considered the MPI+thread version of MaPHyS and exploited the modular software architecture to substitute the multithreaded Mkl and PaStiX libraries with the task-based Chameleon and prototype task-based version of PaStiX, respectively. Preliminary experiments on a cluster of multicore and on a cluster of heterogeneous nodes support the discussion, showing the feasibility of this approach.

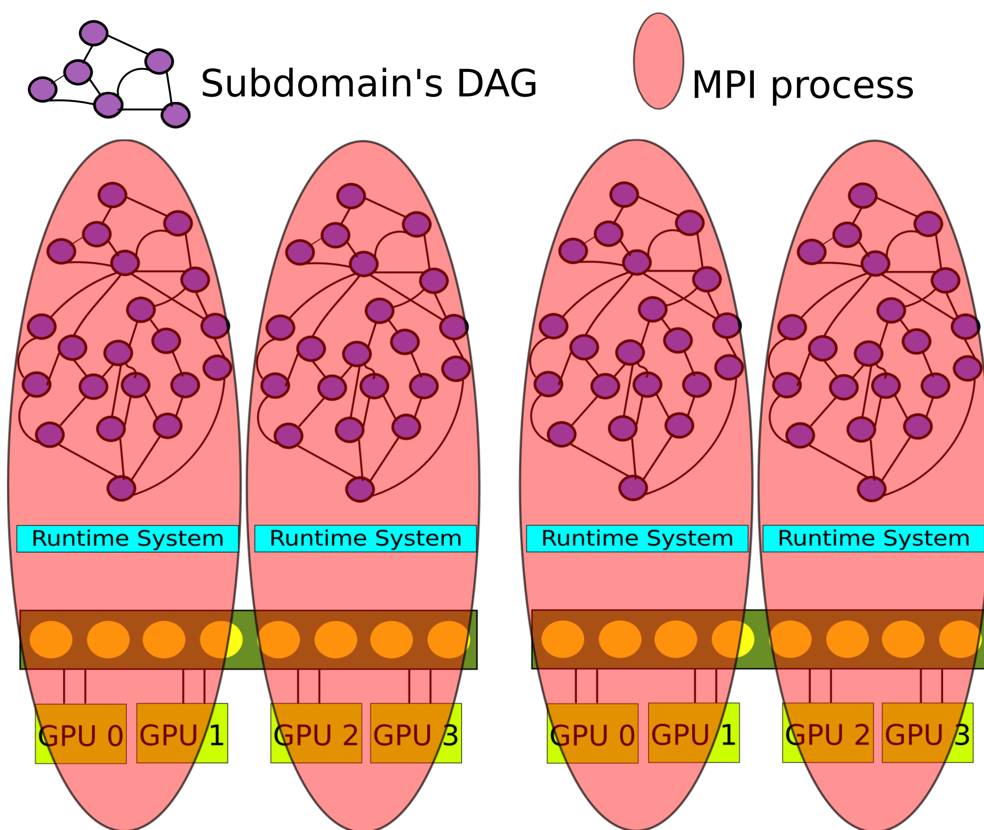



Figure 3.1: MPI + task paradigm applied on two nodes (composed of eight cores and four GPUs each) with four subdomains.

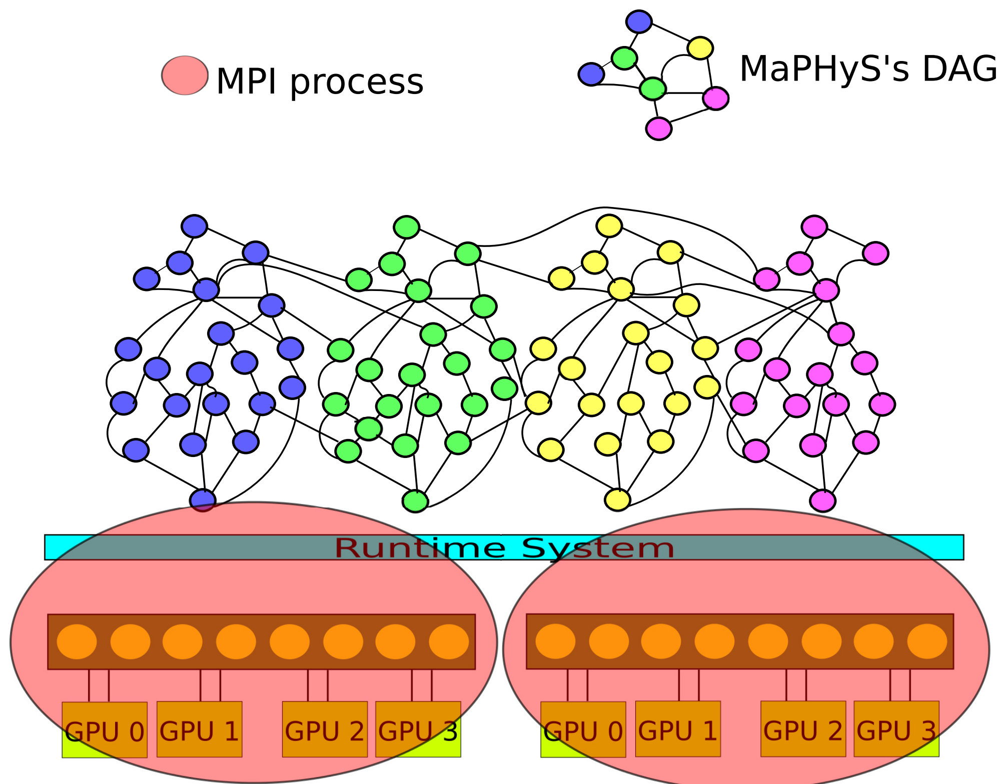



Figure 3.2: Full task-based paradigm applied on two nodes (composed of eight cores and four GPUs each) with four subdomains. Different colors represent different subdomains's DAGs.

**The second limitation** for efficiently exploiting the large spectrum of hardware architectures on top of which are built today's supercomputers is the static mapping imposed by the MPI design of the solver. We showed in the previous chapter that the proposed MPI+thread design could provide a significant flexibility but was still limited to map one subdomain to one or multiple cores of a node. The MPI+task approach proposed in Section 3.3 provides further flexibility but remains limited to map one subdomain to one or multiple resources (a set of CPU cores and GPUs) within a node. One possibility to relieve the burden of such a static mapping would consist in implementing an MPI-based dynamic mapping strategy. However, very few fully-featured libraries implement such a scheme in practice. One of the very few exceptions in linear algebra is the MUMPS sparse direct solver [15]. Furthermore, the scheduling engines that those libraries implement tend to be designed as task queue managers. An alternative approach consists in fully abstracting the MPI scheme of the solver in terms of a DAG of tasks where vertices represent fine grain tasks and edges represent dependencies between them. Once the solver has been designed at such a high-level of abstraction, advanced fine-grain mapping strategies can be implemented as the burden of moving data in the system and ensuring their consistency is delegated to a runtime system. However, such a design prevents from relying on SPMD paradigms proposed in the previous chapter (MPI, MPI+thread) or in Section 3.3 (MPI+task). As a consequence, it requires to fully rewrite the solver in terms of a DAG of tasks as illustrated in Figure 3.2. While there has been lots of progress in that direction for dense (such as the DPLASMA [33] and CHAMELEON [6] task-based libraries, derived from PLASMA [3] and MAGMA [2]) and sparse direct methods (such as the task-based version of PASTIX [95] and qrm_StarPU [9] proposed during the same period as the work done during this thesis), limited work we are aware of has been done to study task-based Krylov methods, the third numerical pillar on top of which hybrid solvers rely. For that reason, we propose and study a task-based CG as a preliminary work towards a fully task-based MAPHYS solver. In this early study presented in Section 3.4, we show that this approach allows one to design advanced 1D and 2D mapping strategies together with fine-grain software pipelining (Section 3.4.5) leading to very competitive performance on a large range (multi-GPUs, multicore, heterogeneous platforms) of hardware architectures (Section 3.4.6) and ultimately combine that with advanced numerical algorithms such as the so-called pipelined CG algorithm from [69] (Section 3.4.7). One of the originality is also that in the case of CG, because of the importance of prefetching, static mapping strategies are more robust than dynamic scheduling strategies [6] usually used by runtime systems in dense and sparse direct methods.

This chapter is organized as follows. Section 3.2 presents further background on task-based runtime systems, related programming models, task-based linear algebra solvers and more broadly GPU-accelerated solvers. Section 3.3 proposes a preliminary and partial prototype MPI+task extension of MAPHYS. Section 3.4 discusses the potential design of a full task-based sparse hybrid solver motivating the implementation of one the key components: a task-based Krylov solver. For this purpose, we propose and study a task-based CG algorithm.

## 3.2 Background

To cope with the complexity of modern architectures, programming paradigms are being revisited. Among others, one major trend consists in writing the algorithms in terms of task graphs and delegating to a runtime system both the management of the data consistency and the orchestration of the actual execution. This paradigm has been intensively studied in the context of dense linear algebra [4, 5, 7, 34, 41, 94, 120, 121] and is now a common utility for related state-of-the-art libraries such as PLASMA [3], MAGMA [2], DPLASMA [33], CHAMELEON [6] and FLAME [142]. Dense linear algebra algorithms were indeed excellent candidates for pioneering in this direction. First, their computational pattern allows one to design very wide task graphs so that many computational units can execute tasks concurrently. Second, the building block operations they rely on, essentially level-three Basic Linear Algebra Subroutines (BLAS), are compute intensive, which makes it possible to split the work in relatively fine grain tasks while fully benefiting from GPU acceleration. As a result, these algorithms are particularly easy to schedule in the sense that state-of-the-art greedy scheduling algorithms may lead to a performance close to the optimum, including on platforms accelerated with multiple GPUs [6].

This trend has then been followed for designing sparse direct methods. The extra challenge in designing task-based sparse direct method is due to indirection and variable granularities of the tasks. The PASTIX team has proposed such an extension of the solver capable of running on the StarPU and PaRSEC runtime systems on cluster of heterogeneous nodes in the context of X. Lacoste thesis [95]. In the meanwhile, the qr_mumps library developed by A. Buttari [40] aims at solving sparse linear least square problems and has been ported on top of those runtime systems in the context of F. Lopez thesis [103]. The sparse hybrid methods considered in this thesis rely on dense and sparse direct methods but also on Krylov solvers. However, Krylov methods are much less regular and compute intensive than direct methods. For these reasons, they are much more challenging to design with a task-based approach and limited work, we are aware of, has been done to study task-based Krylov methods. We propose to address the design of such a missing component in the context of this thesis (Section 3.4). Finally, note that task-based solvers out of the scope of linear algebra have also recently been successfully designed; we do not propose an exhaustive list here, but we may quote a few applications such as Fast Multiple Methods [8, 104], Finite Element Methods [66] and Galerkin Discontinuous Methods [32].

In the rest of this section, we further introduce task-based runtime systems and the programming models they offer in Section 3.2.1. We then propose a brief overview of recent efforts that have been devoted in sparse linear algebra for handling modern architectures in Section 3.2.2. Finally, we present in more details the StarPU runtime system that is used to illustrate our discussion all along this chapter in Section 3.2.3.

### 3.2.1 Task-based runtime systems and related programming models

Computing platform hardware has dramatically evolved ever since the computer science began, always striving to provide new convenient accelerating features. Each new accel-

erating hardware feature inevitably leaves programmers to decide whether to make their application dependent on that feature (and break compatibility) or not (and miss the potential benefit), or even to handle both cases (at the cost of extra management code in the application). This common problem is known as the performance portability issue.

The first purpose of runtime systems is thus to provide *abstraction*. Runtime systems offer a uniform programming interface for a specific subset of hardware (e.g., OpenGL or DirectX are well-established examples of runtime systems dedicated to hardware-accelerated graphics) or low-level software entities (e.g., POSIX-thread implementations). They are designed as thin user-level software layers that complement the basic, general purpose functions provided by the operating system calls. Applications then target these uniform programming interfaces in a portable manner. Low-level, hardware dependent details are hidden inside runtime systems. The adaptation of runtime systems is commonly handled through drivers. The abstraction provided by runtime systems thus enables portability. Abstraction alone is however not enough to provide portability of performance, as it does nothing to leverage low-level-specific features to get increased performance.

Consequently, the second role of runtime systems is to *optimize* abstract application requests by dynamically mapping them onto low-level requests and resources as efficiently as possible. This mapping process makes use of scheduling algorithms and heuristics to decide the best actions to take for a given metric and the application state at a given point in its execution time. This allows applications to readily benefit from available underlying low-level capabilities to their full extent without breaking their portability. Thus, optimization together with abstraction allows runtime systems to offer portability of performance.

In the specific case of parallel work mapping, other approaches have occasionally been adopted instead of using runtimes. Many scientific applications and libraries, including linear system solvers, integrate their own, customized dynamic scheduling algorithms or even resort to static scheduling techniques, either for historical reasons, or to avoid the potential overhead of an extra runtime layer.

However, as multicore processors densify, as cache and memory hierarchies deepen, the resulting increase in complexity now makes the use of work-mapping runtime systems virtually unavoidable. Such work-mapping runtime systems take elementary task descriptions and dependencies as input and are responsible for dynamically scheduling the tasks on available computing units so as to minimize a given cost function (usually the execution time) under some pre-defined set of constraints.

Work-mapping runtime systems themselves are now facing new challenges with the recent move of the high performance community towards the use of specialized accelerating cores together with traditional general-purpose cores. They not only have to decide about the interest (or not) to use some specific hardware features, but also have to decide whether some entire application tasks should rather be performed on an accelerated core or is better left on a standard core.

In the case where specialized cores are located on an expansion card having its own memory (e.g., most existing GPUs), the input data of a task have to be copied from central memory to the card memory before the task can be run. The output results must also be copied back to the central memory once the task computation is complete. The cost of

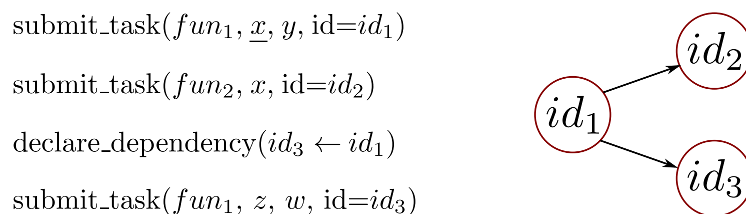



Figure 3.3: Pseudo-code (left) and associated DAG (right). Arguments corresponding to data that are modified by the function are underlined. The $id_1 \to id_3$ dependency is declared explicitly while the $id_1 \to id_2$ dependency is implicitly inferred with respect to the data hazard on $x$.

copying data between central memory and accelerator memory is not negligible. This cost, as well as data dependencies between tasks, must therefore also be taken into account by the scheduling algorithms when deciding whether to offload a given task, to avoid unnecessary data transfers. Transfers should also be done in advance and asynchronously so as to overlap communication with computation.

#### 3.2.1.1 Programming models for task-based applications

Modern task-based runtime systems aim at abstracting the low-level details of the hardware architecture and enhance the portability of the performance of the code designed on top of them. As it is the case in this thesis, in most cases, this abstraction relies on a *DAG of tasks*. In this DAG, vertices represent the tasks to be executed while edges represent the dependencies between them.

While tasks are almost systematically explicitly encoded, runtime systems offer multiple ways to encode the dependencies of the DAG. Each runtime system usually comes with its own API which includes one or multiple ways to encode the dependencies and their exhaustive listing would be out of the scope of this thesis. However, we may consider that there are two main modes for encoding dependencies. The most natural method consists in declaring *explicit dependencies* between tasks. In spite of the simplicity of the concept, this approach may have a limited productivity in practice as some algorithms may have dependencies that are difficult to express. Alternatively, dependencies may be implicitly computed by the runtime system thanks to the *sequential consistency*. In this latter approach, tasks are provided in sequence and the data they operate on are also declared.

We illustrate these two dominant modes of expression of dependencies with a simple example relying on a minimum number of pseudo-instructions. Assume we want to encode the DAG shown in Figure 3.3 (right) relying on an explicit dependency between tasks $id_1$ and $id_3$ and an implicit dependency between tasks $id_1$ and $id_2$. A task can be defined as an instance of a function working on a specific set of data, different tasks possibly being different instances of a same function. For instance, in our example we assume that tasks $id_1$ and $id_3$ are instances of function $fun_1$ while task $id_2$ is an instance of function $fun_2$. While tasks are instantiated with the `submit_task` pseudo-instruction (see Figure 3.3, left), the explicit dependency between tasks $id_1$ and $id_3$ can simply be encoded with a

`declare_dependency` pseudo-instruction (see Figure 3.3, left). On the other hand, implicit dependencies aim letting the runtime system automatically infer dependencies thanks to so-called superscalar analysis [11] which aims at ensuring that the parallelization does not violate dependencies, following the sequential consistency. While CPUs implement such a superscalar analysis on chip at the instruction level [11], runtime systems implement it in a software layer on tasks. Superscalar analysis is performed on tasks and the associated input/output data they operate on. Assume that task $id_1$ operates on data $x$ and $y$ in read/write mode and read mode (calling $fun_1(\underline{x}, y)$ if the arguments corresponding to data which are modified by a function are underlined) respectively, while task $id_2$ operates on data $x$ in read mode ($fun_2(x)$). Because of possible data hazards occurring on $x$ between tasks $id_1$ and $id_2$, the superscalar analysis detects that a dependency is required to respect the sequential consistency.

Another important paradigm for handling dependencies consists of *recursive submission*. Indeed, it may be convenient for the programmer to let tasks trigger other tasks. Sometimes, one may furthermore need the task to be fully completed and cleaned up before triggering other tasks. Runtime systems often support this option through a so-called *call-back* mechanism consisting of a post-processing portion of code executed once the task is completed and cleaned up. Depending on the context, the programmer affinity and portion of the algorithm to encode, different paradigms may be considered as natural and appropriate. For instance, the first implementation of `qrm_starpu` relied on a combination of these four types of dependencies (explicit, implicit, recursive, call-back) [9].

Alternatively, one may rely on a well-defined and more simple programming model in order to design a relatively more simple code, easier to maintain and benefit from properties provided by the model. The *Sequential Task Flow* (STF) programming model consists on fully relying on `sequential consistency` using only implicit dependencies. The STF model, therefore, consists of submitting a sequence of tasks through a non blocking function call that delegates the execution of the task to the runtime system. Upon submission, the runtime system adds the task to the current DAG along with its dependencies which are automatically computed through data dependency analysis [11]. The actual execution of the task is then postponed to the moment when its dependencies are satisfied. As mentioned above, this paradigm is also sometimes referred to as *superscalar* since it mimics the functioning of superscalar processors where instructions are issued sequentially from a single stream but can actually be executed in a different order and, possibly, in parallel depending on their mutual dependencies. Section 3.3 proposes a preliminary and partial prototype MPI+task extension of MaPHyS. Because the proposed approach relies on the modular design of MaPHyS, the programming paradigms used to implement the building block libraries is transparent. However, we show the limits of such a black-box approach and discuss a potential extension to further exploit the fact that both the dense (Chameleon) and sparse direct (PaStiX) solvers follow an STF design to further optimize the overall MPI+STF approach. Section 3.4 discusses the potential design of a full STF task-based sparse hybrid solver motivating the implementation of one the key components: an STF CG solver. We show that the simplicity of the model allows for designing more advanced numerical algorithms such as pipelined CG [69] with a concise yet effective expression.

One challenge in scaling to large scale manycore systems is how to represent extremely large DAGs of tasks in a compact fashion. [55] presented a model, namely the *Parametrized*

*Task Graph* (PTG), which addresses this issue. In this model, tasks are not enumerated but parametrized and dependencies between tasks are explicit. For instance, in the DAG represented in Figure 3.3 (right), tasks $id_1$ and $id_3$ are two instances of the same type of task implementing $fun_1$. This property can be used to encode the DAG in a compact way inducing a lower memory footprint for its representation as well as ensuring limited complexity for parsing it while the problem size grows. For this reason, the memory consumption overhead in the runtime system for representing the DAG can be much lower for the PTG model than for the STF model. In addition with a STF model the DAG has to be completely unrolled whereas with a PTG model the DAG is only partially unfolded during the execution following the task progression. From this point of view, the advantage of the PTG approach over the STF one can be crucial in a distributed memory context because the DAG is pruned on every nodes and only a portion of the DAG is represented on each node. This could considerably reduce the runtime overhead for the management of the DAG. On the other hand, knowing the entire DAG can be useful to compute the schedule of the DAG or give information to the dynamic scheduler by prepossessing the DAG. However, this latter approach is not considered in this thesis and will be addressed in future work.

#### 3.2.1.2 Task-based runtime systems for modern architectures

Many initiatives have emerged in the past years to develop efficient runtime systems for modern heterogeneous platforms. Most of these runtime systems use a task-based paradigm to express concurrency and dependencies by employing a task dependency graph to represent the application to be executed. The main differences between all the approaches are related, to the programming model, to whether or not they manage data movements between computational resources and to which extent they focus on task scheduling.

Some runtime systems have been specifically designed for the development of parallel linear algebra applications. One of these is the TBLAS runtime system [136], which provides a simple interface to create dense linear algebra applications and automates data transfers. TBLAS assumes that programmers should statically map data on the different processing units but it supports heterogeneous data block sizes (i.e., different granularity of computations). The QUARK runtime system [93] was specifically designed for scheduling linear algebra kernels on multi-core architectures. It is characterized by a scheduling algorithm based on work-stealing and by its higher scalability in comparison with other dedicated runtime systems. Finally, the SuperMatrix runtime system [47] follows nearly the same idea as it represents the matrix hierarchically: the matrix is viewed as blocks that serve as units of data where operations over those blocks are treated as units of computation. The implementation transparently queues the required operations, internally tracking dependencies, and then executes the operations utilizing out-of-order execution techniques.

Most of the available runtime systems, however, do not target any specific type of applications and provide a general API. Qilin [106], for example, provides an interface to submit kernels that operate on arrays which are automatically dispatched between the different processing units of an heterogeneous machine. Moreover, Qilin dynamically compiles parallel codes for both CPUs (by relying on the Intel TBB [124] technology) and for GPUs, using CUDA. Another relevant framework is Charm++ [89] which is a parallel variant of the C++

language that provides sophisticated load balancing and a large number of communication optimization mechanisms. Charm++ has been extended to provide support for accelerators such as the Cell processors as well as GPUs [92]. Many runtime systems propose a task-based programming paradigm. Runtime systems like KAAPI/XKAAPI [65] or APC+ [79], Legion [39], Realm [139] offer support for hybrid platforms mixing CPUs and GPUs. Their data management is based on a DSM-like mechanism: each data block is associated with a bitmap that permits to determine whether there is already a copy locally available to a specific processing unit or not. Moreover, task scheduling within KAAPI is based on work-stealing mechanisms or on graph partitioning. The StarSs project is actually an umbrella term that describes both the StarSs language extensions and a collection of runtime systems targeting different types of platforms [21,22]. StarSs provides an annotation-based language which extends C or Fortran applications to offload pieces of computation on the architecture targeted by the underlying runtime system. The PaRSEC [35, 36] (formerly DAGuE) runtime system dynamically schedules tasks within a node using a rather simple strategy based on a locality-aware work-stealing strategy. It was first introduced for linear algebra but was later extended to more generic applications. It takes advantage of the specific shape of the task graphs (in the sense that there are few types of tasks) to represent the task dependency graph in an algebraic fashion. More details on this tool are given in Section 3.2.3. The StarPU runtime system provides a generic interface for developing parallel, task-based applications. It supports multicore architectures equipped with accelerator as well as distributed memory systems. This runtime is capable of transparently handling data and provides a rich panel of features. The details of this runtime system are given in Section 3.2.3. All the above mentioned efforts have contributed to proving the ease of use, the effectiveness and portability of general purpose runtime systems to the point where the OpenMP board has decided to include similar features in the latest OpenMP standard 4.0: the `task` construct was extended with the `depend` clause which enables the OpenMP runtime to automatically detect dependencies among tasks and consequently schedule them. The same OpenMP standard also provides constructs for using accelerator devices.

Whereas task-based runtime systems were mainly research tools in the past years, their recent progress makes them now solid candidates for designing advanced scientific software as they provide programming paradigms that allow the programmer to express concurrency in a simple yet effective way and relieve him from the burden of dealing with low-level architectural details.

The work presented in this thesis relies on the StarPU runtime system. This is mostly due to its large set of features which include full control over the scheduling policy, support for accelerators and transparent handling of data. For this reason, this runtime is described in more details in Section 3.2.3.

### 3.2.2 Sparse linear algebra on modern architectures

Adapting sparse linear algebra solvers to modern heterogeneous systems is an active area of research. From the sparse direct solvers point of view, a lot of efforts have been made to port or adapt existing solvers to exploit GPUs. Among these efforts, we can cite the work proposed in [67, 105, 151]. These approaches mainly target the multifrontal method for LU

or Cholesky factorizations due to its very good data locality properties. The main idea is to treat some parts of the computations (mostly, trailing submatrix updates) entirely on the GPU. Therefore the main originality of these efforts is in the methods and algorithms used to decide whether or not a task can be processed by a GPU. In most cases, this was achieved through a threshold based criterion on the size of the computational tasks. More complex approaches can be found in [125, 132, 153]. These works improve over previous efforts mostly by proposing techniques for aggregating fine grain operations to form large grain tasks which maximize the GPU occupancy (either by grouping basic BLAS operations or by treating a complete subtree as a single task) and pipelining to overlap communications with computations. In a more recent work, [131] extend the SuperLU_Dist package to support Xeon Phi architectures using analogous techniques as in their previous effort [132].

Concerning iterative solvers, the adaptation of existing methods to the new context of GPU enhanced platform was an active area of research in the past decade. An initial step was to focus the SpMV kernel which is the core kernel of sparse iterative methods. In [147] (resp. [145]), an overview is given on the performance of the CSR-based SpMV (resp. the CG algorithm) for a number of modern CPU architectures. Then, with the democratization of GPUs, several studies have focused on the improvement of the performance of the SpMV kernel on GPUs. These efforts have mainly targeted the sparse matrix representation format to improve the memory access pattern/footprint of the kernel. From the format point of view of the matrix representation, several layouts were studied such as the compressed sparse row (CSR) format, the coordinate format (COO) format, the diagonal (DIA) format, the ELLPACK (ELL) format and an hybrid (ELL/COO) format. An exhaustive description of different implementations of the SpMV operation for GPU architectures can be found in [27]. More recently, a new row-grouped CSR format was considered [113] as well as a sliced ELLPACK format [111]. In [27], Bell and Garland propose several methods for efficient sparse matrix-vector multiplications that take into account the structure of the input matrices. They implemented efficient multiplication routines for various sparse matrix layouts including a new hybrid layout (this layout stores part of the matrix using ELLPACK and the remaining elements using the COO format in order to reduce the memory footprint). Their hybrid layout is most suitable for unstructured matrices and delivers in general the best performance for such matrices. This work was followed by many efforts targeting these hybrid representations [53,110]. A model-driven auto-tuning approach was introduced in [53] in order to find the best parameters for the hybrid storage format. Finally, in [80] a new implementation of the SpMV operation, namely segSpMV, was introduced. This new algorithm can enjoy full coalesced memory access compared to existing approaches and supports multiple GPUs in a native way.

From the sparse iterative methods point of view, many efforts have been conducted to adapt the existing algorithms and exploit GPU. The block-ILU preconditioned GMRES method is studied for solving unsymmetric sparse linear systems on the NVIDIA Tesla GPUs in [144]. SpMV kernels on GPUs and the block-ILU preconditioned GMRES method are used in [98] for solving large sparse linear systems in black-oil simulations. More recently, several works have addressed the multi-GPU case where the computational node is enhanced with more than one GPU. In [12], a specific CG method with incomplete Poisson preconditioning is proposed for the Poisson problem on a multi-GPU platform. In [68], the authors present an implementation of the CG algorithm for multi-GPU platforms based on

a fast distributed SpMV kernel using optimized data formats (combining ELL and padded CSR) to allow a good overlapping between communication and computation. In [44], the authors present a CG algorithm running on multiple GPUs using a data parallel approach where the number of non-zeros is balanced among GPUs. The authors have then refined their approach in [45], by using a new partitioning for the matrix based on an hypergraph model to reduce the amount of communications needed by the algorithm. In [143], the authors describe mappings for the SpMV kernels and show how they parallelize the CG algorithm over the GPUs by implementing parallel operations (i.e., kernels running on multiple devices). Moreover, they present results illustrating the fact that reordering the input matrix can improve the performance of the method. Concerning GMRES, it has been studied in the context of multi-GPU platforms [23, 56] where authors mainly identify the operations that have to be performed by GPUs (e.g., SpMV) and the operations that need to be processed by CPUs (reductions for example). More recently, a lot of efforts targeted the reduction of the amount of communications between the GPU devices and the host to improve the performance of the considered iterative solvers. These works either concerned the minimization of the amount of communications by means of sophisticated partitioning [49, 51], or by reformulating the kernels to improve data reusability [16], and finally by relying on the communication avoiding class of algorithms [148, 150]. Finally, one initiative studied the problem of designing high-performance iterative methods for clusters of nodes enhanced with GPU devices [152].

Although all the studies mentioned above were implemented on top of low-level programming interfaces like CUDA [1] or OpenCL [112], we present in Section 3.4 an implementation of iterative methods using a task-based runtime system, namely StarPU, to assess the opportunities provided by task-based programming paradigm in this context.

### 3.2.3 The StarPU task-based runtime system

StarPU is a runtime system developed by the STORM (formerly RUNTIME) team at Inria Bordeaux specifically designed for the parallelization of algorithms on heterogeneous architectures. A complete description of StarPU can be found in [20].

StarPU provides an interface which is extremely convenient for implementing and parallelizing applications or algorithms that can be described as a graph of tasks. Tasks have to be explicitly submitted to the runtime system along with the data they work on and the corresponding data access mode. Through data analysis, StarPU can automatically detect the dependencies among tasks and build the corresponding DAG. Once a task is submitted, the runtime tracks its dependencies and schedules its execution as soon as these are satisfied, taking care of gathering the data on the unit where the task is actually executed. In StarPU the execution is initiated by a *master thread*, running on a CPU, which is commonly in charge of submitting the tasks; the execution of the tasks, instead, is performed by *worker threads* (or simply *workers*) whose number and type (e.g., CPU or GPU) can be chosen by the programmer or by the user at run time. A CPU worker is bound to a CPU core whereas a GPU worker is bound to a GPU and a CPU core which is used to drive the work of the GPU. Note that nothing prevents worker threads from submitting tasks although this does not comply with the Sequential Task Flow model. Because StarPU has

full control over a task and the associated data, these have to be declared to the runtime prior to the task submission. A task type can be declared through a *codelet* which specifies, among other things, the name of the task type, the units where it can be executed (e.g., CPU and/or GPU), the corresponding implementations (one for each type of unit) and the number of input data. Data, instead, is declared through a specific function call where the programmer informs StarPU about the location of the data (i.e., the data pointer) and about its properties such as the size, the rank, the leading dimension. Upon execution of this function call, StarPU returns a *handle*; once declared, the data is not meant to be directly accessed by the programmer anymore but only through the handle and the associated methods. A task can roughly be defined as an instance of a task type coupled with a set of handles which represent the data used by the task itself.

Among the advanced features provided by the StarPU runtime system, in the work described in the rest of this chapter, we will use the following ones:

- StarPU provides a framework for developing task scheduling policies in a portable way. The implementation of a scheduler consists in creating a task container and defining the code that will be triggered each time a new task gets ready to be executed (`push`) or each time a worker thread has to select the next task to be executed (`pop`).The implementation of each queue may follow various strategies (e.g., FIFO or LIFO) and sophisticated policies such as work-stealing may be implemented. StarPU comes with a number of predefined scheduling policies suited for different types of architectures and workloads. We use the built-in work-stealing policy provided by StarPU in Section 3.3 for running an MPI+STF prototype extension of MaPHyS. However, because Krylov methods are extremely irregular and memory-bound, dynamic schedulers tend to take bad decisions when orchestrating their computation. As a consequence we propose a static scheduling policy in Section 3.4.3 in order to rule the execution of the proposed task-based CG algorithm studied in Section 3.4.

- StarPU offers several advanced utilities for data management. In this study, we specifically consider the data partitioning and data unpartitioning operations. Data partitioning divides a piece of data in several parts. Data unpartitioning is the opposite operation; it assembles several parts of a partitioned data in the original form. Those operations are the counterpart of the scatter/gather operations in the SPMD programming model used by MPI and similar instructions existed also on vector computers a few years ago. Both these operations are blocking operations. That is, they are control instructions for the task flow construction, which prevent the user from submitting any new task as long as the corresponding tasks, partitioning or unpartitioning, have been processed. We study how to relieve such synchronizations in Section 3.4.5.2.

- If a task is assigned to a worker sufficiently ahead of its execution, the data it needs can be automatically prefetched.

- In StarPU, it is possible to associate a *callback* function with a task. This is just a function which is executed upon termination of the task itself. We will use this feature in Section 3.4.5.3 to help the runtime system in performing data prefetching as soon as possible based on the knowledge of the mapping at the application level.

- StarPU defines several built-in data types often referred as *primitive data types* in programming languages. There is also an option for extending the set of primitives with a user defined data type. We use this feature for designing advanced packing strategies in Section 3.4.5.3.

- StarPU also implements multiple policies for transferring data between GPUs. The GPU-CPU-GPU communication mechanism consists of moving first the data to the main memory before transferring it back to the destination GPU. On the contrary the direct GPU-GPU mechanism directly transfer the data from the source GPU to the destination GPU. We discuss in Section 3.4.5 how to best benefit from this feature depending on the packing scheme used.

- StarPU has several facilities which allow for detailed performance profiling of an application. For example, it can generate execution traces containing accurate timings of all the executed tasks, of data transfers, .... We rely on it for performing a detailed performance analysis in Section 3.4.6.

## 3.3 Prototype design of an MPI+task extension of MaPHyS

In the previous chapter we have proposed a modular yet relatively low-level design for exploiting hierarchical supercomputer composed of multiple modern multicore nodes. The proposed 2-level MPI+thread approach could successfully achieve an efficient trade-off between the numerical behavior of the method and the usage of the computational resources up to tens of thousands of cores. In this section, we propose to substitute the MPI+thread paradigm with an MPI+task approach. As illustrated in Figure 3.1, this latter approach aims at abstracting the hardware architecture relying on task-based programming and delegating the orchestration of the task within computational nodes to a runtime system.

The goal of this very preliminary study is to show the feasibility of the approach. To illustrate our discussion, we have designed a partial MPI+task sparse hybrid solver. We restricted our scope to the Symmetric Positive Definite (SPD) case. To do so, we considered the MPI+thread version of MaPHyS and exploited the modular software architecture to substitute the multithreaded Mkl and PaStiX libraries with the task-based dense Cholesky solver from Chameleon [6] and the prototype task-based version of PaStiX from X. Lacoste thesis [95], respectively. Following MaPHyS principles (and MPI+thread design, see Section 2.3.2), we have enhanced the task-based version of PaStiX in order to retrieve a Schur complement. Note that in this prototype MPI+task version of MaPHyS, other operations involved in the iterative solution step such as level-one BLAS and matrix-vector product have not been changed (these considerations are further studied in Section 3.4 in the case of a full task-based paradigm).

The tests presented in this section were performed on the `PlaFRIM 2` platform, more precisely on the *sirocco* nodes. These nodes are composed of two Dodeca-core Haswell Intel Xeon E5-2680, for a total of 24 cores per node, and 128 GB of RAM memory. Each node is equipped with 4 Nvidia K40-M GPUs, each one having 12 GB of RAM. We consider

the SPD Audi_kw matrix presented earlier in Table 2.1 to illustrate the behavior of the proposed prototype solver. Both Chameleon and PaStiX rely on the version 1.1 of the StarPU runtime system described in Section 3.2.3.

Figures 3.4 and 3.5 show the traces obtained on one node of the platform, using only CPU cores or both GPUs and CPU cores, respectively. In both cases, the matrix has been decomposed in four subdomains. Each subdomain is associated to one MPI process in charge of a subset of six CPU cores, or six CPU cores and one GPU, respectively. The runtime system orchestrates the execution of the tasks on the different processing units. The traces represent the execution on one particular subdomain. In the heterogeneous case, each GPU has a CPU core dedicated to handle it (see Section 3.2.3). The resulting

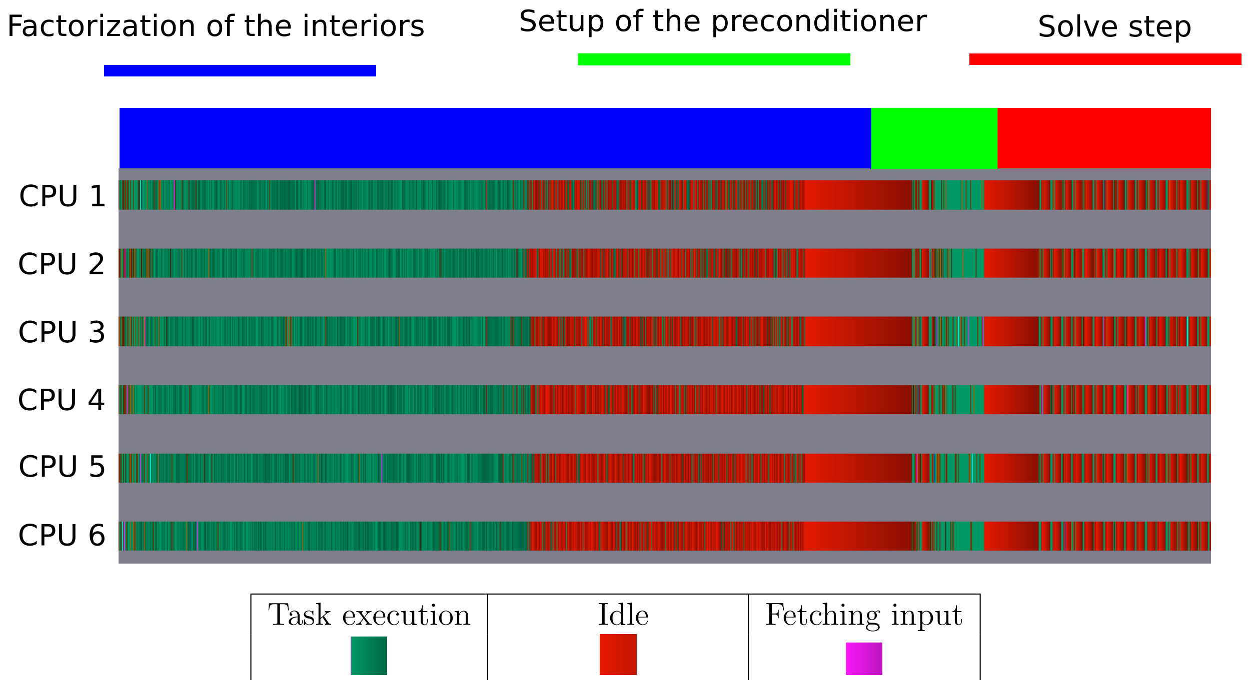



Figure 3.4: Multicore execution trace associated to one subdomain of the MPI+task MaPHyS prototype processing the Audi_kw matrix. Four subdomains (hence four processes) are used in total and a dense preconditioner is applied.

traces show the versatility of the approach. The processing units are abstracted and the same code may be executed indistinguishably on the homogeneous or on the heterogeneous cases. Table 3.1 shows that the resulting timings allow for accelerating all three numerical steps with the use of one GPU per subdomain in spite of the very preliminary design. The *setup of the preconditioner* benefits from the highest acceleration as it mostly consists of a dense factorization. The *factorization of the interiors* has a limited (but not negligible) acceleration because PaStiX internal kernel has not been tuned for the Nvidia K40-M GPU.

Although this prototype is working properly and showed the feasibility of the proposed approach, designing a solid MPI+task version of MaPHyS would require further work. First of all, the proposed approach still follows a bulk-synchronous parallelism [141] (also sometimes designated as fork-join approach) pattern. Indeed, the calls to PaStiX and

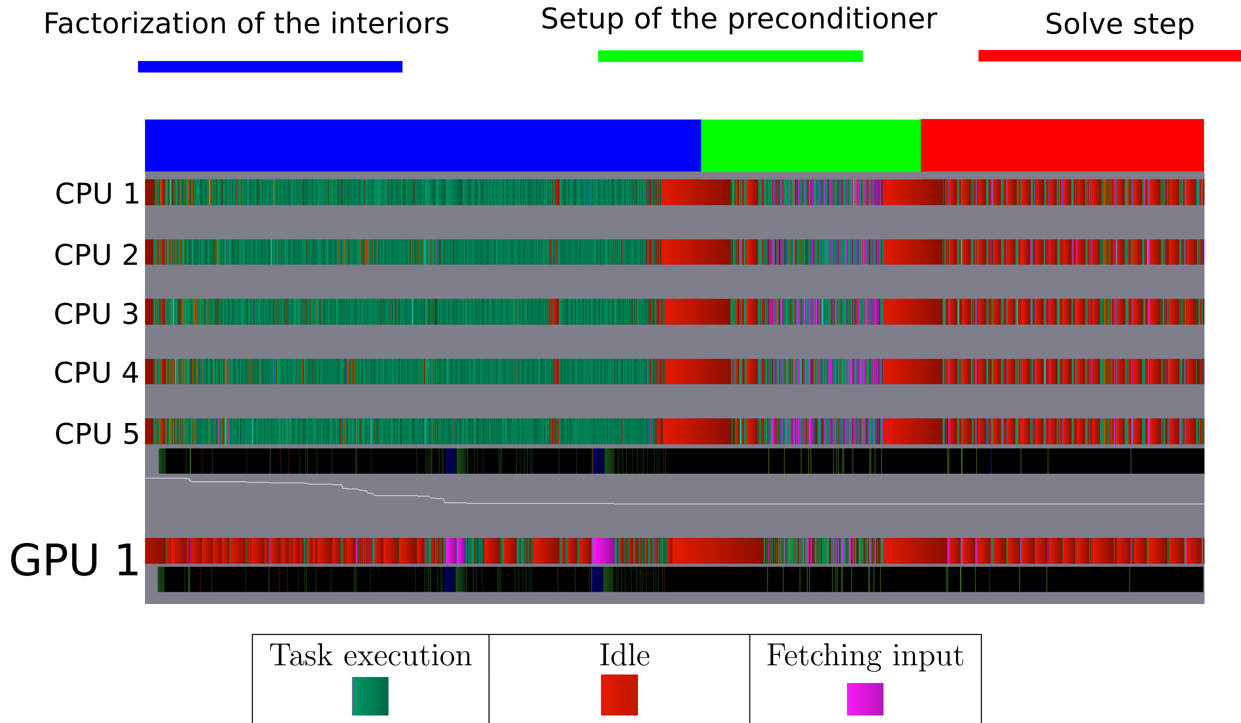



Figure 3.5: Heterogeneous execution trace associated to one subdomain of the MPI+task MaPHyS prototype processing the `Audi_kw` matrix. Four subdomains (hence four processes) are used in total and a dense preconditioner is applied.

| | | Multicore case | Heterogeneous case |
|---|---|---|---|
| *Factorization of the interiors* | min | 19.6 | 23.3 |
| | avg | 37.2 | 31.7 |
| | max | 50.8 | 38.2 |
| *Setup of the preconditioner* | min | 4.80 | 1.10 |
| | avg | 7.02 | 3.63 |
| | max | 9.81 | 7.37 |
| *Solve step* | min | 13.1 | 11.8 |
| | avg | 13.2 | 11.8 |
| | max | 13.2 | 11.8 |

Table 3.1: Minimum, average and maximum time per subdomain for the MPI+task MaPHyS prototype for the multicore case (Figure 3.4) and the heterogeneous case (Figure 3.5) processing the `Audi_kw` matrix. Four subdomains (hence four processes) are used in total and a dense preconditioner is applied.

Chameleon, yet local to each subdomain, induce costly preprocessing. On the one hand, PaStiX need to perform a reordering of the variables to limit fill-in and a symbolic factorization (see Section 1.2.2). These steps are sequential in the present prototype. Although there exist parallel implementations of these steps, they are known to have a very limited parallel efficiency. To overcome the subsequent synchronizations, it would therefore be necessary to overlap these symbolic preprocessing steps with other numerical operations. On the other hand, following Plasma design, Chameleon first decomposes the dense matrix in tiles, which is also a synchronizing operation. As for Plasma, there exists an advanced interface allowing for tackling matrices already decomposed into tiles. Using this interface would certainly alleviate the bottleneck occurring within the *setup of the preconditioner* when calling the dense solver.

Other operations involved in the iterative solution step such as level-one BLAS and matrix-vector product could be implemented with a task-based approach. In the case of a dense preconditioner, these operations could also be implemented by calling BLAS operations implemented in Chameleon. However, in the present state, without using the advanced interface discussed above, the synchronizations would occur multiple times per iteration. We propose a full task-based CG solver in Section 3.4 and discuss in details how synchronization points can be alleviated.

To completely alleviate the synchronizations between the different sequences into which MaPHyS is decomposed, it would be necessary to further overlap communications with computations. This could be performed with a clever usage of asynchronous MPI calls. This approach is relatively difficult to implement and has been applied to overlap main stages in MaPHyS, both in the MPI+thread version from Chapter 2 and in the MPI+task prototype discussed in this section. However, relying on this paradigm for performing fine-grain overlapping would be extremely challenging and certainly result in a code very complex to maintain. Alternatively, the MPI calls can be appended to the task flow. Doing so, the task-based runtime system can dynamically decide when to perform the actual MPI call to the MPI layer and interleave them with fine-grain computational tasks. Modern runtime systems such as StarPU provide such an opportunity. However, even in that case, the task-flow would still have to be designed accordingly to the mapping between tasks and processes. On the contrary, the approach proposed in Section 3.4 allows for fully abstracting the hardware architecture and makes the mapping issues practically orthogonal to the design of the task flow.

## 3.4 Towards a full task-based version of MaPHyS: case study with the CG algorithm

In the previous section, we have discussed the possible design of an MPI+task extension of MaPHyS and proposed a preliminary prototype implementation following this paradigm. Apart from the discussed limits due to the fact that the proposed prototype is preliminary, we highlighted that the proposed MPI+task paradigm still had an important limitation for exploiting efficiently the large spectrum of hardware architectures on top of which are built today's supercomputers. Indeed, although more flexible for exploiting versatile processing

units than the approach proposed in Chapter 2, the MPI+task approach proposed in Section 3.3 remains limited to map one subdomain to one or multiple resources (a set of CPU cores and GPUs) within a node. We investigate here an alternative approach consisting in fully abstracting the MPI scheme of the solver within the task graph. As mentioned earlier, there has been lots of progress in that direction for dense (such as the DPLASMA [33] and CHAMELEON [6] task-based libraries) and sparse direct methods (such as the task-based version of PASTIX [95] and qrm_StarPU [9]), limited work we are aware of has been done to study task-based Krylov methods. Nonetheless, as discussed in Section 3.3, this is the part of the hybrid solver that requires the most effort to be ported in terms of a task-based method because of the potential recurrent synchronizations occurring at each iteration. For that reason, we propose and study a task-based CG as a preliminary work towards a fully task-based MAPHYS solver. Here we focus on a efficient pipelining of the CG algorithm on one heterogeneous CPU+GPU node. We present detailed performance analysis for the multi-GPU, multicore and the hybrid CPU+GPU cases when one node is used of our optimized task-based version of the CG algorithm.

We propose a task-based expression of CG in Section 3.4.1. We then present the experimental set up in Section 3.4.2. Although the considered platform is composed of both GPU accelerators and CPU cores, we motivate and illustrate the design of an advanced task-based formulation with the example of a multi-GPU architecture as the difficulties are emphasized on that type of architecture. We indeed raise the problems of scheduling (Section 3.4.3) and choosing an appropriate data and task granularity (Section 3.4.4) when operating on GPUs. We then address the problem of pipelining efficiently those operations in Section 3.4.5. Although only illustrated in the multi-GPU case in those sections, we show that the proposed algorithms lead to very competitive performance on all the target range (multi-GPUs, multicore, heterogeneous platforms) of hardware architectures (Section 3.4.6). Finally, in Section 3.4.7, we combine the proposed task-based CG design with an advanced formulation of CG that implements the pipeling idea at the numerical level [69].

### 3.4.1 Baseline STF conjugate gradient algorithm

In this section, we present a first task-based expression of the CG algorithm whose pseudo-code is given in Algorithm 8. This algorithm can be divided in two phases, the initialization phase (lines 1-5) and the main loop (lines 6-16). The initialization phase being executed only once, we only focus on an iteration occurring in the main loop in this study.

Three types of operations are used in an iteration of the algorithm: SpMV (line 7), scalar operations (lines 9, 13, 14) and level-one BLAS operations (lines 8, 10, 11, 12, 15). In particular three different level-one BLAS operations are used: scalar product (*dot*, lines 8 and 12), linear combination of vectors (*axpy*, lines 10, 11 and 15) and scaling of a vector by a scalar (*scal*, line 15). The *scal* kernel at line 15 is used in combination with an *axpy*. Indeed, in terms of BLAS, the operation $p \leftarrow r + \beta p$ consists of two successive operations: $p \leftarrow \beta p$ (*scal*) and then $p \leftarrow r + p$ (*axpy*). In our implementation, the combination of these level-one BLAS operations represents a single task called *scal-axpy*. The key operation in an iteration is the *SpMV* (line 7) and its efficiency is thus critical for the performance of

the whole algorithm.

**Algorithm 8** Pseudo-Code of Conjugate Gradient algorithm.

1: $r \leftarrow b$
2: $r \leftarrow r - Ax$
3: $p \leftarrow r$
4: $\delta_{new} \leftarrow dot(r, r)$
5: $\delta_{old} \leftarrow \delta_{new}$
6: **for** $j = 0, 1, ...,$ until $\frac{\|b-Ax\|}{\|b\|} \leq eps$ **do**
7: $q \leftarrow Ap$ /* SpMV */
8: $\alpha \leftarrow dot(p, q)$ /* BLAS-1 */ (dot)
9: $\alpha \leftarrow \delta_{new}/\alpha$ /* scalar operation */
10: $x \leftarrow x + \alpha p$ /* BLAS-1 */ (axpy)
11: $r \leftarrow r - \alpha q$ /* BLAS-1 */ (axpy)
12: $\delta_{new} \leftarrow dot(r, r)$ /* BLAS-1 */ (dot)
13: $\beta \leftarrow \delta_{new}/\delta_{old}$ /* scalar operation */
14: $\delta_{old} \leftarrow \delta_{new}$ /* scalar operation */
15: $p \leftarrow r + \beta p$ /* BLAS-1 */ (scal-axpy)
16: **end for**

According to our STF programming paradigm (see Section 3.2.1 and Section 3.2.3 ), data need to be decomposed in order to provide opportunities for executing concurrent tasks. We consider a 1D decomposition of the matrix, dividing the matrix in several block-rows. The number of non-zero values per block-rows is balanced and the rest of the vectors follows the same decomposition as illustrated in Figure 3.6.

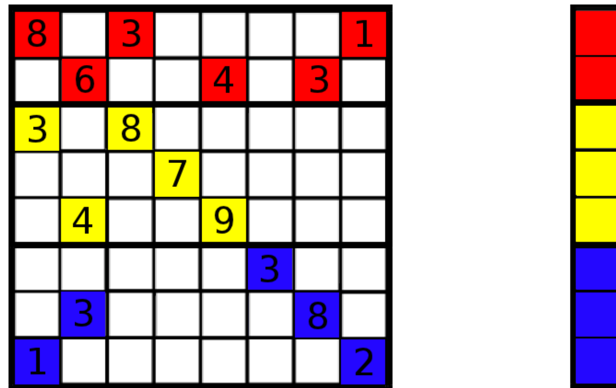



Figure 3.6: Matrix and vector decomposition. 1D decomposition is applied to the matrix balancing the number of non-zero values per block-rows (left). The rest of the vectors follows the same decomposition (right).

After decomposing the data, tasks that operate on those data can be defined. The tasks derived from the main loop of Algorithm 8 are shown in Figure 3.7, when the matrix is divided in six block-rows. Each task is represented by a box, named after the operation executed in that task, and edges represent the dependencies between tasks. Let us examine in more details the task flow in Figure 3.7. The first instruction executed in the main loop of Algorithm 8 is the *SpMV*. *SpMV* performs the operation $q \leftarrow Ap$ where A is a

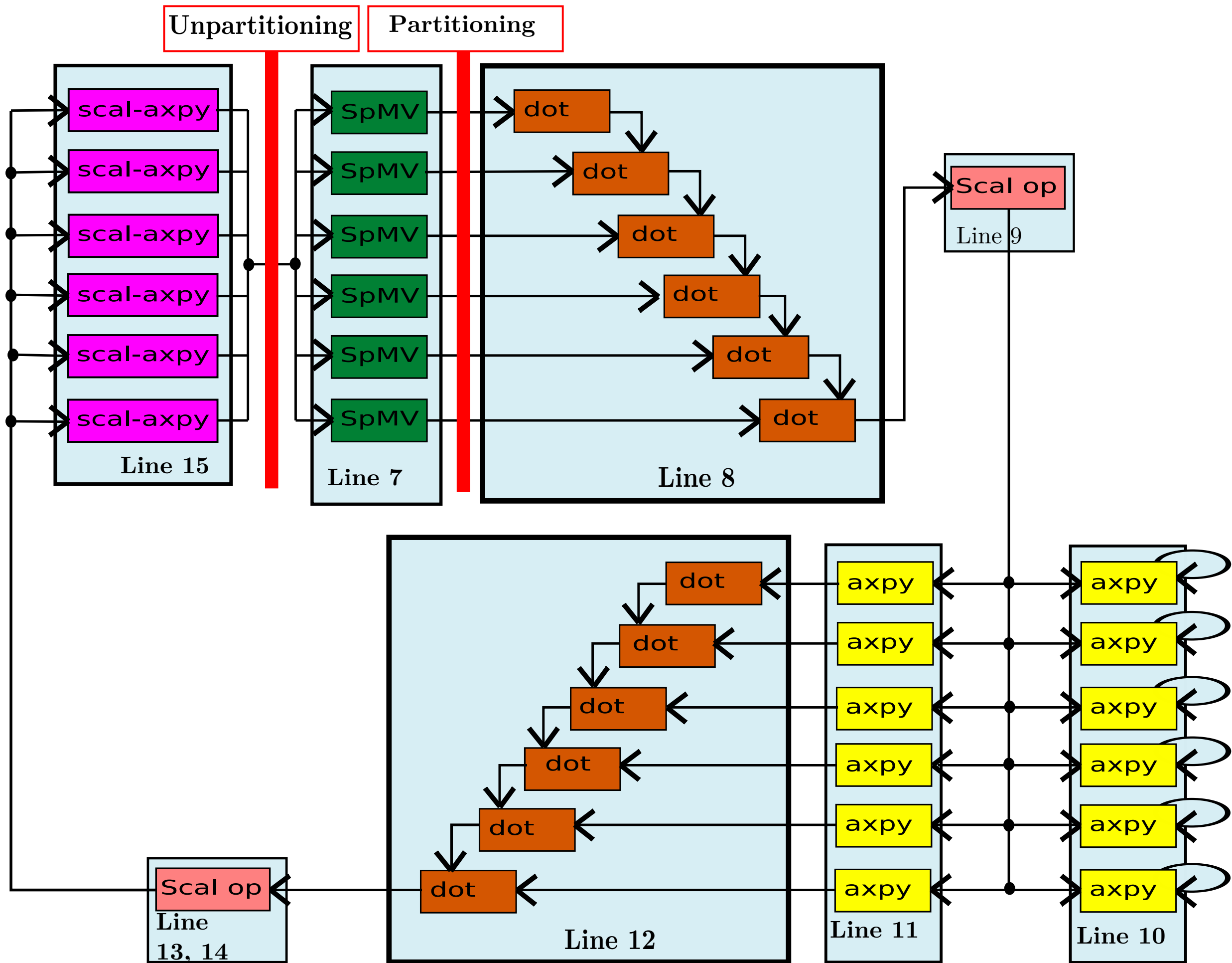



Figure 3.7: Task flow of the main loop of CG Algorithm (lines 7-15 in Algorithm 8). The matrix is divided in six block-rows. Vertices of the graph represent tasks and edges represent dependencies between them. The red vertical bars between the $scal - axpy$ and $SpMV$ tasks and between the $SpMV$ tasks and the $dot$ tasks represent the partitioning and the unpartitioning of the vector $p$, respectively.

sparse matrix and $q$ and $p$ are dense vectors. When a 1D decomposition is applied to the matrix, dividing it in six parts implies that six tasks are submitted for this operation (the green tasks in Figure 3.7): $q_i \leftarrow A_i p, i \in [1, 6]$, one for each block-row $A_i$ of the matrix. For these tasks, a copy of the whole vector $p$ is needed (vector $p$ is unpartitioned). But in order to extract parallelism of other level-one BLAS operations where vector $p$ is used (lines 8 and 15 in Algorithm 8), it needs to be partitioned. As discussed in Section 3.2.3, the partitioning operation is a blocking call; it thus represents a synchronization point in this task flow (represented with the red vertical bar after *SpMV* tasks in Figure 3.7). Once vector $p$ is partitioned, the tasks corresponding to the scalar product (line 8 in Algorithm 8) can then be submitted. Both vectors $p$ and $q$ are partitioned in six parts $(p_i, q_i, i \in [1, 6])$, so six tasks are again submitted. Each *dot* operation accesses $\alpha$ in read-write mode, which induces a serialization of the operation, as shown in Figure 3.7 with the dependencies between the successive *dot* tasks. This sequence thus introduces new synchronizations in the task flow. The final value of $\alpha$ for the current iteration is obtained after the scalar operation $\alpha \leftarrow \delta_{new}/\alpha$ (line 9). The twelve *axpy* tasks (six at line 10 and six at line 11) can then all be executed in parallel (see Figure 3.7 again). Another *dot* operation is then performed (line 12) and induces other serializations (as it was the case for the *dot* at line 8). After the scalar operations at lines 13 and 14 in Algorithm 8, the last operation of the loop can be executed. Similarly to the *axpy* operations, this operation is completely parallel. After this last operation, the new value of vector $p$ is obtained. At this stage, it is partitioned in multiple pieces $(p_i, i \in [1, 6])$. In order to perform the *SpMV* tasks (line 7, next iteration) which all need the whole vector $p$, these pieces are unpartitioned to form the input variable $p$.

All in all, this task flow contains four synchronization points per iteration and is very thin. Section 3.4.5.1 exhibits the induced limitation in terms of pipelining, while Sections 3.4.5.2, 3.4.5.3 and 3.4.5.4 propose successive improvements allowing us to alleviate the synchronizations and design a wider task flow, thus increasing the concurrency and the performance.

### 3.4.2 Experimental setup

All the tests presented in Section 3.4 have been run on a cache coherent Non Uniform Memory Access (ccNUMA) machine with two hexa-core processors Intel Westmere Xeon X5650, each one having 18GB of RAM, for a total of 36GB. It is equipped with three NVIDIA Tesla M2070 GPUs, each one having 6GB of RAM memory.

The task-based CG algorithm proposed in Section 3.4.1 is implemented on top of the StarPU runtime system (see Section 3.2.3). We use the opportunity offered by StarPU to control each GPU with a dedicated CPU core. We rely on the CUBLAS v2.0 and CUSPARSE v1.0 vendor libraries to implement the GPU tasks that execute the level-one BLAS and SpMV operations.

To illustrate our discussion we consider the matrices presented in Table 3.2. As discussed above, matrices are divided in block-rows and each block-row is mapped on a processing unit (GPU or CPU core). In all the experiments presented in this study, each block-row is thus initially prefetched on the processing unit that will perform the corresponding SpMV

| Matrix name | $nnz$ | $\mathcal{N}$ | $nnz/\ \mathcal{N}$ | flop / iteration |
|---|---|---|---|---|
| `11pts-256` | 183 M | 16,7 M | 10.9 | 1,90 G |
| `11pts-128` | 22,8 M | 2,10 M | 10,9 | 224 M |
| `Audi_kw` | 154 M | 943 K | 163 | 317 M |
| `af_0_k101` | 17,5 M | 503 K | 34 | 38.6 M |

Table 3.2: Overview of sparse matrices used in this study. The `11pts-256` and `11pts-128` matrices are obtained from a 3D regular grid with 11pt discretization stencil. The `Audi_kw` and `af_0_k101` matrices come from structural mechanics simulations on irregular finite element 3D meshes.

task, *before* the experiment is timed, in order to represent the behavior occurring in a CG iteration (except for the first iteration, which is not considered here). Furthermore, when assessing the behavior of the building block operations (Section 3.4.4), the corresponding block-vectors are also prefetched on the memories associated with those processing units.

### 3.4.3 Scheduling and mapping strategy

As discussed in Section 3.4.1, the task flow derived from Algorithm 8 contains four synchronization points per iteration and is very thin, ensuring only a very limited concurrency. Furthermore, as we will show in Section 3.4.4, the tasks are not very compute intensive so that moving data between processing units is expensive relatively to the actual execution of the task. Pipelining the task flow efficiently is thus very challenging. In particular, dynamic strategies that led to close to optimum scheduling in dense linear algebra [5] are not well suited here. We have indeed experimented such a strategy (Minimum Completion Time (MCT) policy [138]) but all studied variants failed to achieve a very high performance. Indeed, with such a thin graph, each inaccurate decision induced a strong imbalance that could not be recovered. We have thus implemented a static scheduling strategy. Each block-row of the matrix is associated with a processing unit and the related tasks are performed on that processing unit. In order to increase concurrency, we have considered the case where there are more block-rows than processing units. In that case, we perform a cyclic mapping of the block-rows on the processing units in order to ensure load balancing.

### 3.4.4 Building block operations

In order to explore the potential parallelism of the CG algorithm, we first study the performance of its building block operations, level-one BLAS and $SpMV$, on our platform.

#### 3.4.4.1 Level-one BLAS operations

As mentioned in Section 3.4.1, three level-one BLAS operations are used in CG: *dot*, *axpy* and $scal - axpy$. These kernels have a computational cost of $2N$ for the *dot* and *axpy* operations and of $4N$ for the $scal - axpy$ operation, where $N$ is the size of the vectors. This low computational cost relatively to the amount of data involved makes them hard to accelerate on GPUs. Figure 3.8 shows this effect. We recall that the matrix is split in balanced

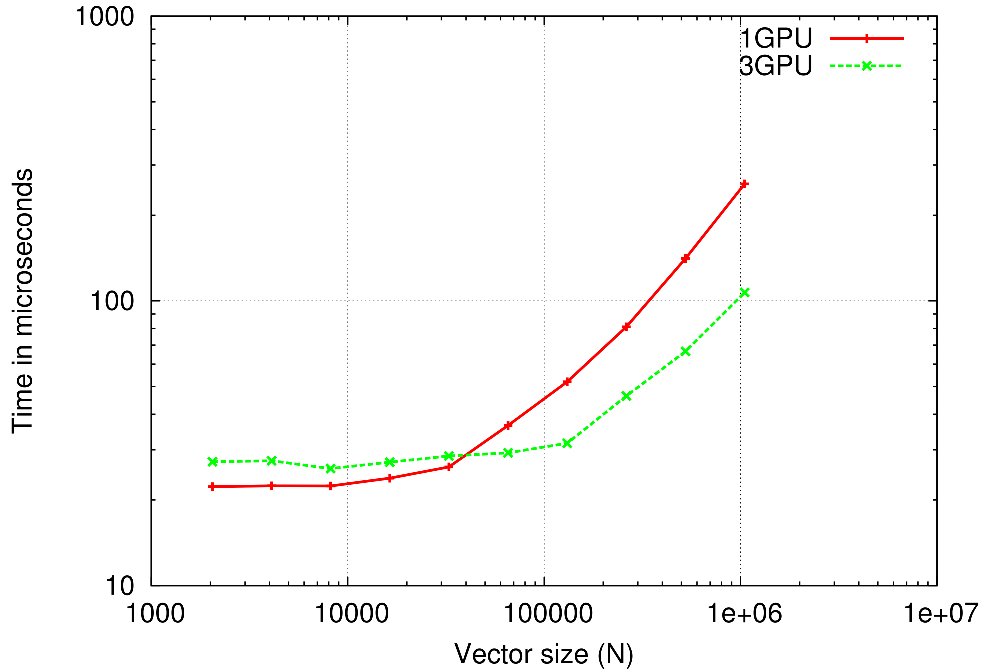



Figure 3.8: Performance of level-one BLAS operations on multiple GPUs. Both axes are expressed in logarithmic scale. Data is divided in equal pieces and is prefetched on every GPU before execution and performance assessment. Here is shown the performance of the *axpy* kernel, but all kernels follow the same behavior.

block-rows and that the vectors are divided accordingly. Following the experimental set up presented in Section 3.4.2, the input vectors are prefetched on the GPUs before execution. Therefore, in this experiment, only the computational time is measured. Nonetheless, very large vectors need to be considered ($N > 10^6$) to benefit from multi-GPU acceleration (relatively to one GPU). When the input vectors are of intermediate size ($10^5 < N < 10^6$), the use of multiple GPUs does not speed up the overall performance anymore. When the vectors are smaller ($N < 10^5$), a multi-GPU context even slows down the execution. In that latter case, the computation time is negligible and there is still no communication; the only measured time is the start-up spent for launching the kernels. Because of a lock occurring within the NVIDIA driver[1] when launching a CUDA kernel, at such a fine granularity, concurrent tasks are thus actually serialized and the launching times are paradoxically and unexpectedly cumulated.

[1]we have reported this surprising behavior to NVIDIA

#### 3.4.4.2 The SpMV operation

When the matrix is split in multiple block-rows, the SpMV operation may be executed as concurrent SpMV tasks. The performance obtained for the *audikw_1* matrix is shown

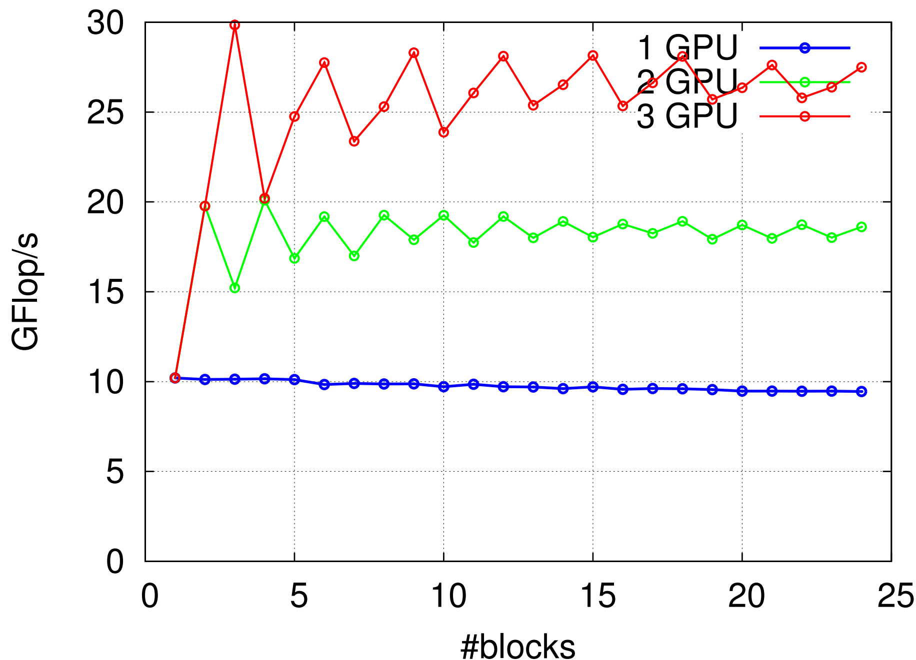



Figure 3.9: Performance of *SpMV* with *audikw_1* matrix when split in nb_block block-rows (x-axis). All needed data is prefetched on GPUs before the experiment.

in Figure 3.9. The impact of the task granularity on performance may be assessed with the performance observed on one GPU. When the matrix is split in multiple block-rows, the performance is not strongly penalized. For instance, the mono-GPU execution only decreases less then 1% when the matrix is split in three blocks (from 10.20 Gflop/s to 10.13 Gflop/s). When multiple GPUs are used, if the number of blocks is a multiple of the number of GPUs, a correct load balancing may be furthermore achieved. In particular, when the matrix is divided in three block-rows, the penalty on granularity is minimal while achieving a decent load balancing. A performance of 29.85 Gflop/s is indeed achieved which represents a speed-up of 2.95 over the mono-GPU execution with the same number of blocks (10.13 Gflop/s) and an overall 2.93 speed-up over the best mono-GPU execution (10.20 Gflop/s).

### 3.4.5 Achieving efficient software pipelining

In accordance with the example discussed in Section 3.4.1, the matrix is split in six block-rows and three GPUs are used. We pursue our illustration with matrix `11pts-128`.

#### 3.4.5.1 Assessment of the proposed task-based CG algorithm

Figure 3.10 shows the execution of one iteration of the task flow (Figure 3.7) derived from Algorithm 8 with respect to the mapping proposed in Section 3.4.3. Figure 3.10 can be interpreted as follows. The top black bar represents the state of the CPU Random Access

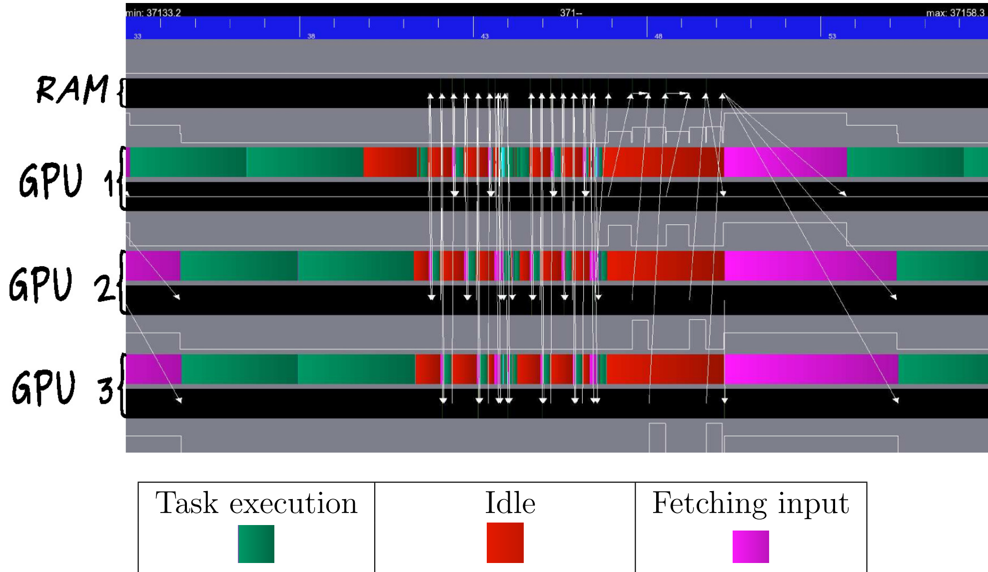



Figure 3.10: Execution trace of an iteration with the CG task flow of Figure 3.7 using three GPUs.

Memory (RAM) during the execution. The periods when the RAM is used for a data transfer from (or to) a GPU are highlighted by an arrow from the source to the destination.

As explained in Section 3.2.3, data transfers between GPUs are handled by the runtime system. In the present execution, the GPU-CPU-GPU communication mechanism (see Section 3.2.3 for details) is used. Below the bar of the CPU RAM (top black bar), the three other couples of bars represent the state of the three GPUs. Indeed, for each GPU, the top bar represents the activity of the GPU itself, whereas the bottom bar represents the state of its GPU memory. The activity of a GPU (top bar) may have one of the three following states: active computation (green), idle (red) or active waiting for the completion of a data transfer (purple).

An iteration starts with the execution of a *SpMV* operation (line 7 in Algorithm 8), time interval $[t_0,t_1]$ in Figure 3.10. As shown in Figure 3.7, the *SpMV* operation is decomposed in six tasks (green tasks in Figure 3.7). Following the cyclic mapping strategy presented in Section 3.4.3, each GPU is thus in charge of two *SpMV* tasks. At time $t_1$, vector $q$ is available, distributed in six pieces but vector $p$ is unpartitioned. The vector $p$ is partitioned into six $p_i$ pieces, $i \in [1, 6]$, with respect to the block-row decomposition of the matrix. However, this data partitioning operation is a blocking call (see Section 3.2.3) which means that no other task can be submitted until it is completed at time $t_1$ (the red vertical bar after the *SpMV* tasks in Figure 3.7). Once vector $p$ is partitioned, tasks for all remaining operations (lines 8-15) are submitted. The scalar product tasks are executed sequentially with respect to the cyclic mapping strategy explained in Section 3.4.3. The reason for

this, as explained in Section 3.4.1, is that the scalar $\alpha$ is accessed in read-write mode. In addition, $\alpha$ needs to be moved to GPU between each execution of a *dot* task (interval $[t_1, t_2]$ in Figure 3.10). Once the scalar product at line 8 is computed, the scalar division follows (line 9) executed on GPU 1 (respecting the task flow in Figure 3.7). The execution of the next two instructions follows (lines 10 and 11). But before the beginning of the execution of the *axpy* tasks on GPU 2 and GPU 3, the new value of $\alpha$ is sent according to the GPU-CPU-GPU transfer model (the purple period at $t_2$ in Figure 3.10). The *axpy* tasks (yellow tasks in Figure 3.7) are executed during the period $[t_2, t_3]$ in parallel. The scalar product at line 12 is then executed during the time interval $[t_3, t_4]$, following the same sequence as explained above for line 8. Next, $\beta$ and $\delta_{old}$ are computed on GPU 1 at time $t_4$ in Figure 3.10, representing the scalar operations from lines 13 and 14 of Algorithm 8. Tasks related to the last operation of the iteration ($scal - axpy$ tasks in Figure 3.7) are then processed during the time interval $[t_4,t_5]$. When all the new vector blocks $p_i$ are calculated, the vector $p$ is unpartitioned (red vertical bar after the $scal - axpy$ tasks in Figure 3.7). As explained in Section 3.2.3, this data unpartition is another synchronization point and may only be executed in the RAM. All blocks $p_i$ of vector $p$ are thus moved from the GPUs to the RAM during the time interval $[t_5,t_6]$ for building the unpartitioned vector $p$. This vector is then used to perform the $q_i \leftarrow A_i \times p$ tasks related to the first instruction of the next iteration (*SpMV* at line 7). We now understand why the iteration starts with an active waiting of the GPUs (purple parts before time $t_0$): vector $p$ is only valid in the RAM and thus needs to be copied on the GPUs.

During the execution of the task flow derived from Algorithm 8 (Figure 3.7), the GPUs are idle during a large portion of the time (red and purple parts in Figure 3.10). In order to achieve more efficient pipelining of the algorithm, we present successive improvements on the design of the task flow: relieving synchronization points (Section 3.4.5.2), reducing volume of communication that is achieved using a packing data mechanism (Section 3.4.5.3) and relying on a 2D decomposition (Section 3.4.5.4).

#### 3.4.5.2 Relieving synchronization points

Alternatively to the sequential execution of the scalar product, each GPU $j$ can compute locally a partial sum ($\alpha^j$) and perform a reduction to compute the final value of the scalar ($\alpha = \sum_{j=1}^{n_gpus} \alpha^j$). Figure 3.11(a) illustrates the benefit of this strategy. The calcualtion of the scalar product, during the time interval $[t_0, t_1]$ is now done in parallel. Every GPU is working on its own local copy of $\alpha$ and once they have finished, the reduction is performed on GPU 1 just after $t_1$.

The partition (after instruction 7 of Algorithm 8) and unpartition (after instruction 15) of vector $p$ represents two of the four synchronization points within each iteration. They furthermore induce extra management and data movement costs. Indeed, after instruction 15, each GPU owns a valid part of vector $p$. For instance, once GPU 1 has computed $p_1$, it sends $p_1$ to the RAM and receives it back, which is useless. Second, vector $p$ has to be fully assembled in the main memory (during the unpartition operation) before prefetching a copy of the fully assembled vector $p$ back to the GPUs (after time $t_3$ in Figure 3.11(a)). We have designed another scheme where vector $p$ is kept in a partitioned form all along the execution (it is thus no longer needed to perform partitioning and unpartitioning operations at each

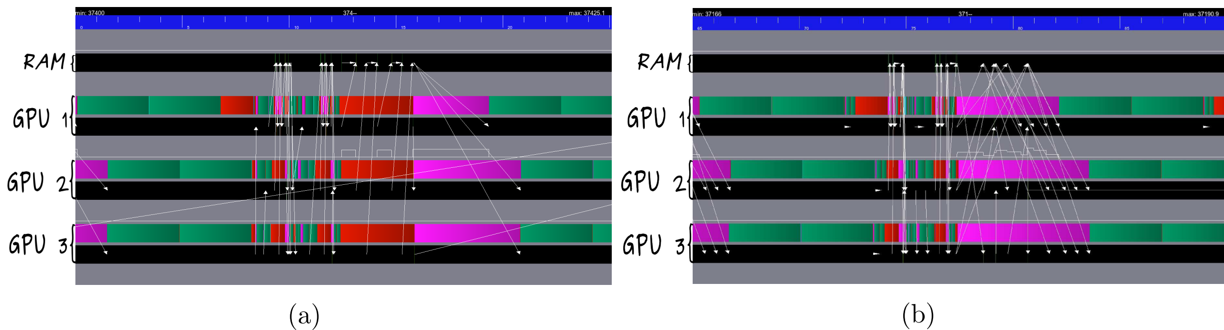



Figure 3.11: Execution trace of an iteration of the Cg algorithm on three GPUs when the vector $p$ is partitioned and unpartitioned (on the left) and a trace when these operations are avoided (on the right).

iteration). Instead of building and broadcasting the whole unpartitioned vector $p$, each GPU gathers only the missing pieces. Assuming if $p$ is decomposed in six pieces, GPU 1 had computed $p_1$ and $p_4$ at the previous iteration (following the cyclic mapping strategy presented in Section 3.4.3). So, by using this vector scheme, the GPU 1 will only receive $p_2$, $p_3$, $p_5$ and $p_6$ vector blocks. This enables us to decrease the overall traffic. Furthermore, each vector block $p_i$ can be copied from the RAM to GPUs as soon it is fetched in main memory without waiting the other vector blocks to be copied, as it was required in the previous case. Finally, we ensure that all the pieces of $p$ get copied in a contiguous fashion on the device, so that vector $p$ is valid when the task $q_i \leftarrow A_i p$ is executed.

Figure 3.11(b) illustrates the benefits of this policy. Avoiding the unpartitioning operation allows us to decrease the time required between two successive iterations from 8.8 ms to 6.6 ms. Furthermore, since the partitioning operation is no longer needed, the corresponding synchronization in the task flow control is removed. The corresponding idle time (red part at time $t_0$ in Figure 3.11(a)) is removed and instructions 7 and 8 are now pipelined (period $[t_0,t_1]$ in Figure 3.11(b)).

Coming back to Figure 3.11(a), one may notice that GPUs are idle for a while just before time $t_1$ and again just before time $t_2$. This is due to the reduction that finalizes each *dot* operation ($dot(p,q)$ at instruction 8 and $dot(r,r)$ at instruction 12, respectively). In Algorithm 8, vector $x$ is only used at lines 10 (in read-write mode) and 6 (in read-only mode). The execution of instruction 10 can thus be moved anywhere within the iteration as long as the other input data of instruction 9, i.e., $p$ and $\alpha$ have been updated to the correct value. In particular, instruction 10 can be moved after instruction 12. This delay enables us to overlap the final reduction of the *dot* occurring at instruction 12 with the computation of vector $x$.

The red part before $t_2$ in Figure 3.11(a) becomes (partially) green in Figure 3.11(b). The considered CG formulation does not provide a similar opportunity to overlap reduction finalizing the *dot* operation at instruction 8. This is why the red part before $t_1$ in Figure 3.11(a) remains red in Figure 3.11(b). We will come back to this limitation in Section 3.4.7 where we consider an alternative formulation of CG to alleviate that ultimate synchronization point but for now we focus on reducing the amount of communication by packing data.

#### 3.4.5.3 Reducing communication volume by packing data

By avoiding data partition and data unpartition operations, the broadcast of vector $p$ has been improved (from period $[t_2, t_4]$ in Figure 3.11(a) to period $[t_2, t_3]$ in Figure 3.11(b)) and the communication time remains the last main performance bottleneck (time interval $[t_2, t_3]$ in Figure 3.11(b)). This volume of communication can be decreased. Indeed, if a column within the block-row $A_i$ is zero, then the corresponding entry of $p$ is not involved in the computation of the task $q_i \leftarrow A_i p$. Therefore, $p$ can be pruned. With a SPMD model, using MPI, the natural method would consist in packing the relevant data into buffers before performing the associated communication.

We now explain how we can achieve a similar behavior with a task flow model. Instead of broadcasting the whole vector $p$ on every GPU, we can only transfer the required subset. Before executing the CG iterations, this subset is identified with a symbolic preprocessing step. Based on the structure of the block $A_{i,j}$, we determine which part of $p_j$ is needed to build $q_i$. If $p_j$ is not fully required, we do not transfer it directly. Instead, it can be packed into an intermediate data, $p_{i,j}$. According to the task-based model, this packing operation can be implemented with a task. The receiving GPU then needs to unpack this data before performing the SpMV (since SpMV operates on a full vector). One possibility would be to implement this unpack operation with a new task. In order to limit the overhead due to the task creation, we could alternatively merge this unpack operation with the $SpMV$ operation, resulting in a $sparse_SpMV$ task, where the input vector is now sparse. However, StarPU provides an elegant support for implementing all these advanced techniques through the definition of new data types (see discussion on primitives in Section 3.2.3 for details). We rely on that mechanism for implementing this packing scheme.

A direct consequence of using a packing scheme is that the amount of data transferred on the critical path is potentially dramatically decreased. It is then advantageous to rely on direct GPU-GPU communication mechanism (see Section 3.2.3 for details) since different parts of vector $p_i$ are now transferred to different GPUs. To do so, we just need to enable the GPU-GPU communication mechanism provided by StarPU.

Other optimizations related to data movement have also been designed. First, the packing operation may have a non negligible cost whereas sometimes the values of $p_{i,j}$ that needs to be sent are almost contiguous. In those cases, it may thus be worth sending an extra amount of data in order to directly send the contiguous superset of $p_{i,j}$ ranging from the first to the last index that needs to be transferred. We have implemented such a scheme. A preliminary tuning is performed for each matrix and each $p_{i,j}$ subvector to choose whether $p_{i,j}$ is packed or transferred in a contiguous way. Second, although StarPU can perform automatic prefetching, the prefetching operation is performed once all the dependencies are satisfied. In the present context, this may be too much late and further anticipation may be worthy. Therefore, we are performing explicit data prefetching as soon as possible (see Section 3.2.3) within the callback of the $scal - axpy$ task. We also do so after the computation of the $\alpha$ and $\beta$ scalar values (lines 9 and 13 in Algorithm 8) for broadcasting them on all GPUs.

Figure 3.12 shows the execution trace. The time interval $[t_2, t_3]$ in Figure 3.11(b) needed for the broadcasting of the vector $p$ has been reduced to the interval $[t_0, t_1]$ in Figure 3.12. In

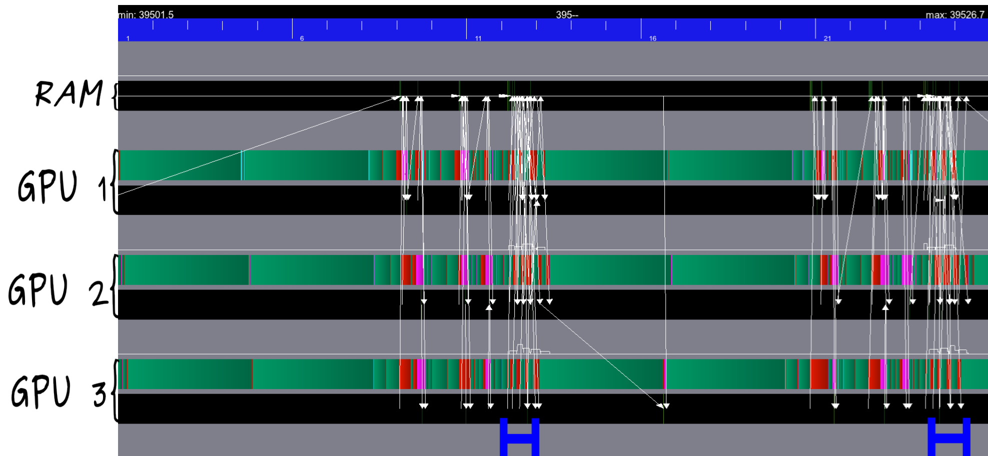



Figure 3.12: Execution trace when furthermore the vector $p$ is packed.

the rest of the chapter we refer to as the *full* algorithm, when all the blocks are transferred or to as the *packed* algorithm if this packing mechanism is used.

#### 3.4.5.4 2D decomposition

The 1D decomposition scheme requires that for each *SpMV* task, all blocks of vector $p$ (packed or not packed) are in place before starting the execution of the task. In order to be able to overlap the time needed for broadcasting the vector $p$ (time interval $[t_0, t_1]$ in Figure 3.12) a 2D decomposition must be applied to the matrix. The matrix is first divided in block-rows, and then the same decomposition is applied to the other dimension of the matrix. Similarly as for a 1D decomposition, the entire block-row will be mapped on a single GPU. Contrary to the 1D decomposition, where we had to wait for the transfer of all missing blocks of the vector $p$, with the 2D decomposition, time needed for the transfer of the vector $p$ can be overlapped with the execution of the tasks for which the blocks of the vector $p$ are already available.

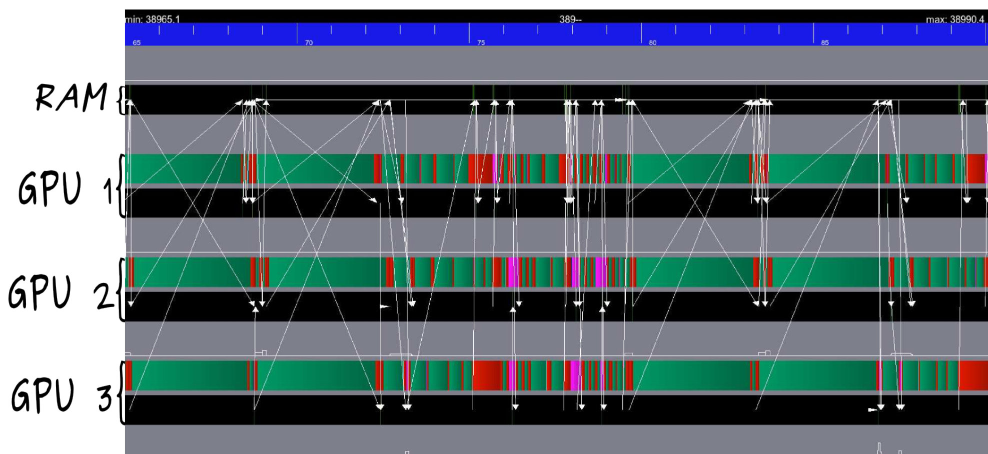



Figure 3.13: Execution trace when relying of a 2D decomposition of the matrix.

The result of the impact of a 2D decomposition is shown in Figure 3.13. During the time

interval $[t_1, t_2]$ in Figure 3.12 there is no communication. All of them are performed before the beginning of the *SpMV* task (time interval $[t_0, t_1]$). In Figure 3.13 during the time interval $[t_0, t_1]$, communications are performed while the *SpMV* is executed. In the rest of the chapter we refer to either 1D or 2D depending on the data decomposition used. The trade-off between large task granularity (1D) and increased pipeline (2D) will be discussed in Section 3.4.6.

### 3.4.6 Performance analysis

We now propose a detailed performance analysis of the task-based CG algorithm designed above. In this thesis, we focus on the design of efficient parallel schemes and rely on externals libraries to perform the inner-most computation. As a consequence, we propose to analyze the behavior of our algorithms in terms of *speed-up* with respect to the execution occurring on one single computational unit (or worker to be consistent with the terminology introduced in Section 3.2.3). When normalized to the number of workers involved in the parallel computation, such a speed-up is called *parallel efficiency* (or simply efficiency). If we denote $T_b(p)$ the *elapsed time* with $p$ workers when the matrix is decomposed in $b$ blocks, these metrics can be defined as follows:

**Definition 1.** *The speed up, denoted by $S$, and the overall efficiency, denoted by $e$, are defined as $S = T_1(1)/T_b(p)$ and $e = T_1(1)/(T_b(p) \times p)$, respectively.*

Generally speaking, three major factors influence the overall efficiency. First data need to be decomposed in order to have concurrent tasks. The drawback of this decomposition is that we operate at a smaller granularity, which may deteriorate the performance of the computing kernels. Second, a given task executed concurrently with other tasks may have a lower performance because it shares resources with those other tasks such as caches, buses, . . . , which may be especially critical on modern multicore chips. Third, to be triggered, a task must satisfy two conditions: its predecessors must have been completed and the data it operates on must have been fetched on memory associated with the worker that will process it. If one of these conditions is not satisfied, some processing units may be idle.

We now show that the overall efficiency may be decomposed as the product of three efficiencies representing those respective effects. To that effect, the *cumulated time* spent by workers executing tasks is denoted as $\mathcal{T}_b^{exe}(p)$ whereas the *cumulated time* they spent being idle is denoted as $\mathcal{T}_b^{idle}(p)$. Note that elapsed times are expressed in units of *time* (say *seconds*) whereas cumulated times are expressed in units of *processing unit* × *time* (say *processors* × *seconds*). Because workers are either idle or executing tasks, we have:

**Property 1.** $T_b(p) \times p = \mathcal{T}_b^{exe}(p) + \mathcal{T}_b^{idle}(p)$.

When using one worker, the cumulated time spents executing tasks when data is decomposed in $b$ blocks (*i.e.*, $\mathcal{T}_b^{exe}(1)$) is likely to be longer than when data is not split and tasks act on the complete matrix (*i.e.*, $\mathcal{T}_1^{exe}(1)$). As observed in Section 3.4.4, this effect may indeed be significant in the mono-GPU case. The measure of this penalty due to the fact we operate at a lower granularity on the efficiency may be quantified as follows:

**Definition 2.** *The effect of operating at a lower granularity on the efficiency is defined as:* $e_{granularity} = \mathcal{T}_1^{exe}(1)/\mathcal{T}_b^{exe}(1)$.

**Definition 3.** *The effect of concurrency on the efficiency of tasks is defined as:* $e_{tasks} = \mathcal{T}_b^{exe}(1)/\mathcal{T}_b^{exe}(p)$.

In the multi-GPU case, the efficiency of tasks is trivially equal to one (*i.e.*, $e_{tasks} = 1$) as multiple GPUs execute tasks independently (without sharing any hardware capability) from one another and are thus not penalized by their mutual concurrency. On the other hand, in the multicore case, a given task executed concurrently with other tasks may have a lower performance because it shares resources with those other tasks such as caches or buses.

Finally, a worker may spend part of its time being idle because no task can be processed at a given time. The ability of a task flow to efficiently pipeline tasks and therefore prevents this drawback can be measured in terms of proportion of the cumulated time spent in tasks $\mathcal{T}_b^{exe}(p)$ with respect to the overall cumulated time $\mathcal{T}_b^{exe}(p) + \mathcal{T}_b^{idle}(p)$:

**Definition 4.** *The effect of achieving a high task pipeline on the efficiency is defined as:* $e_{pipeline} = \mathcal{T}_b^{exe}(1)/(\mathcal{T}_b^{exe}(p) + \mathcal{T}_b^{idle}(p))$.

In the mono-worker case (*i.e.*, $p = 1$), the idle time (*i.e.*, $\mathcal{T}_b^{idle}(1)$) may be neglected with respect to the elapsed time (experiments not further detailed showed that we consistently had $\mathcal{T}_b^{idle}(1) << 0.1\% \times T_b(1)$) and we can thus consider that $T_b(1) = \mathcal{T}_b^{exe}(1)$. Furthermore, given that the elapsed time can be expressed in terms of the sum of cumulated time spent in execution and idle modes normalized with the number of processing units (Property 1), the overall efficiency $e$ can then precisely be expressed as the product of the three effects as stated by Property 2.

**Property 2.** $e = e_{granularity} \times e_{tasks} \times e_{pipeline}$.

#### 3.4.6.1 Multi-GPU experiments

Tables 3.3, 3.4, 3.5 and 3.6 present the performance achieved for all four matrices for the multi-GPU case. The optimal performance is represented for each scheme. The objective of this section is to assess the benefits of using multiple GPUs over a mono-GPU execution. We thus rely on the metrics defined above and present a detailed performance assessment.

The first thing to be observed is that for all the matrices the packed version of our algorithm where only just the needed part of the vector $p$ is broadcasted yields the optimal performance. Broadcasting entire sub-blocks is too expensive and thus considerably slows down the execution of the CG algorithm on multi-GPU platforms. For the matrices that have a regular distribution of the non zeros, *i.e.*, the `11pts-256`, `11pts-128` and the `af_0_k101` matrices, the 1D algorithm outperforms the 2D algorithm as shown in Tables 3.3, 3.4 and 3.6, respectively. On the other hand, in the case of the `Audi_kw` matrix that has an unstructured pattern, the 2D algorithm exhibits more parallelism, which enables the communication time needed for broadcasting larger portion of the vector $p$ to be overlapped with the execution of the 2D block of the *SpMV* that is already in-place.

| | 1D | | 2D | |
|---|---|---|---|---|
| # GPUs | full | packed | full | packed |
| 1 | **9.74** | | | |
| 2 | 12.33 | 19.10 | 16.66 | 17.24 |
| 3 | 11.70 | **28.39** | 13.26 | 23.17 |

Table 3.3: Performance (in Gflop/s) of CG on the `11pts-256` matrix for the multi-GPU case.

| | 1D | | 2D | |
|---|---|---|---|---|
| # GPUs | full | packed | full | packed |
| 1 | **9.58** | | | |
| 2 | 11.5 | 17.6 | 14.3 | 16.1 |
| 3 | 9.01 | **24.2** | 9.22 | 20.6 |

Table 3.4: Performance (in Gflop/s) of CG on the `11pts-128` matrix for the multi-GPU case.

| | 1D | | 2D | |
|---|---|---|---|---|
| # GPUs | full | packed | full | packed |
| 1 | **10.0** | | | |
| 2 | 15.6 | 15.6 | 16.3 | 16.7 |
| 3 | 17.7 | 20.0 | 22.0 | **23.6** |

Table 3.5: Performance (in Gflop/s) of CG on the `Audi_kw` matrix for the multi-GPU case.

| | 1D | | 2D | |
|---|---|---|---|---|
| # GPUs | full | packed | full | packed |
| 1 | **9.84** | | | |
| 2 | 12.1 | 16.3 | 13.6 | 15.0 |
| 3 | 11.1 | **19.4** | 12.5 | 18.2 |

Table 3.6: Performance (in Gflop/s) of CG on the `af_0_k101` matrix for the multi-GPU case.

| Matrix | 11pts-256 | 11pts-128 | Audi_kw | af_0_k101 |
|---|---|---|---|---|
| $S$ | 2.91 | 2.52 | 2.36 | 1.97 |
| $e$ | 0.97 | 0.84 | 0.79 | 0.65 |
| $e_{granularity}$ | 0.99 | 0.98 | 0.87 | 0.96 |
| $e_{tasks}$ | 1.00 | 1.00 | 1.00 | 1.00 |
| $e_{pipeline}$ | 0.97 | 0.86 | 0.91 | 0.68 |

Table 3.7: Speed-up and efficiency. If $T_b(p)$ represents the execution time with $p$ GPUs when the matrix is decomposed in $b$ block-rows, $S$ the speed-up, $e$ the overall efficiency, the effects of granularity on efficiency $e_{granularity}$, the effect of using multiple GPUs at the same time $e_{tasks}$ and the effects of pipeline on efficiency $e_{pipeline}$ are defined with Definition 1, 2, 3 and 4, respectively. In our case $p = 3$ and all values in this table are obtained using the algorithm which yields the best overall performance (bold values in tables 3.3, 3.4, 3.5 and 3.6).

Table 3.7 allows for analyzing how the overall efficiency is decomposed according to the metrics proposed above. Dividing the `11pts-256` matrix in several block-rows does not induce a penalty on the task granularity ($e_{granularity} = 0.99 \approx 1$). Furthermore, thanks to all the improvements of the task flow proposed in Sections 3.4.5.2 and 3.4.5.3 a pipeline efficiency of $e_{pipeline} = 0.97$ is achieved. All in all, a global efficiency of $e = 0.97$ is obtained. For the `11pts-128` matrix, the matrix decomposition induces a similar granularity penalty $e_{granularity} = 0.98$. The slightly lower granularity efficiency is a direct consequence of the matrix order. For smaller matrices, the tasks are preformed on smaller sizes, thus the execution time per task is decreased. This makes our algorithm more sensitive to the overhead created by the communications induced by the dot-products and the broadcasting of the vector $p$, ending up with a pipeline efficiency of $e_{pipeline} = 0.86$ with the 1D algorithm. The global efficiency for this matrix is $e = 0.84$. This phenomenon is amplified when the matrix order is getting lower, such as in the case of the `af_0_k101` matrix, resulting in a global efficiency of $e = 0.65$ and mainly due to a limited pipeline ($e_{pipeline} = 0,68$).

The `Audi_kw` matrix yields optimal performance with the 2D algorithm (see Section 3.4.5.4). Although the 2D algorithm requires to split the matrix in many more blocks inducing a higher penalty on granularity ($e_{granularity} = 0.87$), it allows for a better overlap of communication with computation ensuring that a higher pipeline ($e_{pipeline} = 0.91$) is achieved. With this trade-off, an overall efficiency of equal to $e = 0.79$ is obtained.

#### 3.4.6.2 Multicore experiments

Table 3.8 presents the performance achieved for all four matrices in the multicore case. The optimal performance is represented for each scheme. For a comparison purpose, we have also

implemented a CG algorithm on top multi-threaded MKL. In this case, for each operation the corresponding MKL kernel is called and the multi-threading is managed internally by the library. The observed performance for the MKL-based code are also presented in Table 3.8.

| | | 11pts-256 | | 11pts-128 | | Audi_kw | | af_0_k101 | |
|---|---|---|---|---|---|---|---|---|---|
| #CPU | version | 1D | 2D | 1D | 2D | 1D | 2D | 1D | 2D |
| 1 | STF | **1.15** | | **1.19** | | **1.32** | | **1.37** | |
| | MKL | **1.10** | | **1.12** | | **1.25** | | **1.37** | |
| 3 | STF | 1.80 | 1.79 | 1.90 | 1.70 | 2.14 | 2.01 | 2.30 | 2.09 |
| | MKL | 2.19 | | 2.21 | | 2.22 | | 2.54 | |
| 6 | STF | 2.13 | 2.16 | 2.21 | 2.07 | 2.51 | 2.40 | 2.60 | 2.48 |
| | MKL | 3.13 | | 3.02 | | 2.75 | | 3.77 | |
| 12 | STF | **3.71** | – | **3.64** | 3.51 | **4.23** | 3.75 | **3.91** | 3.71 |
| | MKL | **3.49** | | **3.41** | | **3.24** | | **4.14** | |

Table 3.8: Performance (in Gflop/s) of our version and the MKL version of the CG algorithm for the multicore case.

For each matrix when only one CPU is used we observe a performance slightly larger than 1 Gflop/s (both for our task-based code and the MKL-based version). As a consequence, when 12 CPU cores are used, ideally we would expected a parallel performance of about 12 Gflop/s. Nevertheless, for all the matrices the optimal performance obtained with 12 CPU cores is approximately 4 Gflop/s. Table 3.9 indeed shows that in spite of a very decent pipeline ($0.89 \leq e_{pipeline} \leq 0.99$), the overall efficiency is strongly limited by the efficiency of the tasks ($0.26 \leq e_{tasks} \leq 0.28$). Indeed, CG relies on memory-bound operations. On this platform, we achieve a bandwidth peak of around 25 GB/s (theoretical peak is 32 GB/s). For instance, the most costly operation for the CG algorithm is the *SpMV* operation. On this platform the optimal performance for this operation is about 5.5 Gflop/s with the CSR spare matrix format. On the other hand, dividing the matrix in several sub-blocks does not deteriorate the achieved performance ($e_{granularity} = 1$ for all matrices). These matrices are large enough to provide enough computational load for the CPU cores even once divided.

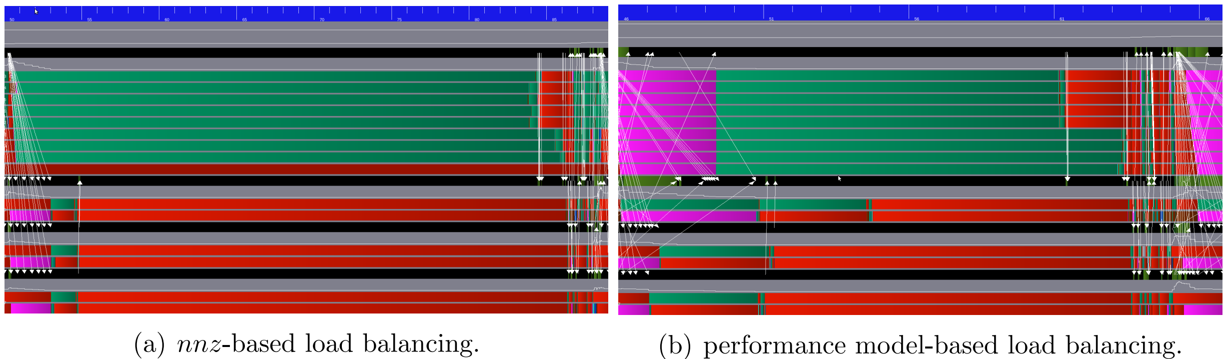
#### 3.4.6.3 Preliminary results in the heterogeneous case

One advantage of relying on task-based programming is that the architecture is fully abstracted. We have shown above that the same code could hence be executed on both a multi-GPU platform and a multicore platform while consistently achieving very competitive performance. We prove here that we can furthermore benefit from this design to run on an heterogeneous node composed of all available computational resources. Because the considered platform has 12 CPU cores and three GPUs but that each GPU has a CPU core dedicated to handle it, we can rely on nine CPU workers and three GPU workers in total.

Figure 3.14 present execution traces relying on two different strategies for balancing the

| Matrix | 11pts-256 | 11pts-128 | Audi_kw | af_0_k101 |
|---|---|---|---|---|
| $S$ | 3.22 | 3.06 | 3.20 | 2.85 |
| $S_{Mkl}$ | 3.17 | 3.04 | 2.59 | 3.02 |
| $e_{granularity}$ | 1.00 | 1.00 | 1.00 | 1.00 |
| $e_{tasks}$ | 0.26 | 0.26 | 0.28 | 0.27 |
| $e_{pipeline}$ | 0.99 | 0.97 | 0.96 | 0.89 |
| $e$ | 0.27 | 0.25 | 0.27 | 0.24 |
| $e_{Mkl}$ | 0.26 | 0.25 | 0.22 | 0.25 |

Table 3.9: Speed-up and efficiency for the multicore case. $S$ the speed-up, $e$ the overall efficiency, the effects of granularity on efficiency $e_{granularity}$, the effect of using multiple CPUs at the same time $e_{tasks}$ and the effects of pipeline on efficiency $e_{pipeline}$ are defined with Definition 1, 2, 3 and 4 respectively. In our case $p = 12$ and all values in this table are obtained using the algorithm which yields the best overall performance (bold values in Tables 3.8).



(a) $nnz$-based load balancing.

(b) performance model-based load balancing.

Figure 3.14: Traces of an execution of one iteration of the CG algorithm in the heterogeneous case (nine CPU and three GPU workers) with different partitioning strategies for the Audi_kw matrix. The $nnz$ is equilibrated per block-row in 3.14(a) ($33\mu s$). In 3.14(b) a feed-back from a history based performance model is used for the partitioning of the matrix ($16\mu s$).

load between CPU cores and GPU. These traces show the ability of task-based programming in exploiting heterogeneous platforms. However, they also show that more advanced load balancing strategies need to be designed in order to achieve a better occupancy. This question has not been further investigated and remains open for future work.

### 3.4.7 Combining software pipelining with numerical pipelining

We have shown above that task-based programming allows for achieving a high efficiency of software pipelining in the case of the classical formulation of the CG algorithm. To cope with the large number of computational units, another possibility consists in rewriting the considered numerical algorithm in order to alleviate numerical synchronizations. In the particular case of CG, a so-called "pipelined CG" algorithm was recently proposed [69]. Equivalent to the classical formulation of CG in exact arithmetic, the idea consists in interleaving part of the computation from one iteration to another as shown in the pseudo-code provided in Algorithm 9. In this section, we study the effects of combining the software pipelining techniques proposed above together with the numerical pipelining from [69] (Algorithm 9). For that, all optimizations discussed in Section 3.4.5 for achieving a high software pipelining are applied to the pipelined CG formulation (Algorithm 9).

**Algorithm 9** Pseudo-code of the pipelined CG algorithm.

1: $r_0 \leftarrow b - Ax_0$
2: $u_0 \leftarrow r$
3: $w_0 \leftarrow A_0 u$
4: **for** $j = 0, 1, ...,$ until $\frac{\|b-Ax\|}{\|b\|} \leq eps$ **do**
5: $\quad \gamma_i = dot(r_i, u_i)$
6: $\quad \delta = dot(w_i, u_i)$
7: $\quad n_i = Aw_i$
8: $\quad$ **if** $i > 0$ **then**
9: $\quad\quad \beta_i = \gamma_i/\gamma_{i-1}; \alpha_i = \gamma_i/(\delta - \beta_i\gamma_i/\alpha_{i-1})$
10: $\quad$ **else**
11: $\quad\quad \beta_i = 0; \alpha_i = \gamma_i/\delta$
12: $\quad$ **end if**
13: $\quad z_i \leftarrow n_i + \beta_i z_{i-1}$
14: $\quad q_i \leftarrow w_i + \beta_i q_{i-1}$
15: $\quad p_i \leftarrow u_i + \beta_i p_{i-1}$
16: $\quad x_{i+1} \leftarrow x_i + \alpha_i p_i$
17: $\quad r_{i+1} \leftarrow r_i - \alpha_i q_i$
18: $\quad u_{i+1} \leftarrow u_i - \alpha_i z_i$
19: $\quad w_{i+1} \leftarrow w_i - \alpha_i z_i$
20: **end for**

Although equivalent to the classical formulation of CG in exact arithmetic, pipelined CG does not consist of the exact same sequence of computation and uses three additional vectors ( vectors $u, w$ and $z$ in Algorithm 9). As a consequence, in finite arithmetic, convergence may be impacted. These effects are out of the scope of this thesis and the reader is invited

to read [69] for further details. Another impact of relying on the pipelined CG formulation is that additional calculation (extra-flop) are performed. More precisely, two additional *scal* − *axpy* (lines 13 and 14 in Algorithm 9) and two additional *axpy* (lines 18 and 19 in Algorithm 9) operations are required. Because the choice of relying on an algorithm performing extra-flop shall be transparent to an end-user we normalize the number of flop performed with pipelined CG to the classical case. This choice also makes performance (in Gflop/s) of pipelined CG presented below immediately comparable to the one of classical CG discussed in Section 3.4.6: the higher, the better.

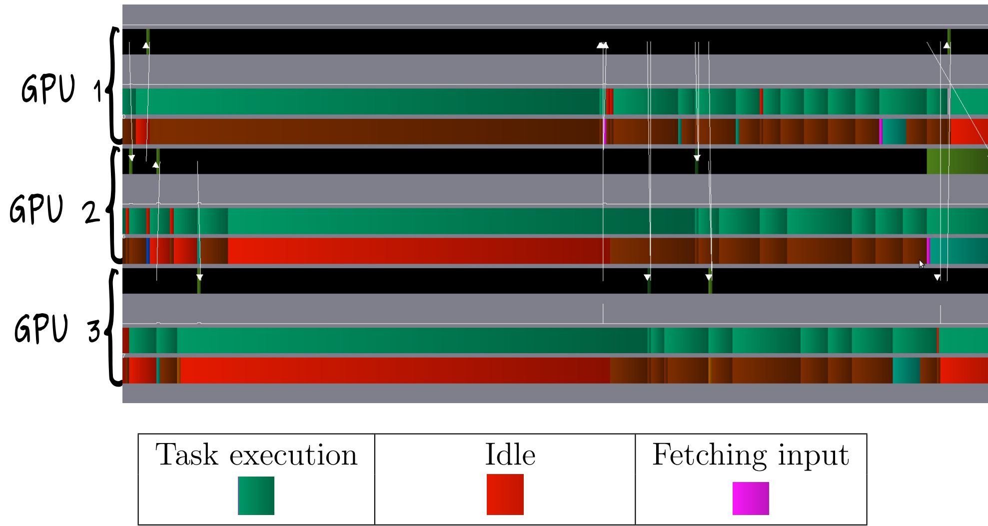



Figure 3.15: Execution trace of an iteration with the 1D pipelined CG using three GPUs with the `11pts-256` matrix.

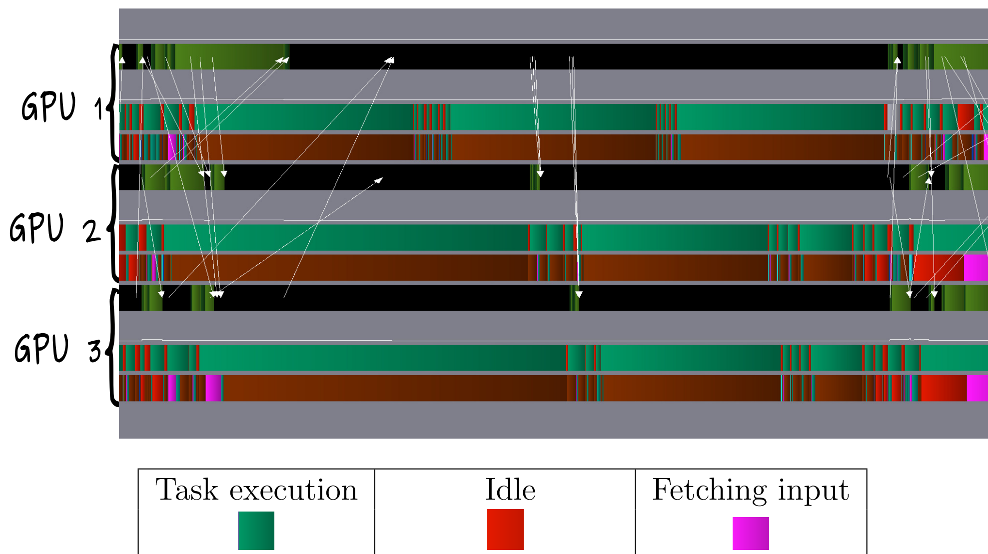



Figure 3.16: Execution trace of an iteration with the 2D pipelined CG using three GPUs with the `Audi_kw` matrix.

Figure 3.15 shows the resulting trace for the `11pts-128` matrix in the 1D case. We

can observe that this pipelined version of the task-based CG algorithm overcomes the numerical synchronization point that was remaining with the previous task-based code based on classical CG that was due to the reduction finalizing the *dot* operation (line 8 of Algorithm 8) as it was highlighted in Section 3.4.5.2. For matrices with an irregular pattern where we showed above that the 2D version is more appropriate, the impact is even more spectacular. Figure 3.16 indeed shows that the `Audi_kw` matrix processed with a 2D scheme is now fully pipelined.

| Matrix | `11pts-256` | `11pts-128` | `Audi_kw` | `af_0_k101` |
|---|---|---|---|---|
| $S$ | 33.1* | 2.73 | 2.42 | 2.29 |
| $e$ | 11.0* | 0.90 | 0.81 | 0.76 |

Table 3.10: Speed-up and efficiency of task-based pipelined CG. If $T_b(p)$ represents the execution time with $p$ GPUs when the matrix is decomposed in $b$ block-rows, the speed-up $S$ and the overall efficiency $e$, are defined with Definition 1. In our case $p = 3$ and all values in this table are obtained using the algorithm which yields the best overall performance (bold values in Tables 3.12, 3.13, 3.14 and 3.15). The '*' symbol indicates that the mono-GPU execution occurred out-of-GPU-memory (the runtime system transparently handles it, preventing a failure but not a performance penalization).

| Matrix | `11pts-128` | `Audi_kw` | `af_0_k101` |
|---|---|---|---|
| $S$ | 3.15 | 3.21 | 3.07 |
| $e$ | 0.62 | 0.28 | 0.26 |

Table 3.11: Speed-up and efficiency of task-based pipelined CG. If $T_b(p)$ represents the execution time with $p$ CPU cores when the matrix is decomposed in $b$ block-rows, the speed-up $S$ and the overall efficiency $e$, are defined with Definition 1. In our case $p = 12$ and all values in this table are obtained using the algorithm which yields the best overall performance (bold values in Table 3.16.)

Table 3.10 presents the speed-up and efficiency of the task-based pipelined CG code on three GPUs according to the best overall performance represented in bold values in Tables 3.12, 3.13, 3.14 and 3.15. Similarly, Table 3.11 shows the speed-up and efficiency when running on 12 CPU cores according the the best overall performance represented in bold values in Table 3.16. All in all we observe that pipelined CG achieves higher speed-ups and efficiency than the corresponding classical CG studied in Section 3.4.6 in all cases. However, because of the extra-flop (which we do not count as mentioned above), the achieved performance (in terms of Gflop/s) is almost consistently lower both for the multi-GPU (Tables 3.12, 3.13, 3.14 and 3.15) and the multicore (Table 3.16) cases. The `Audi_kw`

matrix processed on a multicore processor is the only exception because its irregular pattern makes any synchronization point prohibitive and the ability of pipelined CG to fully remove them brings a definite benefit (see again Figure 3.16).

This observation goes beyond the study of task-based algorithms. Indeed, the proposed task-based classical CG and pipelined CG implementations can be viewed as highly optimized versions of the algorithms. As a consequence, the above results show that adding additional pipeline through numerical pipelining is not necessarily beneficial if the original algorithm already benefits from a high level of software pipelining. Nonetheless, having the flexibility of adding up this extra level of pipelining provides the opportunity to further accelerate CG in the cases where the remaining numerical synchronization point would otherwise be too much penalizing.

| | 1D | | 2D | |
|---|---|---|---|---|
| # GPUs | full | packed | full | packed |
| 1 | **0.70*** | | | |
| 2 | 11.09 | 15.55 | 11.50 | 14.27 |
| 3 | 11.09 | **23.18** | 14.20 | 19.47 |

Table 3.12: Performance (in Gflop/s) of pipelined CG on the `11pts-256` matrix for the multi-GPU case. The '*' symbol indicates that the execution occurred out-of-GPU-memory (the runtime system transparently handles it, preventing a failure but not a performance penalization).

| | 1D | | 2D | |
|---|---|---|---|---|
| # GPUs | full | packed | full | packed |
| 1 | **7.76** | | | |
| 2 | 10.63 | 14.88 | 11.05 | 13.69 |
| 3 | 10.40 | **21.14** | 12.51 | 18.49 |

Table 3.13: Performance (in Gflop/s) of pipelined CG on the `11pts-128` matrix for the multi-GPU case.

| | 1D | | 2D | |
|---|---|---|---|---|
| # GPUs | full | packed | full | packed |
| 1 | **9.33** | | | |
| 2 | 15.42 | 14.92 | 13.92 | 14.61 |
| 3 | 18.59 | 19.03 | 20.88 | **22.60** |

Table 3.14: Performance (in Gflop/s) of pipelined CG on the `Audi_kw` matrix for the multi-GPU case.

| | 1D | | 2D | |
|---|---|---|---|---|
| # GPUs | full | packed | full | packed |
| 1 | **8.56** | | | |
| 2 | 12.05 | 15.24 | 13.22 | 14.56 |
| 3 | 12.09 | **19.58** | 13.22 | 18.45 |

Table 3.15: Performance (in Gflop/s) of pipelined CG on the `af_0_k101` matrix for the multi-GPU case.

| | `11pts-128` | | `Audi_kw` | | `af_0_k101` | |
|---|---|---|---|---|---|---|
| #CPU | 1D | 2D | 1D | 2D | 1D | 2D |
| 1 | **0.94** | | **1.22** | | **1.18** | |
| 3 | 1.26 | 1.22 | 1.68 | 1.56 | 1.75 | 1.48 |
| 6 | 1.71 | 1.66 | 2.30 | 2.19 | 2.37 | 2.08 |
| 12 | **2.96** | 2.70 | **3.92** | 2.62 | **3.62** | 3.22 |

Table 3.16: Performance (in Gflop/s) of task-based pipelined CG in the the multicore case.

## 3.5 Conclusion

In this chapter, we investigated the opportunities of designing a task-based version of MaPHyS. We first proposed a MPI+task formulation. This formulation had the advantage of being relatively conservative and incremental with respect to the approach proposed in the previous chapter in the sense that it was still an MPI+X paradigm. We however showed that it was disruptive in the sense that we could abstract the hardware architecture, hence exploiting homogeneous platforms as well as heterogeneous machines enhanced with GPUs.

To further abstract the hardware architecture, we have discussed the missing ingredients towards a full task-based expression of a sparse hybrid solver. We have identified that one key component deserved to be first studied: the Krylov subspace method. Subsequently, we have proposed a full task-based formulation of the CG algorithm. We managed to refine the task flow by removing almost all synchronizations of the classical CG algorithm, ensuring an efficient pipelining of the task flow such that GPUs are almost fully exploited. Although not so penalizing thanks to an efficient software pipelining, the remaining synchronization point could be alleviated with the use of the alternative pipelined CG formulation from [69]. There is a trade-off between task granularity and concurrency, controlled by the number of block-rows. Our performance assessment has shown that the optimum case consistently corresponds to a perfect matching between the number of blocks and the number of GPUs, meaning that benefits in terms of scheduling opportunities by splitting the matrix in a larger number of blocks are by far dominated by the penalty of operating at a smaller granularity. Contrary to dense linear algebra, where state-of-the art dynamic scheduling algorithms are sufficient to achieve nearly optimum performance, scheduling opportunities are thus much more constrained for CG. As a result, each individual scheduling decision

may be fatal to the overall performance and we have proposed a carefully defined static scheduling algorithm to prevent such effects. This statement also tells us what is (and what is not) a runtime system. A runtime system is a software layer that allows the user to express *what* to do (which task flow, which scheduling algorithm, ...) and delegate the question of *how* to do it (and how to do it efficiently) to a third party. This approach also ensures performance portability of the code, as illustrated with the benefits of being immediately able to run the code on different hardware configurations.

Another interesting conclusion is that the proposed task-based classical CG and pipelined CG implementations can be viewed as highly optimized versions of the algorithms. As a consequence, this study shows that adding additional pipeline through numerical pipelining is not necessarily beneficial if the original algorithm already benefits from a high level of software pipelining. Nonetheless, having the flexibility of adding up this extra level of pipelining provides the opportunity to further accelerate CG in the cases where the remaining numerical synchronization point would otherwise be too much penalizing.

# Perspectives and concluding remarks

In this thesis we have studied a few approaches for designing sparse hybrid solvers for modern supercomputers using the MaPHyS library as a testbed to illustrate our discussions. We have first proposed a relatively conservatory paradigm consisting in adding a second level of parallelism to the baseline MPI version of MaPHyS. To efficiently exploit hierarchical supercomputer architectures, designed as clusters of multicore processors, we have proposed an MPI+thread design. Intensive and detailed experiments up to tens of thousands of CPU cores showed that this approach could achieve a very competitive performance while maintaining enough flexibility to remain numerically robust.

The drawback of that approach was thus not lying in the achievable performance neither on the numerical robustness that could be ensured but in the maintanibilty and extensibility of the design. Indeed, the hardware versatility of modern supercomputers advocates for higher level programming paradigms. We have subsequently investigated the potential of task-based programming paradigms. We first proposed an MPI+task formulation. This formulation had the advantage of being relatively conservative with respect to the approach proposed earlier in the sense that it was still an MPI+X paradigm while versatile in the sense that we could abstract the hardware architecture, hence exploiting homogeneous platforms as well as heterogeneous machines enhanced with GPUs. To illustrate our discussion, we have designed a prototype implementation of an MPI+task version of MaPHyS. Although this prototype is working properly and showed the feasibility of the proposed approach, designing a solid MPI+task version of MaPHyS would require further work. First of all, the proposed approach still follows a bulk-synchronous parallelism. To overcome the subsequent synchronizations, it would be necessary to overlap symbolic preprocessing steps occurring within the internal PaStiX and Chameleon solvers. Second, except the application of the preconditioner, other operations occurring within the iterative part of the *solve step* step have not been implemented following a task-based internal design. Finally, to completely alleviate the synchronizations between the different sequences into which MaPHyS is decomposed, it would be necessary to further overlap communication with computation. This could be either performed with a clever usage of asynchronous MPI calls (but would certainly be hard to achieve at a fine-grain level) or by appending the MPI calls to the task flow so that the runtime system can dynamically decide when to perform the actual MPI calls to the MPI layer and interleave them with fine-grain computational tasks. Modern runtime systems such as StarPU provide such an opportunity and investigating such an approach is one of our main perspectives.

We have finally considered a full abstraction of the hardware architecture by considering a full task-based expression of the solver. After having discussed the missing ingredients towards a full task-based expression of a sparse hybrid solver, we have identified that one key component deserved to be first studied: the Krylov subspace method. Subsequently, we have proposed a full task-based formulation of the CG algorithm. We managed to refine the task flow up to removing almost all synchronizations of the classical CG algorithm, ensuring an efficient pipelining of the task flow. We showed that the considered approach was extremely efficient to achieve high performance on multiple types of modern architectures. We relied on the STF paradigm to design our prototype task-based CG solver. We showed that such a design could ensure a high productivity, making reasonable to implement advanced numerical scheme such as the pipelined CG formulation from [69] and subsequently achieving an even higher performance.

The potential bottleneck of the STF paradigm is that the dependencies are computed at runtime based on sequential consistency. As a consequence, a centralized view of the task flow needs to be built, which might be a bottleneck for going at scale. One possibility to overcome it would consist on another full task-based paradigm that explicitly encodes dependencies such as PTG discussed in Section 3.2.1.1. This approach is not considered in this thesis and will be addressed in future work relying on a modern runtime system that can support it such as PaRSEC.

At longer term, this thesis motivates for the implementation of a full task-based sparse hybrid solver and compare the underlying potential STF and PTG designs for extreme scale computing.

# Bibliography


[1] NVIDIA CUDA Compute Unified Device Architecture - Programming Guide. http://developer.download.nvidia.com, 2007.

[2] MAGMA Users' Guide, version 0.2. http://icl.cs.utk.edu/magma, November 2009.

[3] PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Architectures, Version 2.0. http://icl.cs.utk.edu/plasma, November 2009.

[4] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Julien Langou, Hatem Ltaief, and Stanimire Tomov. LU factorization for accelerator-based systems. In Howard Jay Siegel and Amr El-Kadi, editors, *The 9th IEEE/ACS International Conference on Computer Systems and Applications, AICCSA 2011, Sharm El-Sheikh, Egypt, December 27-30, 2011*, pages 217–224. IEEE, 2011.

[5] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Hatem Ltaief, Samuel Thibault, and Stanimire Tomov. QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators. In *IPDPS*, pages 932–943. IEEE, 2011.

[6] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In Wen mei W. Hwu, editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann, September 2010.

[7] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In Wen mei W. Hwu, editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann, September 2010.

[8] Emmanuel Agullo, Bérenger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner, and Toru Takahashi. Task-based FMM for multicore architectures. *SIAM J. Scientific Computing*, 36(1), 2014.

[9] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. Multifrontal QR factorization for multicore architectures over runtime systems. In *Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*, pages 521–532, 2013.

[10] Gabrielle Allen, Jaroslaw Nabrzyski, Edward Seidel, G. Dick van Albada, Jack Dongarra, and Peter M. A. Sloot, editors. *Computational Science - ICCS 2009, 9th International Conference, Baton Rouge, LA, USA, May 25-27, 2009, Proceedings, Part I*, volume 5544 of *Lecture Notes in Computer Science*. Springer, 2009.

[11] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, 2002.

[12] Marco Ament, Günter Knittel, Daniel Weiskopf, and Wolfgang Straßer. A parallel preconditioned conjugate gradient solver for the Poisson problem on a multi-GPU platform. In Marco Danelutto, Julien Bourgeois, and Tom Gross, editors, *PDP*, pages 583–592. IEEE Computer Society, 2010.

[13] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[14] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, 32(2):136–156, 2006.

[15] Patrick Amestoy, Iain S. Duff, Jean-Yves L'Excellent, and Xiaoye S. Li. Analysis and comparison of two general sparse solvers for distributed memory computers. *ACM Trans. Math. Softw.*, 27(4):388–421, 2001.

[16] Hartwig Anzt, Stanimire Tomov, Piotr Luszczek, William Sawyer, and Jack Dongarra. Acceleration of gpu-based krylov solvers via data transfer reduction. *IJHPCA*, 29(3):366–383, 2015.

[17] W. E. Arnoldi. The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Quart. Appl. Math.*, 9(1):17–29, 1951.

[18] C. Ashcraft. The fan-both family of column-based distributed Cholesky factorisation algorithm. In A. George, J.R. Gilbert, and J.W.H Liu, editors, *Graph Theory and Sparse Matrix Computations*, pages 159–190. Springer-Verlag NY, 1993.

[19] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.

[20] Cedric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.

[21] Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. An extension of the starss programming model for platforms with multiple GPUs. In *Euro-Par*, pages 851–862, 2009.

[22] Rosa M. Badia, José R. Herrero, Jesús Labarta, Josep M. Pérez, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. Parallelizing dense and banded linear algebra libraries using SMPSs. *Concurrency and Computation: Practice and Experience*, 21(18):2438–2456, 2009.

[23] Jacques Bahi, Raphaël Couturier, and Lilia Ziane Khodja. Parallel sparse linear solver GMRES for GPU clusters with compression of exchanged data. In *HeteroPar'11, 9-th Int. Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms*, LNCS, Bordeaux, France, August 2011. Springer. To appear.

[24] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. Curfman McInnes, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

[25] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, PA, second edition, 1994.

[26] Hélène Barucq, Rabia Djellouli, and Elodie Estecahandy. Efficient DG-like formulation equipped with curved boundary edges for solving elasto-acoustic scattering problems. *International Journal for Numerical Methods in Engineering*, 2014. To appear.

[27] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC*. ACM, 2009.

[28] M. Benzi, C. D. Meyer, and M. Tůma. A sparse approximate inverse preconditioner for the conjugate gradient method. *SIAM Journal on Scientific Computing*, 17:1135–1149, 1996.

[29] M. Benzi and M. Tůma. A sparse approximate inverse preconditioner for nonsymmetric linear systems. *SIAM Journal on Scientific Computing*, 19:968–994, 1998.

[30] P. Bjørstad and O. Widlund. Iterative methods for the solution of elliptic problems on regions partitioned into substructures. *SIAM Journal on Numerical Analysis*, 23(6):1097–1120, 1986.

[31] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM Press, 1997.

[32] Lionel Boillot. *Contributions to the mathematical modeling and to the parallel algorithmic for the optimization of an elastic wave propagator in anisotropic media*. Theses, Université de Pau et des Pays de l'Adour, December 2014. Collaboration Inria-Total.

[33] G. Bosilca, A. Bouteiller, A Danalis, M. Faverge, H. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra.

Distributed-Memory Task Execution and Dependence Tracking within DAGuE and the DPLASMA Project. *Innovative Computing Laboratory Technical Report*, 2010.

[34] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Hérault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, Piotr Luszczek, Asim YarKhan, and Jack Dongarra. Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA. In *IPDPS Workshops*, pages 1432–1441. IEEE, 2011.

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

[36] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Hérault, Pierre Lemarinier, and Jack Dongarra. DAGuE: A generic distributed DAG engine for high performance computing. *Parallel Computing*, 38(1-2):37–51, 2012.

[37] J.-F. Bourgat, R. Glowinski, P. Le Tallec, and M. Vidrascu. Variational formulation and algorithm for trace operator in domain decomposition calculations. In Tony Chan, Roland Glowinski, Jacques Périaux, and Olof Widlund, editors, *Domain Decomposition Methods*, pages 3–16, Philadelphia, PA, 1989. SIAM.

[38] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In IEEE, editor, *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Pisa, Italy, February 2010.

[39] Alfredo Buttari. Fine granularity sparse QR factorization for multicore based systems. In *Proceedings of the 10th international conference on Applied Parallel and Scientific Computing - Volume 2*, PARA'10, pages 226–236, Berlin, Heidelberg, 2012. Springer-Verlag.

[40] Alfredo Buttari. Fine-grained multithreading for the multifrontal QR factorization of sparse matrices. *SIAM J. Scientific Computing*, 35(4), 2013.

[41] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurrency and Computation: Practice and Experience*, 20(13):1573–1590, 2008.

[42] X.-C. Cai and Y. Saad. Overlapping domain decomposition algorithms for general sparse matrices. *Numerical Linear Algebra with Applications*, 3:221–237, 1996.

[43] L. M. Carvalho, L. Giraud, and G. Meurant. Local preconditioners for two-level non-overlapping domain decomposition methods. *Numerical Linear Algebra with Applications*, 8(4):207–227, 2001.

[44] Ali Cevahir, Akira Nukada, and Satoshi Matsuoka. Fast conjugate gradients with multiple GPUs. In Allen et al. [10], pages 893–903.

[45] Ali Cevahir, Akira Nukada, and Satoshi Matsuoka. High performance conjugate gradient solver on multi-GPU clusters using hypergraph partitioning. *Computer Science - R&D*, 25(1-2):83–91, 2010.

[46] F. Chaitin-Chatelin and V. Frayssé. *Lectures on Finite Precision Computations*. SIAM, Philadelphia, 1996.

[47] E. Chan, F. G. Van Zee, P. Bientinesi, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. A. van de Geijn. Supermatrix: a multithreaded runtime scheduling system for algorithms-by-blocks. In *PPOPP*, pages 123–132, 2008.

[48] Tony F. C. Chan and Tarek P. Mathew. The interface probing technique in domain decomposition. *SIAM J. Matrix Anal. Appl.*, 13(1):212–238, January 1992.

[49] Langshi Chen, Serge Petiton, Leroy Drummond, and Maxime Hugues. A communication optimization scheme for basis computation of krylov subspace methods on multi-gpus. In *High Performance Computing for Computational Science – VECPAR 2014*, volume 8969 of *Lecture Notes in Computer Science*, pages 3–16. Springer International Publishing, 2015.

[50] Yanqing Chen, Timothy A. Davis, William W. Hager, and Sivasankaran Rajamanickam. Algorithm 887: CHOLMOD, supernodal sparse cholesky factorization and update/downdate. *ACM Trans. Math. Softw.*, 35(3):22:1–22:14, October 2008.

[51] Zhangxin Chen, Hui Liu, and Bo Yang. Accelerating iterative linear solvers using multiple graphical processing units. *Int. J. Comput. Math.*, 92(7):1422–1438, 2015.

[52] C. Chevalier and F. Pellegrini. PT-SCOTCH: a tool for efficient parallel graph ordering. *Parallel Computing*, 34(6-8), 2008.

[53] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. Technical report, 2010.

[54] E. Chow. A priori sparsity patterns for parallel sparse approximate inverse preconditioners. *SIAM Journal on Scientific Computing*, 21(5):1804–1822, 2000.

[55] M. Cosnard and M. Loi. Automatic task graph generation techniques. In *System Sciences, 1995. Proceedings of the Twenty-Eighth Hawaii International Conference on*, volume 2, pages 113–122 vol.2, Jan 1995.

[56] Raphaël Couturier and Stéphane Domas. Sparse systems solving on GPUs with GMRES. *The Journal of Supercomputing*, 59(3):1504–1516, 2012.

[57] Y.-H. De Roeck and P. Le Tallec. Analysis and test of a local domain decomposition preconditioner. In Roland Glowinski, Yuri Kuznetsov, Gérard Meurant, Jacques Périaux, and Olof Widlund, editors, *Fourth International Symposium on Domain Decomposition Methods for Partial Differential Equations*, pages 112–128. SIAM, Philadelphia, PA, 1991.

[58] J. W. Demmel, J. R. Gilbert, and X. S. Li. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. *SIAM Journal on Matrix Analysis and Applications*, 20(4):915–952, October 1999.

[59] J. Drkošová, M. Rozložník, Z. Strakoš, and A. Greenbaum. Numerical stability of the GMRES method. *BIT*, 35:309–330, 1995.

[60] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, London, 1986.

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

[62] I. S. Duff and J. K. Reid. The multifrontal solution of unsymmetric sets of linear systems. *SIAM Journal on Scientific and Statistical Computing*, 5:633–641, 1984.

[63] Mathieu Faverge and Pierre Ramet. Dynamic scheduling for sparse direct solver on NUMA architectures. In *PARA'08*, LNCS, Trondheim, Norvège, 2008.

[64] J. Gaidamour and P. Hénon. A parallel direct/iterative solver based on a schur complement approach. *2013 IEEE 16th International Conference on Computational Science and Engineering*, 0:98–105, 2008.

[65] Thierry Gautier, Fabien Le Mentec, Vincent Faucher, and Bruno Raffin. X-kaapi: A multi paradigm runtime for multicore architectures. In *42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, October 1-4, 2013*, pages 728–735, 2013.

[66] Damien Genet, Abdou Guermouche, and George Bosilca. Assembly operations for multicore architectures using task-based runtime systems. In *Euro-Par 2014: Parallel Processing Workshops - Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part II*, pages 338–350, 2014.

[67] T. George, V. Saxena, A. Gupta, A. Singh, and A. R. Choudhury. Multifrontal factorization of sparse SPD matrices on GPUs. In *Proceedings of 25th International Parallel and Distributed Processing Symposium (IPDPS'11)*, pages 372–383, 2011.

[68] Serban Georgescu and Hiroshi Okuda. Conjugate gradients on multiple GPUs. *International Journal for Numerical Methods in Fluids*, 64(10-12):1254–1273, 2010.

[69] Pieter Ghysels and Wim Vanroose. Hiding global synchronization latency in the preconditioned conjugate gradient algorithm. *Parallel Computing*, 40(7):224–238, 2014.

[70] L. Giraud, A. Haidar, and L. T. Watson. Parallel scalability study of hybrid preconditioners in three dimensions. *Parallel Computing*, 34:363–379, 2008.

[71] L. Giraud and R. Tuminaro. Algebraic domain decomposition preconditioners. In F. Magoules, editor, *Mesh partitioning techniques and domain decomposition methods*, pages 187–216. Saxe-Coburg Publications, 2007.

[72] Luc Giraud and A. Haidar. Parallel algebraic hybrid solvers for large 3D convection-diffusion problems. *Numerical Algorithms*, 51(2):151–177, 2009.

[73] Luc Giraud, A. Haidar, and S. Pralet. Using multiple levels of parallelism to enhance the performance of domain decomposition solvers. *Parallel Computing*, 36(5-6):285–296, 2010.

[74] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[75] A. Greenbaum. *Iterative methods for solving linear systems*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1997.

[76] M. Grote and T. Huckle. Parallel preconditionings with sparse approximate inverses. *SIAM Journal on Scientific Computing*, 18:838–853, 1997.

[77] Anshul Gupta and Anshul Gupta. Wsmp: Watson sparse matrix package. Technical report, IBM, 2000.

[78] Azzam Haidar. *On the parallel scalability of hybrid linear solvers for large 3D problems*. PhD thesis, Institut National Polytechnique de Toulouse, December 17 2008.

[79] T. D. R. Hartley, E. Saule, and Ü. V. Çatalyürek. Improving performance of adaptive component-based dataflow middleware. *Parallel Computing*, 38(6-7):289–309, 2012.

[80] Kai He, Sheldon X.-D. Tan, Hengyang Zhao, Xue-Xin Liu, Hai Wang, and Guoyong Shi. Parallel GMRES solver for fast analysis of large linear dynamic systems on GPU platforms. *Integration, the VLSI Journal*, 52:10 – 22, 2016.

[81] P. Hénon, P. Ramet, and J. Roman. PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. *Parallel Computing*, 28(2):301–321, January 2002.

[82] Pascal Hénon and Yousef Saad. A parallel multistage ILU factorization based on a hierarchical graph decomposition. *SIAM J. Sci. Comput.*, 28(6):2266–2293, December 2006.

[83] M. Heroux. AztecOO user guide. Technical Report SAND2004-3796, Sandia National Laboratories, Albuquerque, NM, 87185, 2004.

[84] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the trilinos project. *ACM Transactions on Mathematical Software*, 31(3):397–423, September 2005.

[85] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear system. *J. Res. Nat. Bur. Stds.*, B49:409–436, 1952.

[86] D. Higham and N. Higham. Structured backward error and condition of generalized eigenvalue problems. *SIAM Journal on Matrix Analysis and Applications*, 20(2):493–512, 1998.

[87] N. J. Higham. *Accuracy and Stability of Numerical Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[88] P. Jolivet. *Méthodes de décomposition de domaine. Application au calcul haute performance.* PhD thesis, Université de Grenoble, 2014.

[89] Laxmikant V. Kalé and Sanjeev Krishnan. CHARM++: A portable concurrent object oriented system based on c++. In *OOPSLA*, pages 91–108, 1993.

[90] G. Karypis and V. Kumar. METIS – *Unstructured Graph Partitioning and Sparse Matrix Ordering System – Version 2.0.* University of Minnesota, June 1995.

[91] L. Yu Kolotilina, A. Yu. Yeremin, and A. A. Nikishin. Factorized sparse approximate inverse preconditionings. III: Iterative construction of preconditioners. *Journal of Mathematical Sciences*, 101:3237–3254, 2000. Originally published in Russian in *Zap. Nauchn. Semin. POMI*, 248:17-48, 1998.

[92] D. M. Kunzman and L. V. Kalé. Programming heterogeneous clusters with accelerators using object-based programming. *Scientific Programming*, 19(1):47–62, 2011.

[93] Jakub Kurzak and Jack Dongarra. Fully dynamic scheduler for numerical computing on multicore processors. *LAPACK working note*, lawn220, 2009.

[94] Jakub Kurzak, Hatem Ltaief, Jack Dongarra, and Rosa M. Badia. Scheduling dense linear algebra operations on multicore processors. *Concurrency and Computation: Practice and Experience*, 22(1):15–44, 2010.

[95] X. Lacoste. *Scheduling and memory optimizations for sparse direct solver on multicore/multi-gpu cluster systems.* PhD thesis, LaBRI, Université Bordeaux, Talence, France, February 2015.

[96] C. Lanczos. Solution of systems of linear equations by minimized iterations. *J. Res. Nat. Bur. Stand.*, 49:33–53, 1952.

[97] Jean-Yves L'Excellent and Wissam M. Sid-Lakhdar. A study of shared-memory parallelism in a multifrontal solver. *Parallel Computing*, 40(3-4):34–46, 2014.

[98] Ruipeng Li, Hector Klie, Hari Sudan, and Yousef Saad. Towards Realistic Reservoir Simulations on Manycore Platforms. *SPE Journal*, pages 1–23, 2010.

[99] X. S. Li, M. Shao, I. Yamazaki, and E. G. Ng. Factorization-based sparse solvers and preconditioners. *Journal of Physics: Conference Series*, 180(1):012015, 2009.

[100] Xiaoye S. Li and James W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Math. Softw.*, 29(2):110–140, June 2003.

[101] Z. Li, Y. Saad, and M. Sosonkina. pARMS: A parallel version of the algebraic recursive multilevel solver. Technical Report Technical Report UMSI-2001-100, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, 2001.

[102] R. J. Lipton, Donald J. Rose, and Robert Endre Tarjan. Generalized Nested Dissection. *SIAM Journal on Numerical Analysis*, 16(2), 1979.

[103] Florent Lopez. *Task-based multifrontal QR solver for heterogeneous architectures.* PhD thesis, University Paul Sabatier, Toulouse, France, 2015. submitted.

[104] Hatem Ltaief and Rio Yokota. Data-driven execution of fast multipole methods. *CoRR*, abs/1203.0889, 2012.

[105] Robert F. Lucas, Gene Wagenbreth, Dan M. Davis, and Roger Grimes. Multifrontal computations on GPUs and their multi-core hosts. In *Proceedings of the 9th international conference on High performance computing for computational science*, VECPAR'10, pages 71–82, Berlin, Heidelberg, 2011. Springer-Verlag.

[106] C.-K. Luk, S. Hong, and H. Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *MICRO*, pages 45–55, 2009.

[107] T. Mathew. *Domain Decomposition Methods for the Numerical Solution of Partial Differential Equations.* Springer Lecture Notes in Computational Science and Engineering. Springer, 2008.

[108] G. Meurant. *The Lanczos and conjugate gradient algorithms: from theory to finite precision computations.* Software, Environments, and Tools 19. SIAM, Philadelphia, PA, USA, 2006.

[109] G. L. Miller and S. A. Vavasis. Density graphs and separators. In *Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 331–336, 1991.

[110] Alexander Monakov and Arutyun Avetisyan. Implementing blocked sparse matrix-vector multiplication on NVIDIA GPUs. In Koen Bertels, Nikitas J. Dimopoulos, Cristina Silvano, and Stephan Wong, editors, *SAMOS*, volume 5657 of *Lecture Notes in Computer Science*, pages 289–297. Springer, 2009.

[111] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In Yale N. Patt, Pierfrancesco Foglia, Evelyn Duesterwald, Paolo Faraboschi, and Xavier Martorell, editors, *HiPEAC*, volume 5952 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2010.

[112] A. Munshi. The OpenCL specification, khronos opencl working group, version 1.1, revision 44, 2011.

[113] Tomás Oberhuber, Atsushi Suzuki, and Jan Vacata. New row-grouped CSR format for storing the sparse matrices on GPU with implementation in CUDA. *CoRR*, abs/1012.2270, 2010.

[114] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[115] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[116] C. Paige, M. Rozložník, and Z. Strakoš. Modified Gram-Schmidt (MGS), least-squares, and backward stability of MGS-GMRES. *SIAM Journal on Matrix Analysis and Applications*, 28(1):264–284, 2006.

[117] Forum Message Passing. MPI: A message-passing interface standard. Technical report, University of Tennessee, Knoxville, TN, USA, 1994.

[118] F. Pellegrini and J. Roman. Sparse matrix ordering with SCOTCH. In *Proceedings of HPCN'97, Vienna, LNCS 1225*, pages 370–378, April 1997.

[119] A. Quarteroni and A. Valli. *Domain decomposition methods for partial differential equations*. Numerical mathematics and scientific computation. Oxford science publications, Oxford, 1999.

[120] G. Quintana-Ortí, E. S. Quintana-Ortí, E. Chan, F. G. Van Zee, and R. A. van de Geijn. Scheduling of QR factorization algorithms on SMP and multi-core architectures. In *Proceedings of PDP'08*, 2008. FLAME Working Note #24.

[121] Gregorio Quintana-Ortí, Francisco D. Igual, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. *ACM SIGPLAN Notices*, 44(4):121–130, April 2009.

[122] G. Radicati and Y. Robert. Parallel conjugate gradient-like algorithms for solving nonsymmetric linear systems on a vector multiprocessor. *Parallel Computing*, 11:223–239, 1989.

[123] S. Rajamanickam, E. G. Boman, and M. A. Heroux. ShyLU: A hybrid-hybrid solver for multicore platforms. *Parallel and Distributed Processing Symposium, International*, 0:631–643, 2012.

[124] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007.

[125] Steven C. Rennich, Darko Stosic, and Timothy A. Davis. Accelerating sparse cholesky factorization on gpus. In *Proceedings of the Fourth Workshop on Irregular Applications: Architectures and Algorithms*, IA3 '14, pages 9–16, Piscataway, NJ, USA, 2014. IEEE Press.

[126] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Comput.*, 14:461–469, 1993.

[127] Y. Saad. ILUT: A dual threshold incomplete LU factorization. *Numerical Linear Algebra with Applications*, 1:387–402, 1994.

[128] Y. Saad. *Iterative Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[129] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comp.*, 7:856–869, 1986.

[130] Y. Saad and H. A. van der Vorst. Iterative solution of linear systems in the 20-th century. Tech. Rep. UMSI-99-152, University of Minnesota, 1999.

[131] Piyush Sao, Xing Liu, Richard Vuduc, and Xiaoye Li. A sparse direct solver for distributed memory xeon phi-accelerated systems. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 71–81, May 2015.

[132] Piyush Sao, Richard W. Vuduc, and Xiaoye Sherry Li. A distributed CPU-GPU sparse direct solver. In *Euro-Par 2014 Parallel Processing*, pages 487–498, 2014.

[133] Olaf Schenk, Klaus Gärtner, Wolfgang Fichtner, and Andreas Stricker. PARDISO: A high-performance serial and parallel sparse linear solver in semiconductor device simulation, 2000.

[134] H. A. Schwarz. Über eine grenzübergang durch alternirendes verfahren. *Gesammelete Mathematische Abhandlungen, Springer-Verlag*, 2:133–143, 1890. First published in Vierteljahrsschrift der Naturforschenden Gesellschaft in Zürich, vol.15, pp. 272-286, 1870.

[135] B. F. Smith, P. Bjørstad, and W. Gropp. *Domain Decomposition, Parallel Multilevel Methods for Elliptic Partial Differential Equations.* Cambridge University Press, New York, 1st edition, 1996.

[136] Fengguang Song, Asim YarKhan, and Jack Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC'09*, 2009.

[137] P. Le Tallec, Y.-H. De Roeck, and M. Vidrascu. Domain-decomposition methods for large linearly elliptic three dimensional problems. *J. of Computational and Applied Mathematics*, 34:93–117, 1991.

[138] H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, Mar 2002.

[139] Sean Treichler, Michael Bauer, and Alex Aiken. Realm: an event-based low-level runtime for distributed memory architectures. In *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, pages 263–276, 2014.

[140] R. S. Tuminaro, M. Heroux, S. Hutchinson, and J. Shadid. Official Aztec user's guide: Version 2.1. Technical Report Sand99-8801J, Sandia National Laboratories, Albuquerque, NM, 87185, Nov 1999.

[141] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[142] Field G. Van Zee, Ernie Chan, Robert A. van de Geijn, Enrique S. Quintana-Orti, and Gregorio Quintana-Orti. The `libflame` Library for Dense Matrix Computations. *Computing in Science and Engineering*, 11(6):56–63, November/December 2009.

[143] Mickeal Verschoor and Andrei C. Jalba. Analysis and performance estimation of the conjugate gradient method on multiple GPUs. *Parallel Computing*, 38:552 – 575, 2012.

[144] Mingliang Wang, Hector Klie, Manish Parashar, and Hari Sudan. Solving sparse linear systems on NVIDIA Tesla GPUs. In Allen et al. [10], pages 864–873.

[145] W.A. Wiggers, V. Bakker, A.B.J. Kokkeler, and G.J.M. Smit. Implementing the Conjugate Gradient algorithm on multi-core systems. In *System-on-Chip, 2007 International Symposium on*, pages 1 –4, nov. 2007.

[146] J. H. Wilkinson. *Rounding Errors in Algebraic Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, 1963.

[147] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 38:1–38:12, New York, NY, USA, 2007. ACM.

[148] Ichitaro Yamazaki, Hartwig Anzt, Stanimire Tomov, Mark Hoemmen, and Jack J. Dongarra. Improving the performance of CA-GMRES on multicores with multiple gpus. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pages 382–391, 2014.

[149] Ichitaro Yamazaki and Xiaoye S. Li. On techniques to improve robustness and scalability of a parallel hybrid linear solver. In *VECPAR*, pages 421–434, 2010.

[150] Ichitaro Yamazaki, Sivasankaran Rajamanickam, Erik G. Boman, Mark Hoemmen, Michael A. Heroux, and Stanimire Tomov. Domain decomposition preconditioners for communication-avoiding krylov methods on a hybrid CPU/GPU cluster. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*, pages 933–944, 2014.

[151] Chenhan D. Yu, Weichung Wang, and Dan'l Pierce. A CPU-GPU hybrid approach for the unsymmetric multifrontal method. *Parallel Comput.*, 37:759–770, December 2011.

[152] Lilia Ziane Khodja, Raphaël Couturier, Arnaud Giersch, and JacquesM. Bahi. Parallel sparse linear solver with gmres method using minimization techniques of communications for gpu clusters. *The Journal of Supercomputing*, 69(1):200–224, 2014.

[153] Dan Zou, Yong Dou, Song Guo, Rongchun Li, and Lin Deng. Supernodal sparse cholesky factorization on graphics processing units. *Concurrency and Computation: Practice and Experience*, 26(16):2713–2726, 2014.