N° d'ordre : XXXX

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Damien Genet**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Conception et réalisation d'un solveur pour les problèmes de dynamique des fluides pour les architectures many-core

---

Design of Generic Modular Solutions for PDE solvers for Modern Architectures

---

Soutenue le : 12 décembre 2014

Après avis des rapporteurs :

Patrick AMESTOY	Professeur, ENSEEIHT
Tiago QUINTINO	Data Handling Team Leader, ECMWF

Devant la commission d'examen composée de :

Frédéric DESPREZ	Directeur de recherche, INRIA	Examinateur
Abdou GUERMOUCHE	MdC, Université de Bordeaux	Examinateur
Thomas HÉRAULT ...	Research scientist, University of Tennessee	Examinateur
François PELLEGRINI	Professeur, Université de Bordeaux	Directeur de Thèse
Mario RICCHIUTO ...	Chargé de recherche (HdR), INRIA	Directeur de Thèse
Jean ROMAN	Professeur, ENSEIRB-MATMECA	Président du jury

– 2014 –

Remerciements

Tout d'abord, je voudrai remercier mes relecteurs pour avoir accepter de relire le document et m'avoir fourni des retours constructifs qui m'ont permis d'améliorer le manuscrit. Je tiens a remercier les membres du jury d'avoir accepter de participer à ma soutenance de thèse. Merci à eux d'avoir fait le déplacement.

Merci à mes directeurs de thèse de m'avoir encadré jusqu'à l'aboutissement de ces travaux. Merci à Abdou, Mario et François. Tout ne fut pas rose tous les jours, mais tout finit bien. C'est surement le plus important. Je ne remercierai jamais assez Rémi de m'avoir permis de faire cette thèse et d'avoir pris de son temps pour répondre à mes questions.

Merci à ceux qui ont travaillé directement sur la plateforme avec moi. Je pense à François Rue, Maxime, Dragan. Un grand merci à Vincent et Mario qui m'ont accordé de leur temps pour m'expliquer les schémas numériques.

Après ces "quelques" années à Bordeaux, et chez Inria, (ou à l'INRIA) j'ai eu l'occasion de croiser plein de collègues qui sont devenus des amis. Des générations de thśards et de stagiaires, des chercheurs, et les autres. Tout d'abord, je voudrai remercier Josy, Anne-Laure, et Nicolas, les assistantes de l'équipe grace à qui j'ai pu remplir toutes cette papperasse pour les missions.

Merci aux différentes équipes que j'ai croisé. D'abord Bacchus, chez qui j'ai fait cette thèse. Merci à la branche italienne de l'équipe, Pietro, Maria-Giovanna, Andrea, Luca, Paola, et Mario. Merci aux membres moins italiens de l'équipe, Cécile, Héloise, Mathieu, Stevan, Leo, Quentin, Kunkun. Merci aux membres de HiePACS, la première équipe où j'ai séjourné, Jean, Luc, Olivier, Pierre, Mathieu. Merci à Manu de m'avoir permis d'aller au États-Unis, c'était une excellente expérience. Merci aux membres de Runtime avec qui j'ai pu échanger, Raymond, Pierre-André, Nathalie, Brice.

Merci à Robin, Adam (Laden), Christelle, Mathieu F, Nico. Vous êtes des modèles pour moi ;), j'ai décidé de faire cette thèse en me basant sur vos expériences, ca en dit long sur mon état...

Merci à Cacal d'avoir éte co-bureau avec moi sur la fin. On a pu souffrir ensemble sur la dernière ligne droite, et on a bien rigolé.

Et maintenant, tout ceux sans qui ces années sur Bordeaux auraient été très différentes, Abdou, Orel, Mario, Anna, Cacal, Manu, Marianne, Méli-mélo, Nico, Patoche, Robin. Merci à vous pour toutes les soirées, les bons restos, les moins bons, et tous les bons moments qu'on a vécu.

Et enfin, je tiens a remercier mes parents sans qui, bien évidemment, rien de tout ca n'aurait été possible. Merci de m'avoir soutenu pendant cette période, merci d'avoir été à l'écoute quand j'ai eu besoin. Je vous aime, vous êtes les meilleurs.

Conception et réalisation d'un solveur pour les problèmes de dynamique des fluides pour les architectures many-core

Résumé :

La CFD, ou simulation de dynamique des fluides est l'étude de système impliquant des écoulements fluides, des transferts de chaleur, ou des réactions chimiques en utilisant des ordinateurs. Ces simulations se font sur des systêmes discrets très grands pour atteindre un haut niveau de précision, et requiert un grand nombre de ressources de calcul. Ces applications, si elles devaient s'exécuter sur un ordinateur de bureau, prendraient des mois, voire des années pour s'achever, si tant est que le problème puisse tenir dans la mémoire dudit ordinateur.

Dans ce contexte où un grand nombre de ressources de calcul sont nécessaires, nous avons à disposition depuis quelques décennies le concept du HPC, ou calcul haute performance. Cela consiste à mettre en commun un ensemble de ressources de calculs pour accélérer les applications. Les ordinateurs ont beaucoup évolué au fil des années, et les architectures actuelles combinent un grand nombre de noeuds de calcul. Chacun de ces noeuds peut posséder un ou plusieurs accélérateurs de calculs. Programmer de telles architectures est devenu un réel problème ; dans la mesure où le materiel évolue vite, maintenir les performances de son application pour chaque nouvelle génération de matériel est coûteux en temps.

Dans ce contexte, cette thèse se propose de concevoir et développer une nouvelle plateforme pour la résolution des équations aux dérivées partielles de la mécanique des fluides. Dans un premier temps, nous présenterons le contexte mathématique, les méthodes numérique que nous envisageons d'intégrer à la plateforme. Puis, dans un second temps, nous ferons un inventaire de quelques plateforme proches en terme de fonctionnalités avant de nous pencher sur la definition des besoins fonctionnels et techniques. Enfin, nous présenterons notre étude d'un problème récurrent dans les applications numérique, le problème d'assemblage. Puis nous concluerons en dressant un bilan de nos travaux et évoquerons quelques pistes d'ameliorations en guise de perspectives.

Contexte général, et motivations

L'objectif de ce chapitre est d'introduire les méthodes aux éléments finis, d'identifier les différences entre ces méthodes pour pouvoir généraliser et ainsi intégrer les différentes approches à la plateforme de manière transparente.
Pour comprendre le type de problème que nous cherchons à résoudre, voici un exemple. Nous cherchons u une fonction réelle, scalaire, définie sur un domaine Ω, sous-ensemble de l'espace $\mathbb{R}^d$, satisfaisant le problème suivant :

$$\begin{cases} \nabla \cdot (\vec{\beta} u) - \nabla \cdot (\nu \nabla u) + \sigma u = f & \text{in } \Omega \\ \hat{n} \cdot \vec{\beta}\, u + \nu \nabla u \cdot \hat{n} = 0 & \text{on } \partial\Omega \end{cases}, \tag{1}$$

avec f une fonction réelle définie sur Ω, $\vec{\beta}$ un vecteur constant de $\mathbb{R}^d$, et $\hat{n} \in \mathbb{R}^d$, la normale unitaire sortante de $\partial\Omega$, et avec σ et ν des constantes, telles que $\nu \geq 0$. Cette équation est une version simplifiée de ce que l'on peut trouver dans des applications faisant du transfert thermique, ou du transport de masse.

Pour discrétiser notre espace, nous avons deux possibilités, des éléments finis continus, ou discontinus. Les premiers distribuent les inconnus sur tous l'élément. Cela veut dire que des éléments vont partager des informations. Les seconds vont garder l'ensemble de leurs inconnues à l'intérieur de l'élément, et ne rien partager avec les autres. Première conséquence, en partageant des informations les premiers consomment moins de mémoire mais le couplage entre les inconnus rend l'ordonnancement des calculs plus compliqué en parallèle. Les seconds ont l'avantage de rendre leur calculs locaux à l'élément, mais ils consommeront plus de mémoire et nécessiteront une phase supplémentaire pour gérer la discontinuité aux interfaces entre les éléments.

Enfin, le problème est transformé en trois étapes. Une formulation variationnelle de l'équation, l'application des conditions aux bords, et enfin la substitution de la solution discrète à la solution continue. Si l'on a choisi d'utiliser des éléments finis discontinus, la troisième étape est réalisée d'abord, puis une formulation variationnelle discrète est faite, faisant apparaitre un flux numérique aux bords de chaque éléments.

Les challenges auxquels nous devons faire face sont les suivants :
- la gestion mémoire de deux méthodes aux éléments finis différentes ;
- le caractère hétérogène des données dû à la présence dans le maillage d'élément de formes différentes, de degrés différents ;
- l'exploitation des architectures modernes aux hiérarchies mémoires complexes.

Conception de la plateforme

L'utilisation d'outils de simulation numérique pour comprendre les phénomènes physiques complexes est critique. Il faut être en mesure de remplacer de coûteuses expérimentations par un résultat de simulation très précis obtenu le plus rapidement possible. La précision s'obtenant en augmentant la taille du système, et donc en augmentant les besoins en terme de ressource de calculs. Nous allons donc considérer dans cette partie la conception d'une plateforme permettant des calculs de haute precision et exploitant les machines de calcul complexes, cela afin de terminer l'exécution de l'application le plus tôt possible. Les besoins que nous avons pour notre plateforme sont multiples :

Maillage. Pour avoir la plateforme la plus générique possible, nous devons supporter des maillages hybrides, en toutes dimensions, éléments courbes ou non.

Méthodes. Dans un premier temps, nous nous intéressons à deux classes de méthodes, les méthodes aux éléments finis continus, telles que Galerkin continu, et les methodes aux éléments finis discontinus telles que Galerkin discontinu.

Éléments finis. Pour obtenir une solution la plus précise possible, nous aurons besoin de supporter les éléments fini d'ordre élévé.

Performance. Dans un monde où les machines évoluent très rapidement, nous avons besoin de maintenir les performances de notre plateforme à mesure que les machines changent.

Un tableau comparatif (voir Section 2.1, page 22) présente plusieurs plateformes faisant de la simulation numérique, ou permettant de construire une application de simulation numérique, utilisant un langage orienté objet pour la modularité que cela apporte. Certaines de ces plateforme mélangent des paradigmes de programmation pour gérer les différents niveaux de parallélisme. Certaines génèrent même les noyaux de calcul pendant l'exécution en fonction des besoins de l'application.

Nous avons choisi de concevoir une nouvelle plateforme pour pouvoir utiliser les différents outils développés dans notre institut de recherche ou dans les équipes collaborant avec nous. Cela va

nous permettre de rester proche de la plateforme, ce qui n'aurait pas été forcément possible si nous avions choisi de partir d'une plateforme existante.

Nos besoins fonctionnels pour la plateforme `AeroSol` sont les suivants :

Contribution et évolution. La prise en main de la plateforme doit être le plus simple possible pour permettre à des collaborateurs de divers champs d'expertise de contribuer. En cloisonnant les fonctionnalités de la plateforme, chaque module sera interchangeable, et l'évolutivité de la plateforme sera assurée.

Opérations élémentaires. Pour pouvoir gérer nimporte quel type de maillage, quelque soit la dimension du problème, les éléments présents dans le maillage, le degré de la solution, quelque soir la méthode d'éléments finis employée, nous avons besoin de trouver une représentation de nos éléments qui permettent d'optimiser la gestion de la mémoire.

Rafinement. Certaines méthodes numériques font du rafinement de maillage. Ceci est une contrainte supplémentaire pour notre plateforme. Le rafinement permet d'augmenter la précision de la solution sur les zones du maillage où un phénomène se produit, ou au contraire, de réduire la précision là où plus rien ne se produit.

En terme de besoin technique, nous avons principalement besoin d'abstraire l'architecture matérielle sous-jacente pour pouvoir nous exécuter sur les machine modernes et pouvoir assurer la portabilité des performances.

Opération d'assemblage

Dans ce chapitre, nous étudions l'opération d'assemblage. Cette opération se retrouve dans divers applications scientifiques, dans des solvers linéaires basés sur des méthodes multi-frontale ou dans des applications de simulation numérique. Puis nous introduirons les supports d'exécution sur lesquels nous avons implanté nos strategies pour l'opération d'assemblage, et présenterons la base de nos stratégies.

Le principe de l'opération d'assemblage est le suivant : un ensemble de blocs de contributions doit être ajouté dans le systême global, ces blocs présentent des zones de recouvrement entre eux, ce qui oblige à séquentialiser l'assemblage de ces blocs dans la structure globale.

Dans le cadre de méthode d'éléments finis, la première technique pour résoudre ce problème consiste à faire de la coloration de graphe. Après avoir déterminer comment les informations sont echangées entre les éléments, par exemple, par les sommets, nous allons prendre chaque sommet du maillage, prendre la liste des éléments possédant ce sommet, et leur attribuer une couleur différente. Une fois tous les éléments coloriés, nous allons construire des listes dans lesquelles nous allons ranger tous les éléments de la même couleur. Nous avons donc dans une liste un ensemble d'éléments pouvant être traités en parallèle. L'opération d'assemblage devient donc un traitement séquentiel des listes, nous traitons les listes une par une, et un assemblage en parallèle des éléments d'une liste. Une deuxième approche, toujours basée sur du coloriage, consiste á colorier les lignes de chaque bloc de contribution plutot que le bloc entier. Certes, á un grain plus fin, on obtient un ordonnancement de qualite, mais cela oblige á disposer de l'ensemble des blocs de contribution dès le début de l'opération d'assemblage.
Le dévelopement d'applications scientifiques pour le calcul haute performance consiste en général à mélanger un ensemble de paradigmes de programmation. Une bibliothèque pour faire du passage de message (MPI) entre les noeuds de calcul et une bibliothèque pour exploiter les processeurs multi-coeurs (OpenMP). Si certains noeuds disposent d'accélérateurs de calculs, il faut rajouter une bibliothèque pour communiquer avec l'accélérateur (CUDA). Développer et

maintenir les performances de ces applications devient très compliqué au fur et à mesure que les machines se complexifient. Une approche pour résoudre ce problème, consiste à passer par un niveau intermédiaire, le support d'exécution. Les supports d'exécution sont des bibliothèques à qui l'on va soumettre du travail (des tâches de calculs), et qui va s'occuper d'ordonnancer les tâches soumises sur la machine de calcul. Au prix d'une re-écriture de l'algorithme sous forme d'un graphe acyclique dirigé (DAG), et de l'implantation des différentes tâches pour les différentes unités d'exécution, le support d'exécution va être capable d'ordonnancer ces tâches sur la machine, il va prendre en charge tous les transferts mémoire et assurer la cohérence des données.

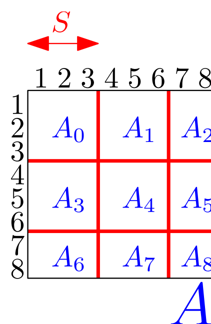


FIGURE 1 – Découpage de la structure globale.

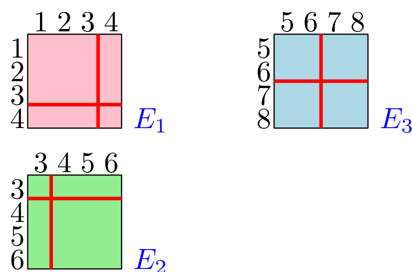


FIGURE 2 – Découpage des blocs de contribution.

Pour adapter l'opération d'assemblage aux supports d'exécution, nous avons dû changer l'approche. Au lieu de chercher á paralléliser les blocs, nous avons choisi de faire apparaitre du parallélisme dans la structure globale en découpant cette structure en blocs comme représenté dans la Figure 1. Puis nous avons projeté ce découpage sur l'ensemble des blocs de contribution comme montré dans la Figure 2. Une fois les blocs decoupés, il est possible de chainer les blocs contribuant au même endroit dans la structure globale, faisant ainsi apparaitre du parallélisme entre les chaînes à assembler.
Nos stratégies ont été testées sur deux cas test. Un issu du solveur à méthode multi-frontale `MUMPS`, et l'autre simulant l'assemblage d'une méthode d'éléments finis. Dans le premier cas, les méthodes de coloration ne peuvent pas fonctionner car la matrice d'adjacence des blocs n'est pas facilement exploitable et les blocs ont des tailles variant de 1 á 100% de la taille de la structure globale, avec des recouvrements très irréguliers.

Conclusion

Nous avons présenté une nouvelle plateforme, `AeroSol`. Sur les aspects paralélisation, la plateforme travaille à plusieurs niveaux. Tout d'abord, une décomposition de domaine, que nous avons délégué à une bibliothèque extérieure, et au niveau mémoire partagée par le biais de

l'utlisation de supports d'exécution. Cela nous permettra aussi d'assurer une portabilité des performances à un coût moindre par rapport á un scénario où nous aurions dû mettre à jour le cœur de la plateforme à chaque nouveau matériel.

La plateforme gère de manière transparente les méthodes aux éléments finis continus ou discontinus. Elle gère des maillages en toute dimension, dispose d'une large bibliothèque d'éléments finis, de formule de quadrature, et de fonctions géométriques.

Notre plateforme `AeroSol` est maintenant utilisée par plusieurs équipes de recherche.

Nos perspectives sont de continuer à améliorer les performances de la plateforme. Nous avons besoin de pousser plus loin le portage sur les supports d'exécution. Á terme nous espérons pouvoir exprimer l'intégralité de la simulation en graphe de tâches et ainsi en s'interfaçant avec un solveur linéaire lui aussi porté sur support d'exécution pour relâcher des points de synchronisation.

Nous avons l'ambition de rendre l'outil plus accessible aux collaborations en ajoutant une couche supplémentaire pour un langage dédié (DSL). Cela permettrait d'écrire moins de code pour faire une application.

Mots clés :

Calcul haute performance, Éléments finis, Support d'exécution

Domaine :

Informatique, Calcul haute performance

Équipe projet Inria BACCHUS [1],
Inria Bordeaux Sud-Ouest,
200, avenue de la vieille tour
33405 Talence Cedex, FRANCE

1. http://www.bacchus.bordeaux.inria.fr .

Design of Generic Modular Solutions for PDE solvers for Modern Architectures

Abstract :

Numerical simulation is nowadays an essential part of engineering analysis, be it to design a new plane, or to detect underground oil reservoirs. Numerical simulations have indeed become an important complement to theoretical and experimental investigation, allowing one to reduce the cost of engineering design processes. In order to achieve a high level of precision, one need to increase the resolution of his computational domain. So to keep getting results in reasonable time, one shall find a way to speed-up computations. To do this, we use high performance computing, HPC, to exploit the complex architecture of modern supercomputers. Under these two constraints, and some other like the genericity of finite elements, or the mesh dimension, we developed a new platform `AeroSol`. In this thesis, we present the mathematical background, and the two types of schemes that are implemented in the platform, the continuous finite elements method, and the discontinuous one. Then, we present the design choices made in the platform, then, we study a sub-problem, the assembly operation, which can be found in linear algebra multi-frontal methods.

Keywords :

HPC, Finite Element methods, Runtime system

Discipline :

Computer science

Inria project-team BACCHUS [2],
Inria Bordeaux Sud-Ouest,
200, avenue de la vieille tour
33405 Talence Cedex, FRANCE

2. http://www.bacchus.bordeaux.inria.fr .

Table des matières

General Introduction **1**

1 Setting and main challenges **5**

1.1 General Context and motivation . . . 5

1.2 Model problem . . . 5

1.3 Mesh and finite element approximation . . . 6

1.3.1 Finite elements . . . 7

1.4 Discrete equations . . . 9

1.4.1 Integrals and matrices . . . 11

1.4.2 Time dependent equation . . . 14

1.5 Challenges . . . 15

1.5.1 Abstracting the memory layout of finite element methods . . . 15

1.5.2 Handling hybrid meshes . . . 16

1.5.3 Mesh adaptation . . . 17

1.5.4 Abstract the complex architecture . . . 18

2 Software architecture **19**

2.1 Introduction . . . 20

2.2 Our needs . . . 21

2.3 Related Works . . . 21

2.3.1 Frameworks . . . 22

2.3.2 `Aghora` . . . 22

2.3.3 `deal.II` . . . 22

2.3.4 `hpGEM` . . . 23

2.3.5 `OP2` . . . 23

2.3.6 `COOLFluiD` . . . 23

2.3.7 `freefem++` . . . 23

2.3.8 `HiFlow`^{3} . . . 23

2.3.9 `PyCUDA` . . . 24

2.3.10 Discussion . . . 24

2.4 General Architecture . . . 24
2.4.1 Functional requirements . . . 24
2.4.2 Proposed architecture . . . 25
2.4.3 Technical requirements . . . 26
2.5 Delegating the functionalities . . . 27
2.5.1 PaMPA . . . 27
2.5.2 Runtime systems . . . 29
2.5.3 Linear Solver . . . 29
2.6 Functional implementation . . . 30
2.6.1 Specializing the entities . . . 30
2.6.2 Solution Object . . . 31
2.6.3 Cell reconstruction . . . 32
2.7 Computational implementation . . . 34
2.7.1 Finite elements collection . . . 34
2.7.2 Equations of state, models and, numerical fluxes . . . 35
2.7.3 Integrator classes . . . 35
2.7.4 Spatial scheme . . . 35
2.7.5 Time scheme . . . 35
2.8 Experiments and results . . . 36
2.8.1 Numerical tests . . . 36
2.8.2 Yee vortex . . . 36
2.9 Conclusion . . . 39

3 The assembly operation **41**
3.1 Introduction . . . 41
3.2 Related Works . . . 42
3.3 The StarPU runtime system . . . 45
3.4 PaRSEC . . . 46
3.5 Finite Element Method . . . 47
3.5.1 Computation of contributions . . . 48
3.5.2 Taskified Finite Element Method using StarPU . . . 49
3.5.3 Taskified Finite Element Method using PaRSEC . . . 50
3.6 The assembly operation . . . 50
3.6.1 A new taskified Assembly Operation . . . 56
3.6.2 Scheduling strategies for taskified assembly operations . . . 58
3.7 Experimental results . . . 59
3.7.1 Experimental setup . . . 59
3.8 Conclusion . . . 61

General Conclusion and perspectives **69**

Bibliography **71**

Personal Publications **77**

Liste des tableaux

2.1 Comparison of state-of-the-art softwares solving PDE problems using finite element methods ("N/A" : information not available from the literature). 22
2.2 Values corresponding to element shapes. 31
2.3 Values corresponding to domain conditions. 31
2.4 Yee vortex problem : time/proc. [s] with Aerosol and Aghora for 3000 Runge-Kutta sub-steps. 37
2.5 Yee vortex problem : relative costs in percent of different stages of the numerical algorithm per physical time step. 38
2.6 Comparison of execution time by core for a polynomial degree equal to 2 (9000 RK sub-steps). We observe that the most efficient execution is obtained for the OpenMPI library on Avakas. 38
2.7 Average elapsed time [s] required for initialization of the MPI universe. 39

Table des figures

1 Découpage de la structure globale. . . . viii
2 Découpage des blocs de contribution. . . . viii

1.1 Unstructured mesh of two-dimensional domain. . . . 7
1.2 Basis functions for standard one dimensional Lagrangian finite elements of different degrees. . . . 8
1.3 Mapping to reference elements in one and two dimensions. . . . 8
1.4 Example of parabolic finite element approximation in one dimension : continuous (left) and discontinuous (right) cases. . . . 9
1.5 Linear finite element approximation in two dimensions : continuous (left) and discontinuous (right) cases. . . . 9
1.6 In red, the considered equation, and in blue, its stencil. . . . 15
1.7 The stencil for discontinuous P2 finite element. . . . 16
1.8 R refinement applied to a grid. . . . 17
1.9 H refinement applied to a grid. . . . 17
1.10 P refinement applied to a grid. . . . 17

2.1 Detailed architecture of the `AeroSol` platform. . . . 26
2.2 Transformation from a simple 4 triangles mesh (on the left) to a rich graph (on the right). . . . 27
2.3 Different examples of iterators provided by `PaMPA`. . . . 28
2.4 The partitioning and the redistribution of the mesh entities. . . . 29
2.5 Decomposition of the EntType integer. Example given for building a straight hexahedron of order 5. . . . 30
2.6 Use of binary masks to decompose an `EntType`. . . . 31
2.7 Hierarchical decomposition of a triangle at different orders with a continuous or a discontinuous solution. . . . 32
2.8 Example of the orientation issue with two triangles. . . . 32
2.9 The corresponding lists of DoFs. . . . 33
2.10 Construction of the orientation. . . . 33
2.11 Orientation and Rotation issue. . . . 34
2.12 Comparison of the weak scalability obtained for `AeroSol` and `Aghora` with the library `mvapich2`. All the results show good scalability. The least scalable case is `Aghora` for $p = 1$, and the most efficient is `Aghora` for $p = 4$. Nevertheless, if we consider the results given in Table 2.4, we see that `AeroSol` is much more costly if execution time is considered. This explains that weak scalability is better, as `AeroSol` has more computations for overlapping communications. . . . 38

3.1 Comparison of the classical approach to develop applications with respect to the runtime system approach. . . . 43

3.2 A basic representation of a Finite Element Method. The method visits each element, builds contributions and assembles them in the global linear system. . . . 47
3.3 Full directed acyclic graph of the computation of the contribution and assembly for the laplacian problem. 48
3.4 Directed acyclic graph of the assembly for the laplacian problem. 49
3.5 Full directed acyclic graph of the resolution of the laplacian problem. 50
3.6 Sequential assembly operation on a simple mesh. 51
3.7 Two elements overlap. Their contributions overlap too. 51
3.8 Each contribution block is divided in rows. And each rows is treated in parallel. . 52
3.9 6 triangles share an equation. Each element receives a different color to be treated in a sequential way. 52
3.10 On the left, a mesh with 6 triangles. On the right, the block diagonal matrix A_E, each block correspond to the contribution C_i built from the triangle T_i. 54
3.11 The matrix M, with its coefficient colored. On the right, The contributions are divided in rows, and each row is colored according to the coloration of M. 55
3.12 Focus on the row 4, and the scheduling for summing the contributions. 55
3.13 Focus on the row 4, with a temporary buffer. The contributions are splitted between the original row 4 and the buffer. A last step will assemble the buffer in the row 4. 56
3.14 Naive assembly of three contribution blocks, with the associated task graph. . . . 56
3.15 Splitting of the data. 57
3.16 Taskified version of the assembly of three contribution blocks, with the associated task graph. 57
3.17 Taskified version of the assembly with the associated weight. 57
3.18 Tasks in a chain are gathered in a single task. 58
3.19 Fixed priorities for each chain. 58
3.20 Adaptive priorities for each chain. 59
3.21 Second test case, 2D grid with blocks of size 121x121, and a matrix of 203k entries. Influence of the tile size for the different strategies with the two runtimes. 62
3.22 Third test case, 3D grid with 512 blocks of size 512x512 and a matrix of 185k entries. Influence of the tile size for the different strategies with the two runtimes. 63
3.23 First test case, `MUMPS` reduction. The assembly is done in a dense block (the father) of size 3061x3061. Contribution blocks sizes vary from 1 to the size of the father. Influence of the tile size for the different strategies with the two runtimes. 64
3.24 Efficiency for the second test case, the 2d grid. 65
3.25 Efficiency for the third test case, the 3d grid. 66
3.26 Efficiency for the first test case, extracted from the direct linear solver `MUMPS`. . . 67

General Introduction

Numerical modeling is nowadays an essential part of engineering analysis, be it is to design a new plane, or to detect underground oil reservoirs. Numerical simulations have indeed become an important complement to theoretical and experimental investigation, allowing one to reduce the cost of engineering design processes. Indeed, practical experimental investigations require the construction of maquettes or even of real scale models, which is time consuming, expensive, and sometimes can even be impossible due to safety reasons (e.g. combustion experiments, space reentry conditions, atomic bomb testings, etc). In all these cases, numerical simulations become the only possible tool of experimentation.

Computational fluid dynamics, or CFD for short, is the study of systems involving fluid flows, heat transfer and/or chemical reactions using computer-based simulation. It is used in various domains, from aerodynamics of vehicles or aircrafts, to weather forecasting, from turbomachinery to biomedical engineering. The simulation of fluids involves the solution of complex and stiff algebraic problems, defined on large computational grids, discretising space with millions of computational cells at each time step, for very large physical times, thus for a large number of time steps. The time to complete some simulation ranges from minutes to weeks or even months, depending on the resolution requirements, and on the physical duration of the phenomena under investigation. The only viable solution to reduce the simulation time is consequently to resort to super computers and high performance parallel computing.

High performance computing, or HPC for short, first appeared in the 60's with the first vector processors computers. Vector computers have the capability to work on a vector of data instead of scalar values. However, vector processors are expensive and unfit for irregular tasks, and after dominating the early 70s, architectures based on standard, less expensive, processors emerged to replace vector processors. Architectures can be based on RISC (Reduced Instruction Set Computing) processors like the IBM Power architecture, and even the ARM architecture that is ubiquitous on smart devices, tablets, smartphones. RISC processors use a smaller set of highly optimized instructions.

In the high performance computing domain, an important observation acts as a law and governs the evolution of the performance of computers. Moore's law states that every two years, the number of transistors in an integrated circuit doubles.
For years, manufacturers delivered CPU with higher clock-rate and higher instruction-level parallelism, but the continuous increase in clock rate stopped when the CPU power dissipation became unsustainable. Though the clock rate has stopped progressing, the miniaturization of transistors kept progressing. Manufacturers proposed new architectures with multiples cores, and even new specialized processing units, in the extra space freed by the miniaturization. These new multi-core architectures introduced a new programming paradigm : the multi-threaded approach.

In the early 2000s emerged the GPGPU, that is general purpose computing on graphical processing units using of the GPU devices as floating-point accelerators. These devices, traditionally used to transform sets of geometrical primitives into pictures, comprise a high number of cores, and a high bandwidth between the central memory and the device memory. It is now common to find nodes in supercomputers with several GPUs. Yet programming these devices is complicated and requires a lot of effort to maintain the code and its performance across successive generations of hardware. Last year, INTEL introduced a new accelerator, the Xeon Phi.

All these new devices complexified the architectures of modern supercomputers. The mix between technologies and programmation paradigm requires new techniques and new levels of abstraction to separate the algorithms and the implementations of tasks, the "separation of concerns".

This thesis considers the study of algorithms allowing to solve PDE problems arising in fluid flow modeling with finite element techniques. The proposed approaches have goals. On the one hand, they aim at ensuring lasting quality of the software produced due to its flexibility and capability for further extension, as well as to a transparent handling of distributed memory parallelism in presence of complex heterogeneous, dynamic data. On the other hand, we have also started the study of issues related to the implementation of finite element techniques on modern heterogeneous architectures involving a complex hierarchy of memory levels, and in particular we have considered the efficiency of the implementation in presence of shared memory levels.

These contributions have translated into the production of the software `AeroSol` : a generic platform for the finite element solution of PDE problems. This platform is developed in C++ and uses object-oriented programming. The platform is divided in different hierarchical levels, each mirroring a corresponding level of the numerical discretization process. Each feature is implemented in a specific set of classes. The main components of the platform are the following :

Hybrid Mesh : `AeroSol` will handle meshes of any dimension, composed of any type of elements based on triangles and quadrangles. The solution supported by the element will be decoupled from the geometry, so on one side, we will support an arbitrary order of the solution on each element, and on the other side, the elements can be curved if needed to fit the domain geometry.

Finite Elements Methods : This platform is developed by two teams, the BACCHUS INRIA team, historically doing residual distribution schemes, using continuous finite elements, and the CAGIRE INRIA project, doing experiments and using discontinuous Galerkin schemes. The platform will handle the two scheme families.

Hybrid Parallelization : To exploit the potential of modern architectures, the platform will have to handle many-core architectures, and will be able to use GPUs and all kinds of co-processors.

The manuscript is organised in three main chapters. The first chapter will cover some of the mathematical aspects and give the main mathematical notations that will be used in the whole document. To do this we will take a generic PDE and apply Galerkin methods using continuous and discontinuous finite element to present the two Galerkin finite element methods. Relying on those methods, we will identify the main challenges that we have had to address in this work. The second chapter will treat the design of a generic and efficient object oriented platform for solving PDE-based problems using finite element methods. We will illustrate our discussion with the example of a continuous method applied to a particular problem : the SUPG scheme applied to a vortex advection problem proposed by Yee [75]. The last chapter will cover the question of the shared memory parallelization of finite element methods. We will present the

state-of-the-art approach based on coloration algorithms applied at different grains. Then, we will introduce the runtime system as a middleware abstracting the architecture specificities, improving the maintainability of the code and of its performance.
This manuscript is ended with a conclusive chapter underlining the main contributions of this work, and the perpectives it openes.

Chapitre 1

Setting and main challenges

Contents

1.1 General Context and motivation **5**
1.2 Model problem **5**
1.3 Mesh and finite element approximation **6**
1.3.1 Finite elements 7
1.4 Discrete equations **9**
1.4.1 Integrals and matrices 11
1.4.2 Time dependent equation 14
1.5 Challenges **15**
1.5.1 Abstracting the memory layout of finite element methods 15
1.5.2 Handling hybrid meshes 16
1.5.3 Mesh adaptation 17
1.5.4 Abstract the complex architecture 18

1.1 General Context and motivation

The objective of this chapter is to introduce finite element methods and the challenges encountered in the modelization of these methods. We will use a generic formulation of a problem to illustrate the path from a Partial Differential Equations (PDE) to a finite element method. We will show the differences between the continuous and discontinuous finite elements. We will describe some challenges induced by these differences, and the consequences they will have on the design of the platform that we want to create.

1.2 Model problem

This section is devoted to the introduction of the main challenges related to the implementation of a generic framework for the finite element solution of PDE. To illustrate this challenge, we will introduce the methods by considering the problem of finding a scalar real function u defined on a domain Ω, sub-set of the d-dimensional space $\Omega \subset \mathbb{R}^d$, satisfying the problem

$$\begin{cases} \nabla \cdot (\vec{\beta} u) - \nabla \cdot (\nu \nabla u) + \sigma u = f & \text{in } \Omega \\ \hat{n} \cdot \vec{\beta}\, u + \nu \nabla u \cdot \hat{n} = 0 & \text{on } \partial\Omega \end{cases}, \tag{1.1}$$

with f a given real function also defined on Ω, $\vec{\beta}$ a given constant vector of $\mathbb{R}^d$, and $\hat{n} \in \mathbb{R}^d$ the exterior unit normal to $\partial\Omega$, and with σ and ν prescribed constants, with $\nu \geq 0$. This equation is a simplified model of the PDEs encountered in several applications such as :

- heat transfer : in which case u represents the temperature, ν the heat conductivity, β accounts for convection, f an external source of heat, and σ is usually null ;
- mass transport : in which case u may represent the mass concentration of a solute, β accounts for mass transport due to an external flow, ν is the molecular diffusivity of the solute, σ represents of consumption production due to chemical reactions, and f an external source of mass.

A complete characterization of the required smoothness of the function f, and of the solution u, for problem (1.1) to be well posed is beyond the scope of this work. We refer the interested reader to more specific material, such as *e.g.* [32]. It is however important to note that solutions to (1.1) are not defined using the local PDE but rather using an integral variational statement of the type : *seek* $u \in H^1$, *such that* $\forall\, v \in H^1$

$$-\int\limits_{\Omega} \nabla v \cdot \vec{\beta} u \, d\Omega + \int\limits_{\Omega} \nu \nabla v \cdot \nabla u \; d\Omega + \int\limits_{\Omega} v \, u \; d\Omega = \int\limits_{\Omega} v \, f \; d\Omega \,, \tag{1.2}$$

where H^1 denotes a particular space of functions, namely the Hilbert space of functions whose first derivatives are integrable (see [32] for more). For the purposes of defining the exact solution of our problem, we have to consider (1.2) instead of (1.1). Note, however, that for a sufficiently smooth solution u, the last statement is equivalent to (1.1). In particular, it can be obtained from the latter upon

1. multiplication by v :

$$v \, \nabla \cdot (\vec{\beta} u) - v \, \nabla \cdot (\nu \nabla u) + \sigma v \, u = v \, f \, ;$$

2. integration by parts over the spatial domain Ω :

$$\int_{\Omega} \nabla \cdot (v \vec{\beta} u) \, d\Omega - \int_{\Omega} \nabla v \cdot \vec{\beta} u \, d\Omega - \int_{\Omega} \nabla \cdot (v \nu \nabla u) \; d\Omega + \int\limits_{\Omega} \nu \nabla v \cdot \nabla u \, d\Omega + \int\limits_{\Omega} \sigma v \, u \; d\Omega = \int\limits_{\Omega} v \, f \; d\Omega \, ;$$

3. use of the Gauss theorem to convert the first and third integrals into boundary integrals :

$$\cancel{\int_{\partial\Omega} v \hat{n} \cdot \vec{\beta} u \, \partial\Omega} - \int_{\Omega} \nabla v \cdot \vec{\beta} u \, d\Omega - \cancel{\int_{\partial\Omega} v \nu \nabla u \cdot \vec{n} \, \partial\Omega} + \int\limits_{\Omega} \nu \nabla v \cdot \nabla u \, d\Omega + \int\limits_{\Omega} \sigma \, v \, u \; d\Omega = \int\limits_{\Omega} v \, f \; d\Omega \, ;$$

4. application of the boundary condition $\hat{n} \cdot \vec{\beta} \, u - \nu \nabla u \cdot \hat{n} = 0$ on $\partial\Omega$.

This four-step procedure is reported here as it constitutes the basis of one of the two classes of methods we have considered in this work. The description of these two families of schemes is the objective of the next sections.

1.3 Mesh and finite element approximation

In order to formulate discrete variants of (1.2) or (1.1), the first step is the discretization of the spatial domain Ω. This is achieved by introducing a mesh, which is a collection of cells

or elements E, constituting a tessellation of the domain, as illustrated on figure 1.1 for the two-dimensional case. In particular, we shall denote the union of all the cells by Ω_h, with

$$\Omega_h = \bigcup E \ . \tag{1.3}$$

The parameter h denotes a reference size for the grid, usually the largest of the diameters of the external spheres (or circles) to the cells. In one dimension, this reduces to the length of the largest cell. Note that *a priori* Ω_h is not exactly equal to omega, unless an exact approximation of the boundary of the domain is employed. Typically, $\partial\Omega_h$ is defined by a piecewise polynomial which, save for particular cases, will only converge to $\partial\Omega$ when the mesh is refined, that is when h tends toward to zero.

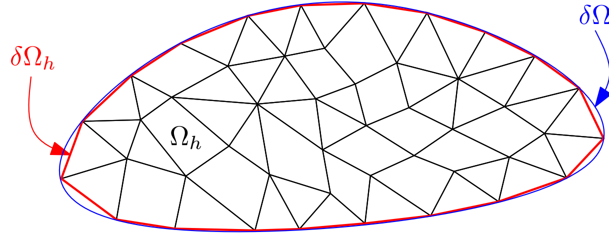


FIGURE 1.1 – Unstructured mesh of two-dimensional domain.

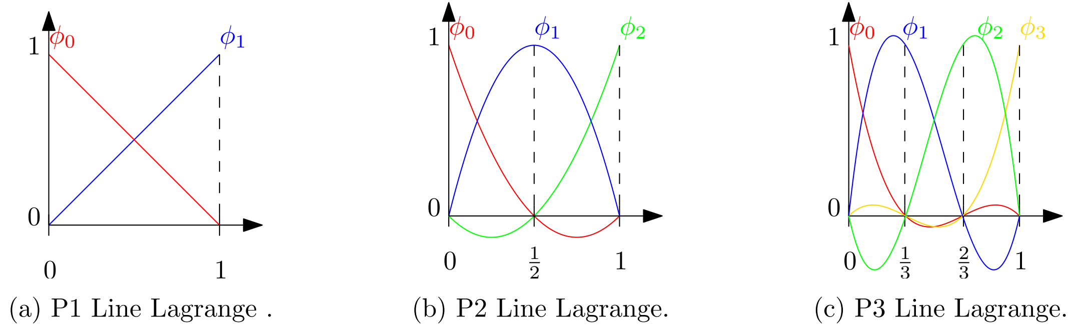
As also illustrated on figure 1.1, the elements composing Ω_h can have different shapes : segments in 1 dimension ; triangles and quadrangles in 2 dimensions ; tetrahedra, prisms, pyramids, or hexahedra in 3 dimensions. In order to improve the resolution of $\partial\Omega$, the faces laying on $\partial\Omega_h$ can be built using some high order polynomial surfaces (curves in two dimensions). For more details concerning these aspects, the interested reader can refer to [72].

1.3.1 Finite elements

The next step is to enrich the mesh with the necessary information required to represent a scalar (or vector) field u. This is done by associating values of this field with certain mesh entities, and by subsequently using these values as coefficients for some polynomial. In the finite element framework, this is done in a cell-wise manner. Firstly, for each cell, one defines a set of *basis functions*. Given a polynomial degree, say p, these functions constitute a basis of the associated space of polynomial, meaning that any polynomial of degree p can be written as a linear combination of these functions. The number, analytical form, and properties of these basis functions depend on the type of approximation chosen, on the number of space dimensions, and on the shape of the element. A thorough discussion is out of the scope of this manuscript and we refer the interested reader to [32]. For simplicity, we will consider here the case of Lagrangian elements. These elements are based on the polynomial interpolation of nodal values of u. In the following, we shall denote by $\{\phi_j\}_{j\geq 1}$ the basis functions, so that on a given element of the mesh, we will write

$$u_h = \sum_{j\geq 1} u_j \phi_j \ , \tag{1.4}$$

with u_j the value of u on a node j of the element, and with the degree of u_h depending on the choice of the basis. As an example, the basis functions for linear, parabolic, and cubic interpolation (on equally spaced points) in one space dimension are shown on figure 1.2.
The choice of the finite element allows one to generate locally to each mesh cell a polynomial u_h which can be used to compute values of the field u at locations in space that differ from



(a) P1 Line Lagrange . (b) P2 Line Lagrange. (c) P3 Line Lagrange.

FIGURE 1.2 – Basis functions for standard one dimensional Lagrangian finite elements of different degrees.

those used for the interpolation, or to evaluate derivatives of u. Note that the basis functions are universal polynomials. This universality is obtained by defining these functions on a so-called reference element, onto which the actual elements of the mesh are mapped. This mapping is defined as

$$x = F_E(\widehat{x}) = \sum_{j \geq 1} x_j \phi_j(\widehat{x}), \tag{1.5}$$

and allows one to account easily for generic elements, including curved faces. The practical use of this mapping in the discretization is discussed in the next section, while examples of mappings are shown on figure 1.3.

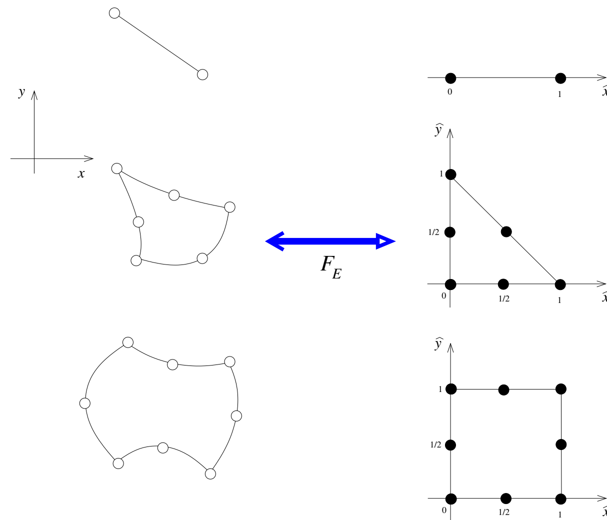


FIGURE 1.3 – Mapping to reference elements in one and two dimensions.

Once we are able to recover values of a field u (and of its derivatives) locally in a cell, we have to provide a global notion of this quantity, in order to be able to process cell faces and mesh nodes. Two radically different approaches exist. The fist one consists in requiring that all quantities should be single values at cell boundaries, thus ensuring the continuity of the approximated field u_h. The second does not impose this constraint, and allows one to have multiple values at cell boundaries. For obvious reasons, these two strategies are referred to as the *continuous finite element* and the *discontinuous finite element* approximations. Visual depictions of the two type of approximations are provided in Figures 1.4 and 1.5 for, respectively, parabolic one

dimensional and linear two dimensional approximations.

FIGURE 1.4 – Example of parabolic finite element approximation in one dimension : continuous (left) and discontinuous (right) cases.

Besides the implication on the numerical discretization of the equation, which will be discussed in the following sections, the main difference between the two methods is the number of point wise values necessary to generate the finite element polynomials. These values are generally referred to as *degrees of freedom*. In the continuous case, we associate a uniquely defined value to a node laying on an entity belonging to an element boundary. In the discontinuous case, these values are defined locally to each element, and are thus multiplied by the number of elements sharing the entity. This implies that the complete locality of the discontinuous approximation induces a much larger number of degrees of freedom. As a final remark, one can note that, in the discontinuous case, the basis functions are uniquely defined over each element, so that, for a node j in element E, function ϕ_j has no meaning outside E. In the continuous case, on the contrary, for a node j on the boundary of E, we may define a global function ϕ_j which has a meaning in all elements sharing j.

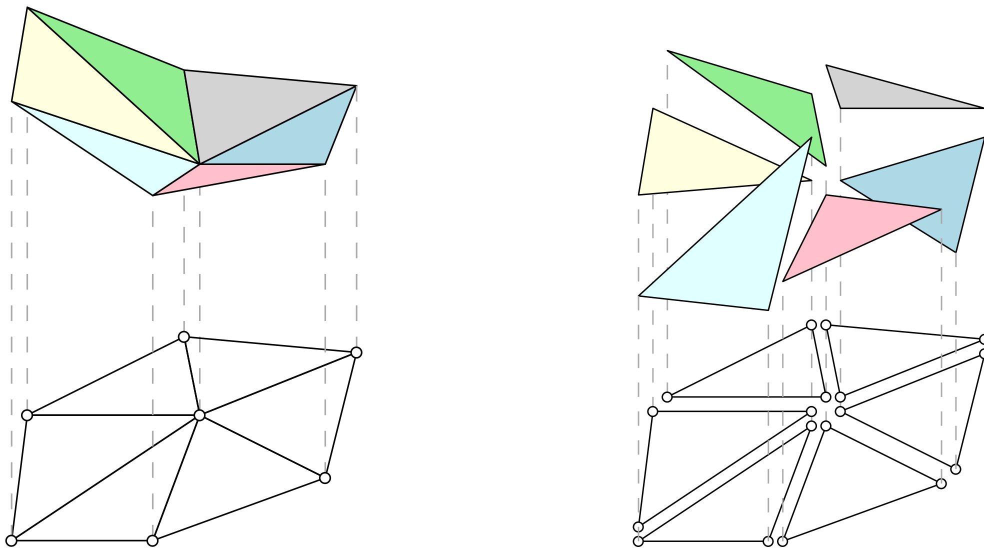


FIGURE 1.5 – Linear finite element approximation in two dimensions : continuous (left) and discontinuous (right) cases.

1.4 Discrete equations

We consider now the derivation of the discrete counterparts of (1.2) or (1.1), based on a continuous or discontinuous finite element approximation. By no means the presentation that follows is meant to be extensive. The objective of these pages is to highlight the main features of the

schemes involved. For a detailed discussion, the interested reader is referred to, *e.g.* [32, 29, 22] and references therein.

The simplest discretization one may consider is the *continuous Galerkin scheme* obtained when using continuous elements, and by restricting the variational statement (1.2) only to the functions handled by the finite element polynomials, namely to the finite dimensional space :

$$V_h = \mathrm{span}\{\phi_j\}_{j\geq 1} \subset H^1 \,.$$

This choice leads to the Galerkin scheme associated with the variational problem : seek $u_h \in V_h$ such that :

$$-\int_\Omega \nabla v_h \cdot \vec{\beta} u_h \, d\Omega + \int_\Omega \nu \nabla v_h \cdot \nabla u_h \; d\Omega + \int_\Omega v_h \, u_h \; d\Omega = \int_\Omega v_h \, f \; d\Omega \quad \forall v_h \in V_h. \tag{1.6}$$

For diffusion dominated cases, that is, when $\nu \gg \|\beta\| h$, the continuous Galerkin scheme provides a stable approximation. When the advective transport becomes relevant, additional care has to be taken in order to guarantee the stability of the numerical method. A possible alternative is the so-called *Streamline Upwind Petrov Galerkin* method, or SUPG for short, which involves the modified variational problem : seek $u_h \in V_h$ such that

$$\begin{aligned} &-\int_\Omega \nabla v_h \cdot \vec{\beta} u_h \, d\Omega + \int_\Omega \nu \nabla v_h \cdot \nabla u_h \; d\Omega + \int_\Omega \sigma v_h \, u_h \; d\Omega \\ &+ \sum_{E\in\Omega_h} \int_E \vec{\beta} \cdot \nabla v_h \, \tau_h \, (\vec{\beta} \cdot \nabla u_h + \sigma u_h) d\Omega = \int_\Omega v_h \, f \; d\Omega + \sum_{E\in\Omega_h} \int_E \vec{\beta} \cdot \nabla v_h \, \tau_h \, f \, d\Omega \quad \forall v_h \in V_h. \end{aligned} \tag{1.7}$$

The SUPG scheme can now be shown to provide a stable approximation also in the advection dominated limit, that is, when $\nu \ll \|\beta\| h$. The so-called SUPG *stabilization parameter* τ_h represents a local time scale which depends on the value of the ratio $\|\beta\| h/\nu$.

The derivation of discrete equations in the discontinuous setting takes a different path. In particular, the main principle is to repeat steps 1. to 4. enumerated at the end of Section §1.2, over each of the mesh elements. This is repeated here for completeness and to introduce some additional notation. The first notion to understand is that of a global solution, which is defined as the collection of all the local (independent) elemental solutions. This means that we consider u_h as a global entity belonging to the ensemble of functions

$$V_h = \bigcup V_h^E \,, \quad V_h^E = \mathrm{span}\{\phi_j\}_{j\in E} \,.$$

The main steps toward the obtention of a Discontinouous Galerkin discretization of 1.1 are

1. For each element $E \in \Omega_h$ multiply 1.1 by, v_h^E an arbitrary element of V_h^E :

$$v_h^E \, \nabla \cdot (\vec{\beta} u_h) - v_h^E \, \nabla \cdot (\nu \nabla u_h) + v_h^E \, u_h = v_h^E \, f;$$

2. use integration by parts over each element E :

$$\begin{aligned} \int_E \nabla \cdot (v_h^E \vec{\beta} u_h) \, d\Omega - \int_E \nabla v_h^E \cdot \vec{\beta} u_h \, d\Omega - \int_E \nabla \cdot (v_h^E \nu \nabla u_h) \; d\Omega + \int_E \nu \nabla v_h^E \cdot \nabla u_h \, d\Omega \\ + \int_E \sigma v_h^E \, u_h \; d\Omega = \int_E v_h^E \, f \; d\Omega \,; \end{aligned}$$

3. use the Gauss theorem to convert the first and third integrals into boundary integrals

$$\int_{\partial E} v_h^E \hat{n} \cdot \vec{\beta} u_h \, \partial\Omega - \int_E \nabla v_h^E \cdot \vec{\beta} u_h \, d\Omega - \int_{\partial E} v_h^E \, \nu \nabla u_h \cdot \vec{n} \, \partial\Omega + \int_E \nu \nabla v_h^E \cdot \nabla u_h \, d\Omega$$
$$+ \int_E \sigma \, v_h^E \, u_h \, d\Omega = \int_E v_h^E \, f \, d\Omega \, ;$$

4. apply the boundary conditions. In this case, since one writes element-wise equations, the boundary condition is not necessarily the physical condition required by problem statement 1.1. One must realize that, while v_h^E is only defined in E, and its values on the boundary of the element is given by the limit of the value inside it, function u_h is defined as a collection of cell-wise fields, *a priori* discontinuous on the boundaries of the elements, as seen in the previous section. For this reason, in the boundary integrals, the quantity $(\vec{\beta} u_h + \nu \nabla u_h) \cdot \hat{n}$ is replaced by a so called *numerical flux* $\widehat{H}$, function of the values of u_h on both sides of the faces of which ∂E is composed. Denoting by E' the (variable) neighbouring element sharing locally a face of ∂E, the final variational statement becomes : seek $u_h \in V_h$ such that

$$\begin{aligned} - \int_E \nabla v_h^E \cdot \vec{\beta} u_h \, d\Omega + \int_E \nu \nabla v_h^E \cdot \nabla u_h \, d\Omega + \int_E \sigma \, v_h^E \, u_h \, d\Omega \\ + \int_{\partial E} v_h^E \, \widehat{H}(u_h^E \, ; u_h^{E'}) \, \partial\Omega = \int_E v_h^E \, f \, d\Omega \quad \forall v_h^E \in V_h^E . \end{aligned} \tag{1.8}$$

The last expression shows that the DG discretization is given by a local variational statement identical to the pure continuous Galerkin one, plus a boundary term providing inter-element coupling and stability. Typical forms of the numerical flux are of the type [29, 22] :

$$\begin{aligned} \widehat{H}(u_h^E \, ; u_h^{E'}) = & \frac{\vec{\beta} u_h^E + \vec{\beta} u_h^{E'}}{2} \cdot \hat{n} + \nu \frac{\nabla u_h^E + \nabla u_h^{E'}}{2} \cdot \hat{n} \\ & + \alpha_1 \nabla v_h^E \cdot \hat{n} (u_h^{E'} - u_h^E) + \alpha_2 \nabla v_h^E \cdot \hat{n} (\nabla u_h^{E'} - \nabla u_h^E) \cdot \hat{n} \, . \end{aligned} \tag{1.9}$$

1.4.1 Integrals and matrices

All of the above finite element methods are formulated as a variational statement in which we seek a discrete solution u_h generated as the linear combination of some basis functions, when we test the original equation against an arbitrary element of a certain polynomial space. To ensure the validity of this statement *for any arbitrary element of the test space* it is enough to write it for *all the elements of the basis generative the space*. In other words, for the continuous Galerkin scheme, we will write :

$$- \int_\Omega \nabla \phi_i \cdot \vec{\beta} u_h \, d\Omega + \int_\Omega \nu \nabla \phi_i \cdot \nabla u_h \, d\Omega + \int_\Omega \phi_i \, u_h \, d\Omega = \int_\Omega \phi_i \, f \, d\Omega \quad \forall i \geq 1. \tag{1.10}$$

Similarly, the SUPG equations are obtained by requiring that :

$$
\begin{aligned}
&-\int_{\Omega} \nabla\phi_i \cdot \vec{\beta} u_h \, d\Omega + \int_{\Omega} \nu \nabla\phi_i \cdot \nabla u_h \; d\Omega + \int_{\Omega} \sigma \phi_i \, u_h \; d\Omega \\
&+ \sum_{E\in\Omega_h} \int_{E} \vec{\beta} \cdot \nabla\phi_i \, \tau_h \, (\vec{\beta} \cdot \nabla u_h + \sigma u_h) d\Omega = \int_{\Omega} \phi_i \, f \; d\Omega + \sum_{E\in\Omega_h} \int_{E} \vec{\beta} \cdot \nabla\phi_i \, \tau_h \, f \, d\Omega \quad \forall i \geq 1.
\end{aligned}
\tag{1.11}
$$

And lastly, the DG equations are obtained by requiring that :

$$
\begin{aligned}
-\int_{E} \nabla\phi_i \cdot \vec{\beta} u_h \, d\Omega + &\int_{E} \nu \nabla\phi_i \cdot \nabla u_h \, d\Omega + \int_{E} \sigma \, \phi_i \, u_h \; d\Omega \\
&+ \int_{\partial E} \phi_i \, \widehat{H}(u_h^E \, ; u_h^{E'}) \, \partial\Omega = \int_{E} \phi_i \, f \; d\Omega \quad \forall \, i \in E, \forall E.
\end{aligned}
\tag{1.12}
$$

We can finally transform these variational statements in algebraic problems. To do that, we must recall that within each element we can use the definition of u_h, namely equation 1.4. This leads to linear systems which can be written as

$$AU = F, \tag{1.13}$$

where the size and entries of A and F, as well as the size of U, depend on which of the three methods discussed above we select. In particular, the size of U, and F is given by the total number of degrees of freedom, denoted in the following by N. We recall, that this number is considerably larger in the discontinuous case, due to the repeated storage of values on element boundaries. Matrix A is a matrix whose range is N, yet is sparse, with a pattern which will be discussed in more detail in the following chapters.

To give an example of how these matrices are actually assembled, we recall that, for the continuous Galerkin scheme we have :

$$A_{i,j} = -\sum_{E|i,j\in E} \int_E \vec{\beta}\phi_j \cdot \nabla\phi_i \, d\Omega + \sum_{E|i,j\in E} \int_E \nu \nabla\phi_i \cdot \nabla\phi_j \, d\Omega + \sum_{E|i,j\in E} \int_E \sigma\phi_i\phi_j \, d\Omega \tag{1.14}$$

while for the right hand side we have :

$$F_i = \sum_{E|i,j\in E} \int_E \phi_i f \, d\Omega. \tag{1.15}$$

We now benefit from the introduction of the mapping to a reference element, which is used to re-cast all of the integrals over these elements. We recall that over E we have :

$$x = F_E(\widehat{x}), \tag{1.16}$$

and that

$$\phi_i(x) = \widehat{\phi}_i(\widehat{x}). \tag{1.17}$$

We then define the Jacobian J_F as the matrix :

$$J_F = \left[\frac{\partial x_j}{\partial \widehat{x}_i}\right]. \tag{1.18}$$

This allows us to evaluate the gradients of the basis functions on the physical element as :

$$\nabla \phi_i(x) = J_F^{-1}.\widehat{\nabla}\widehat{\phi}_i(\widehat{x}). \tag{1.19}$$

with the $\widehat{\phi}_i$ universal basis functions defined on the reference elements. Finally, to evaluate the integrals, we perform a change of variables :

$$dx = \det(J_F)d\widehat{x}. \tag{1.20}$$

Using 1.20, we can for example write

$$\begin{aligned} A_{i,j} = &- \sum_{E|i,j\in E} \int_{\widehat{E}} \vec{\beta}\widehat{\phi}_j \cdot (J_F^{-1}\nabla\widehat{\phi}_i)\det(J_F)d\widehat{x} + \sum_{E|i,j\in E} \int_{\widehat{E}} \nu(J_F^{-1}\nabla\widehat{\phi}_i)\cdot(J_F^{-1}\nabla\widehat{\phi}_j)\det(J_F)d\widehat{x} \\ &+ \sum_{E|i,j\in E} \int_{\widehat{E}} \sigma\widehat{\phi}_i\widehat{\phi}_j\det(J_F)d\widehat{x}. \end{aligned} \tag{1.21}$$

The main advantage of this formulation is that it allows us to evaluate these integrals using numerical quadrature formulas defined on the reference element, thus using quadrature points positions which are the same for all the elements of the mesh having the same shape. The expressions obtained for the other schemes are similar and omitted for brevity.
It is interesting to note that, independently from the continuous or discontinuous nature of the finite element approximation, the linear system obtained is *global* in the sense that it couples all the degrees of freedom. In particular, comparing expressions 1.10 and 1.12 it is obvious that the continuous Galerkin system matrix is obtained as an assembled sum of the discontinuous Galerkin expressions. In particular, we can distinguish three classical blocks in finite element analysis :

Convection matrix :

$$A^c_{i,j} = \int_E \vec{\beta}\phi_j \cdot \nabla\phi_i \, d\Omega\,;$$

Stiffness matrix :

$$A^s_{i,j} = \int_E \nabla\phi_i \cdot \nabla\phi_j \, d\Omega\,;$$

Mass matrix :

$$A^m_{i,j} = \int_E \phi_i \, \phi_j \, d\Omega\,.$$

The continuous Galerkin method is obtained upon the assembly of these three blocks, namely :

$$A^{\mathrm{CG}} = \sum_{E\in\Omega_h} \left(A^c_E + \nu A^s_E + \sigma A^m_E\right).$$

This assembly is what gives the method a global character, coupling unknowns stored in nodes in all the elements of the mesh. In the DG method, on the other hand, one only considers one elemental instance at the time, thus in principle retaining a completely local, element-wise character. However, this locality is broken by the term containing the flux which introduces what we could call *flux matrix entries* of the type (cf. equation (1.9)) :

$$A^f_{i,j} = \int\limits_{\partial E} \phi_i\phi_j \frac{\vec{\beta}\cdot\widehat{n}}{2}\,\partial\Omega + \int\limits_{\partial E} \nu\phi_i \frac{\nabla\phi_j\cdot\widehat{n}}{2}\,\partial\Omega \pm \alpha_1 \int\limits_{\partial E} \nabla\phi_i\cdot\widehat{n}\phi_j\,\partial\Omega \pm \alpha_2 \int\limits_{\partial E} \nabla\phi_i\cdot\widehat{n}\nabla\phi_j\cdot\widehat{n}\,\partial\Omega$$

with the plus or minus sign depending on whether node j belongs to a neighbouring element or to element E itself, respectively. These entries lead to a large global algebraic system, coupling all the degrees of freedom in all the elements of the mesh.

1.4.2 Time dependent equation

Before discussing issues related to the implementation of these methods, let us discuss shortly the time problem

$$\begin{cases} \partial_t u + \nabla \cdot (\vec{\beta} u) - \nabla \cdot (\nu \nabla u) + \sigma u = f & \text{in } \Omega, \\ \hat{n} \cdot \vec{\beta}\, u + \nu \nabla u \cdot \hat{n} = 0 & \text{on } \partial\Omega \\ u(x,t) = u_0(x) & \text{for } t = 0. \end{cases} , \tag{1.22}$$

In order to extend the previous methods to this case, we first discretize in time, and then apply the schemes including the time increment as a source/sink term. We briefly discuss two simple cases that illustrate explicit and implicit time integration.

In the first case, we discretize the time derivative with the explicit Euler formula, obtaining :

$$\frac{u^{n+1} - u^n}{\Delta t} + \nabla \cdot (\vec{\beta} u^n) - \nabla \cdot (\nu \nabla u^n) + \sigma u^n = f. \tag{1.23}$$

Applying the schemes yields the following linear algebraic equations :

— *Continuous Galerkin.* In this case we have :

$$AU^{n+1} = BU^n + \Delta t F,$$

where

$$A = A^m \,, \quad B = (1 - \Delta t \sigma) A^m - \Delta t A^c - \Delta t \nu A^s.$$

This shows that a global system needs to be solved, whose matrix is the mass matrix introduced in the previous section ;

— *Streamline Upwind Petrov Galerkin.* In this case we obtain again :

$$AU^{n+1} = BU^n + \Delta t F.$$

However now

$$A = A^m + A^{m-su} \,, \quad B = (1 - \Delta t \sigma)(A^m + A^{m-su}) - \Delta t A^c - \Delta t \nu A^s - \Delta t A^{su},$$

having introduced the additional matrices :

$$A^{m-su}_{i,j} = \sum_{E|i,j \in E} \int_E \vec{\beta} \cdot \nabla \phi_i \, \tau_h \, \phi_j \, d\Omega \,, \quad A^{su}_{i,j} = \sum_{E|i,j \in E} \int_E \vec{\beta} \cdot \nabla \phi_i \, \tau_h \, \vec{\beta} \cdot \nabla \phi_j \, d\Omega$$

arising from the streamline diffusion terms (cf. Equations (1.7) and (1.11)). As in the previous case, the system matrix obtained is global, and couples all degrees of freedom.

— *Discontinuous Galerkin.* In this case, we can write the system as :

$$A_E U_E^{n+1} = BU^n + F \,, \quad \forall\, E,$$

where U_E represents the array containing only the degrees of freedom of element E and

$$A_E = A_E^m \,, \quad B = (1 - \Delta t \sigma) A_E^m - \Delta t A_E^c - \Delta t \nu A_E^s + \Delta t A^f.$$

Clearly we obtain here a simpler problem, involving smaller local element-wise blocks. The inter-element coupling is introduced in the right hand side via A^f.

Instead if we discretize the time derivative with the implicit Euler method, we obtain :

$$\frac{u^{n+1} - u^n}{\Delta t} + \nabla \cdot (\vec{\beta} u^{n+1}) - \nabla \cdot (\nu \nabla u^{n+1}) + \sigma u^{n+1} = f. \tag{1.24}$$

This problem can be recast as a particular case of (1.1) with modified values of the coefficients and of the right hand side. In particular, setting $w = u^{n+1}$, and

$$\vec{\beta}_{\Delta t} = \Delta t \vec{\beta}\,, \quad \nu_{\Delta t} = \Delta t \nu\,, \quad \sigma_{\Delta t} = \Delta t \sigma - 1\,, \quad f_{\Delta t} = \Delta t f + u^n\,,$$

we are now solving

$$\nabla \cdot (\vec{\beta}_{\Delta t} w) - \nabla \cdot (\nu_{\Delta t} \nabla w) + \sigma_{\Delta t} w = f_{\Delta t}\,.$$

As seen in the previous section, in this case, *all the methods give a global system coupling all the degrees of freedom in all the elements of the mesh.* In particular, the DG methods will give a considerably larger system for a fixed level of required accuracy, that is, for a fixed degree of the finite element polynomials.

1.5 Challenges

In order to develop a generic platform allowing the use of all the methods discussed in the previous section, we have identified some major challenges related to :

- the management of the memory for the two different classes of finite element approximations ;
- the heterogeneity of data in space related to the presence of elements of different shapes, or of different local polynomial approximations in the mesh ;
- the management of dynamic changes in the data layout due to local mesh or polynomial adaptation ;
- the efficient exploitation of modern multi-core platforms comprising complex memory hierarchies.

A more detailed discussion of these issues is provided in the following sections.

1.5.1 Abstracting the memory layout of finite element methods

The first challenge is to find a unified model to represent the unknowns of the system. We have shown that the finite elements differ between the two methods. At order 1, and with a continuous method, the unknowns are localized on the vertices. Then, elements share information on the vertices. By increasing the order of the solution, unknowns appear on the edges and the faces, and inside the elements as depicted in Figure 1.6. As the figure shows, the coupling terms come from all the elements sharing the unknown.

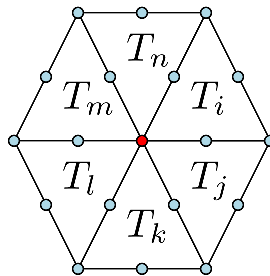
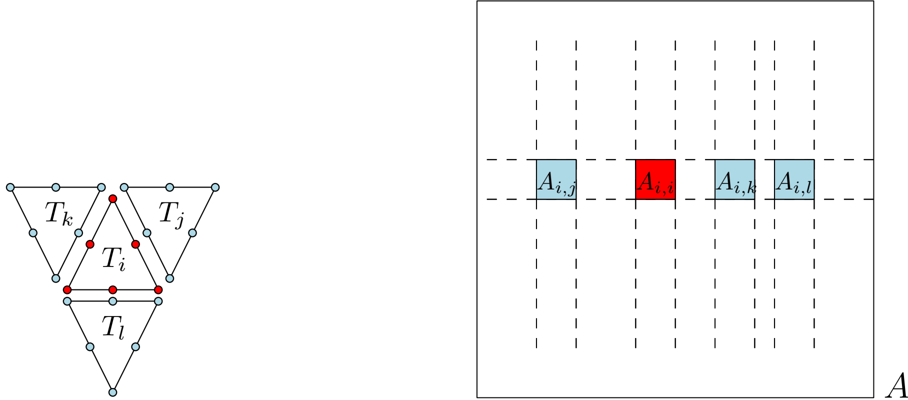


FIGURE 1.6 – In red, the considered equation, and in blue, its stencil.

On the other side, discontinuous elements have the same number of unknowns but do not share a single unknown with their neighbours. The direct consequence is a bigger global matrix whose size is the sum of the number of unknown for each element of the system. When the order of the solution increases, the number of unknowns in the system increases by a factor of $\mathcal{O}(order^{dimension})$. The coupling between the unknowns is performed during a phase where two elements sharing a face compute a numerical flux to update the corresponding blocks in the matrix and in the right hand side.



(a) Local equations of the element T_i with discontinuous finite elements of type P2 and their stencil.

(b) The equations of element T_i have coupling terms for each unknown of the neighbouring elements.

FIGURE 1.7 – The stencil for discontinuous P2 finite element.

To parallelise these methods, we act on two levels. First, we perform a domain decomposition, to split our problem across multiple processors. This implies that we will have extra work to do on the boundaries of the domain. The easiest way is to duplicate cells on both processors, such as each local cell has all the information to perform the computation. Here lies the first difference between continuous and discontinuous methods. In the former ones, cells exchange information between their vertices and, when doing high order methods, between their edges and faces. In the case of discontinuous methods, information is always exchanged between the faces. It means that we will have more ghost cells in the overlapping region with continuous methods than with discontinuous methods.

The parallelization diffes from a method to the other. In discontinuous methods, some of the terms of the equation can be fully computed locally, while the terms related to the numerical fluxes computed on a face between two elements must be performed carefully as they update elements on both sides of the face. Two faces can not be treated in parallel. On the other side, continuous methods compute contributions locally to an element, and the following phase, the assembly operation, must be performed carefully as elements sharing unknowns will try to update the same equations.

1.5.2 Handling hybrid meshes

The idea here will be to handle any kind of element shapes, ranging from segments in one dimension, triangles, quadrangles in two dimensions, to tetrahedra, pyramids, prisms, and hexahedra in three dimensions. Elements will be straight or curved. Also, the polynomials approximating the solution could have have different degrees on different elements.

1.5.3 Mesh adaptation

The idea of mesh refinement is to modify the mesh in order to get a better resolution of a phenomenon on a given region of the mesh. We can distinguish three types of mesh refinement :

— **r-refinement** : (Figure 1.8). It amounts to a deformation of the mesh while preserving its topology. The idea here is to change the coordinates of the points without changing connectivities.

(a) A simple grid of quadrangular elements with the region to refine in red.

(b) Nodes are moved to follow the phenomenon in the selected region.

FIGURE 1.8 – R refinement applied to a grid.

— **h-refinement** : (Figure 1.9). By changing the topology of the mesh, and interpolating the old solution on the new mesh, one can refine the mesh where needed and increase the resolution to follow small phenomenon or decrease the resolution where nothing relevant is happening anymore and computational resources are no longer required.

(a) A simple grid of quadrangular elements with the region to refine in red.

(b) The area has been remeshed with triangles.

FIGURE 1.9 – H refinement applied to a grid.

— **p-refinement** : (Figure 1.10). It consists in increasing or decreasing the order of the polynomials on the selected elements in order to capture the phenomenon or, conversely, to reduce the computational cost of a region of the mesh where nothing relevant is expected to happen.

(a) A grid composed exclusively of quadrangular Q2 with the selected region to refine in red.

(b) The order of the elements in the selected region has increased.

FIGURE 1.10 – P refinement applied to a grid.

1.5.4 Abstract the complex architecture

A few years ago, supercomputers were essentially composed of multi-core CPUs. Programming these architectures was easy with respect to the current situation. An hybrid approach with a message passing paradigm like PVM or MPI, coupled with a multi-threaded library to exploit the intra-node parallelism worked, very well. But this time is over, and now, the current generation of supercomputers includes different accelerators, from the classic GPUs to the newest co-processor from INTEL, the Xeon Phi. Programming these architectures is tedious, as distributing work between devices, handling the different levels of memory, and maintaining data consistency is complicated. Worse, maintaining the platform, and maintaining the performance of the platform on these fast-evolving architectures, can be a serious issue. These points will be discussed later in Chapter 3.

Chapitre 2

Software architecture

Contents

2.1 Introduction . . . **20**
2.2 Our needs . . . **21**
2.3 Related Works . . . **21**
2.3.1 Frameworks . . . 22
2.3.2 `Aghora` . . . 22
2.3.3 `deal.II` . . . 22
2.3.4 `hpGEM` . . . 23
2.3.5 `OP2` . . . 23
2.3.6 `COOLFluiD` . . . 23
2.3.7 `freefem++` . . . 23
2.3.8 `HiFlow`^{3} . . . 23
2.3.9 `PyCUDA` . . . 24
2.3.10 Discussion . . . 24
2.4 General Architecture . . . **24**
2.4.1 Functional requirements . . . 24
Evolutivity and contributing . . . 24
Elemental operations . . . 25
Refinement of the mesh . . . 25
2.4.2 Proposed architecture . . . 25
2.4.3 Technical requirements . . . 26
2.5 Delegating the functionalities . . . **27**
2.5.1 `PaMPA` . . . 27
2.5.2 Runtime systems . . . 29
2.5.3 Linear Solver . . . 29
2.6 Functional implementation . . . **30**
2.6.1 Specializing the entities . . . 30
2.6.2 Solution Object . . . 31
2.6.3 Cell reconstruction . . . 32
2.7 Computational implementation . . . **34**
2.7.1 Finite elements collection . . . 34
2.7.2 Equations of state, models and, numerical fluxes . . . 35
2.7.3 Integrator classes . . . 35
2.7.4 Spatial scheme . . . 35
2.7.5 Time scheme . . . 35
2.8 Experiments and results . . . **36**

2.8.1 Numerical tests 36
2.8.2 Yee vortex 36
2.9 Conclusion 39

2.1 Introduction

The use of numerical simulation based on finite element discretizations of partial differential equations describing physical processes is nowadays critical for the comprehension of physical phenomena in various application fields. It aims at delivering accurate simulation results, allowing to replace costly experimentation. Complexity of geometries and models and demand for higher accuracy typically result in a huge number of modeled degrees of freedom (DoF) with complex couplings. Due to the complexification of high performance architectures in the last decade (multicore systems with complex memory hierarchies which may be equipped with accelerators, etc), the design of high performance applications becomes a challenging task, especially from the performance portability point of view. This results in a strong need for flexible, portable, and adaptable frameworks which allow domain specialists to design efficient and accurate methods with moderate programming efforts.
In this chapter, we consider the problem of designing a flexible platform for the design of complex numerical simulation scheme and present the `AeroSol` platform that we have developed for this purpose. Even though several frameworks that target this same problem already exist, we will show how the `AeroSol` approach is reliable and flexible. We will discuss and motivate the design choices of the global architecture with respect to the challenges we found. Finally we will show how we addressed all these issues.

Algorithm 1 General space/time iteration scheme.

1: $t \leftarrow 0$
2: $\delta t \leftarrow \delta t_0$
3: $U \leftarrow U_0$
4: **while** $t < T_{max}$ **do**
5: $\quad B \leftarrow 0$
6: $\quad A \leftarrow 0$
7: $\quad$ **for each** Element e **do**
8: $\quad\quad U_e \leftarrow localSolution(U, e)$
9: $\quad\quad G_e \leftarrow localGeometry(G, e)$
10: $\quad\quad a_e \leftarrow f(G_e, U_e, e)$
11: $\quad\quad b_e \leftarrow g(G_e, U_e, e)$
12: $\quad\quad$ assemble $(A,\ a_e,\ e)$
13: $\quad\quad$ assemble $(B,\ b_e,\ e)$
14: $\quad$ **end for**
15: $\quad$ Solve $A.X = B$
16: $\quad U \leftarrow U - \delta t.X$
17: $\quad t \leftarrow t + \delta t$
18: **end while**

We recall in Algorithm 1 the general space/time iteration scheme used for an explicit finite elements solvers. After a description of existing generic solvers for FEM-based numerical simu-

lation frameworks, we give in the following sections a description of the design choices for the platform `AeroSol`.

2.2 Our needs

Our needs for a CFD platform are multiple. We can classify them in two categories. First, we enumerate all our needs related to the mathematical aspect of the methods we are going to use :

Mesh. In order to have to most generic platform possible, we need to support any dimension and any shape of element possible. To fit the boundaries of the computational domain, curved elements have to be supported by the platform.

Methods. Our objective here is modest : we have to support continuous and discontinuous methods. In our context, we do not need to extend our platform to other methods. We are targeting two types of methods. Firstly, the methods based on continuous Galerkin method. Among their specificities, we can cite lower memory consumption due to data sharing between elements. The second class of methods are the discontinuous methods like the discontinuous Galerkin method. Two important things differ from the continuous method : the memory layout, and the algorithmic aspect. The data, normally shared between elements is replicated in each element, enforcing a discontinuity between elements. Consequently, an additional step of computation is performed on each interface to handle this discontinuity.

Finite Elements. In a very classical way, our platform must handle low order and high order solutions, coupled with a large variety of element shapes. It means that we will have to provide a large bank of finite elements, quadrature formulae, and geometrical functions.

The second and last category relates to the high performance computing (HPC) aspect. The ecosystem of HPC is quickly evolving and hardware architectures can be deprecated in a short time, replaced by a new, yet close, generation, forcing the developer of such a platform to keep his platform up-to-date. Thus, keeping a high level of performance on these architectures requires a tremendous amount of effort. Our platform should ensure performance portability by relying on third-party runtime systems.

2.3 Related Works

Several initiatives have emerged in the last twenty years targeting the design of solvers for mesh-based numerical simulations. These efforts can be categorized according to modelization criteria (e.g., mesh type, numerical scheme, type of discretization) or implementation criteria (e.g., parallelization scheme, genericity, etc). We display in Table 2.1 a comprehensive set of software projects and their respective characteristics. We classified these different software according to several criteria :

Mesh. Does the platform support any kind of element, in any dimension ? Does it support curved elements ?

Parallelism. Does the platform uses a regular MPI approach, or does it exploit the parallelism at different levels and/or use accelerators ?

Methods. Does the platform handle only the continuous or discontinuous Galerkin method ? Does it support other methods ?

As we can see, most of the software projects are dedicated to either a certain type of discretization, a given class of methods or are putting most of their effort in the parallel features. Only few of them are designed in such a generic way that they can be viewed as a toolbox on

Software	Hybrid Mesh	Hybrid Parallelism	Methods		
			DG	CG	other
Aghora [27, 59]	yes	yes	yes	N/A	
BEM++ [69]	no	no	yes	yes	
COOLFluiD [44, 67]	yes	yes	yes	yes	
deal.II [13]	no	yes	yes	yes	
DROPS [71]	N/A	yes	N/A	yes	
Dune-Fem [28]	yes	no	yes	N/A	
FEniCS [55, 56]	yes	yes	yes	N/A	
freefem++ [38]	no	no	no	yes	
HiFlow^3 [7, 40]	yes	yes	yes	yes	
hpGEM [65]	yes	no	yes	N/A	
ModFEM [60]	N/A	yes	yes	yes	
OP2 [35, 61]	no	yes	N/A	N/A	finite volume
OpenFOAM [42, 63]	yes	yes	no	yes	finite volume

TABLE 2.1 – Comparison of state-of-the-art softwares solving PDE problems using finite element methods (“N/A” : information not available from the literature).

top of which one can implement any kind of numerical simulation solver. We will refer to this category as *frameworks* and will give a brief description of those that are closest to our AeroSol platform.

2.3.1 Frameworks

Dune-Fem is a framework for solving PDEs problems using discontinuous Galerkin methods. It is developed in C++, and is parallelized using MPI. Two platforms are based on Dune-Fem : BEM++ and ModFEM.
OpenFOAM is a toolbox for simulating finite volume and finite element methods. Developed in C++, it uses a domain decomposition to handle distributed memory parallelism, and can use multi-core architecture as well. It handles any kind of element, in any dimension.
The FEniCS project [1] is a framework for evaluating variational forms. It uses hybrid parallelism to handle shared and distributed memory. It supports high order solution on hybrid meshes, and can simulate continuous or discontinuous methods.

2.3.2 Aghora

The development of the Aghora library began in 2012 with the PRF [3] of the same name. It is a high order, arbitrary order, finite elements library based on discontinuous elements. The models solved are the 3D compressible Euler or Navier-Stokes and RANS equations (with BR2 formulation for diffusive fluxes, see [15]) based on straight-sided and curved tetrahedra, hexahedra and prisms. It is developed in FORTRAN 95, and its development is based on previous experiments with two dimensional codes that were developed for example within the ADIGMA project [49].

2.3.3 deal.II

deal.II [12] is the successor of the DEAL (Differential Equations Analysis Library) platform. The aims of deal.II are flexibility, efficiency and user-friendliness. The goals of the project are

3. Projet de Recherche Fédérateur

its extensibility and its usability by hiding all the complex details from the developer. Moreover, performance is considered a key objective.
It is developed in C++, and is heavily documented. It uses `CMake` and `CTest` [13] for compilation and testing. The platform simulates finite element methods [11]. The main particularity of this platform is that it only supports segments in 1D, quadrilaterals in 2D, and hexahedra in 3D [14].

2.3.4 hpGEM

`hpGEM` is a framework developed in C++ that focuses on the simulation of discontinuous Galerkin finite element methods. It can handle any dimension of hybrid meshes. It supports high order solutions.
It makes use of the object-oriented concepts to provide a flexible and modular platform. One of the main objectives of its authors is accessibility. They want their platform to be easy to use for beginners who have no strong knowledge in C++.
They only handle discontinuous Galerkin methods, while we want to handle either continuous or discontinuous methods.

2.3.5 OP2

`OP2` [61] is a library framework for unstructured meshes. It is the follower of the `OPlus` [21] library (Oxford Parallel Library for Unstructured Solvers). `OP2` aims at decoupling the implementation of mathematical methods from their parallel implementation, in order to achieve performance portability. The idea behind `OP2` is to develop a user code using their API, to specify the data manipulated and to let the runtime generate dynamically the kernels required for scheduling and running computations on the target architectures.

2.3.6 COOLFluiD

`COOLFluiD` [67] is a platform developed in C++ at the von Karman Institute for Fluid Dynamics (Belgium). The main objectives of the `COOLFluiD` project are flexibility, efficiency and usability of the platform.
To construct an application, a set of elementary components are composed to achieve more and more complex features. A component implements a functionality using a set of frameworks. The framework structures the design patterns assembled from the different classes.

2.3.7 freefem++

`freefem++` is a software solving PDE in two or three dimensions using continuous Galerkin methods. It supports only triangular elements in 2D and tetrahedra in 3D. The software is parallelized using MPI. They provide a high-level input language for the end-user.

2.3.8 HiFlow^3

`HiFlow`^{3} is a multi-purpose finite element software package that is portable across a wide variety of platforms, including emerging technologies. It relies on a modular approach, based on object-oriented programming, to provide an evolutive toolbox for designing numerical simulations. It is, together with `COOLFluiD`, the platform closest to the `AeroSol` framework.

2.3.9 PyCUDA

The PyCUDA and PyOpenCL are python wrappers [45, 47, 48] to the CUDA and OpenCL low-level programming interfaces for GPUs. In [46] they have been used in the specific context of numerical simulation to build a numerical simulation for discontinuous Galerkin finite elements methods on GPU architectures. The main idea of these platforms is to generate at run time the required kernels to perform the computations. By generating on-the-fly the source code required for the problem, they allow the compiler to optimize the source code.

2.3.10 Discussion

Modern platforms dealing with the simulation of finite element methods use an object oriented language. This ensures the modularity, the flexibility, and the evolutivity of the platform. Furthermore, some of them use compilation and testing frameworks, like CMake/CTest, to perform continuous integration and to ensure software robustness. Some of them dynamically generate optimized kernels, and use them to speed-up the simulation.

In order to be modular and to enable a good re-usability, our platform will be developed in an object-oriented language. An inheritance mechanism will simplify the code that needs to be written by the programmer, and will allow him/her to extend/replace blocks of code. Moreover, our platform will use CMake and CTest to compile, deploy and run tests. With non regression testing, the developers will be able to detect problems when contributing to the software.

To ensure performance, our platform will run on modern architectures. It will require to handle distributed memory parallelism as well as shared memory parallelism. To deal with distributed memory parallelism, our platform will be built on top of a high level middleware which will handle the mesh and the communications. It will produce a good domain decomposition and a good renumbering that will ensure good load balance and improved data locality during the computations. Each domain will be mapped onto a node of the computer and, in each node, we will exploit shared memory parallelism with different techniques. This subject will be covered in Section 3.1.

In the next section, we will discuss the technical requirements for our platform and see how our platform, namely AeroSol will be structured.

2.4 General Architecture

2.4.1 Functional requirements

Evolutivity and contributing

In order to keep the platform easily maintainable and to help developers to contribute to it, we will rely on the principle of separation of concerns. This means that we have to identify a set of functionalities and modules and define the interactions between each module and its peers. By using an object-oriented programming language, the separation of concerns becomes a simple separation in objects and the definition of interfaces to manage the interactions between these objects. Thanks to the inheritance mechanism, an external contributor can develop a new method, a new finite element, or link the platform with his favorite linear solver.

By partitioning the functionalities into separate objects, we enable two things. Firstly, it is easier to extend the platform by adding a new module following the rules (respecting the interfaces of the connected objects) of the original module. Secondly, developers of various fields can join and contribute in the platform without even considering what occurs in the other modules.

Elemental operations

To be able to support any kind of mesh, as well as continuous and discontinuous methods, we have to represent our element in such a way that there is no memory over-consumption. Like we saw before, the targeted mathematical methods differ in memory layout and in the localization of the memory on each element. Discontinuous elements embeds all the information and share nothing with the surrounding elements, while continuous elements share a lot of information on their interface with other elements. When the order of the solution increases on these elements, not only does the amount of shared information increase, but the localization of this information changes, too. In the three dimensional case, information is shared on the vertices and, as the order increases, edges become involved and, then faces. To solve this problem, we split our element into a hierarchical set of entities. Each entity receives an integer characterizing it. This integer embeds information such as the shape of the entity, whether the entity is curved or not, whether the entity is a boundary entity and thus should receive special treatment with respect to its position on the computational domain, or the order of the solution on this entity.
With this fine decomposition of the element, we can see that it is possible to allocate the right amount of memory for each element on an hybrid mesh. Data will simply be distributed across this hierarchical set of entities.

Refinement of the mesh

As we saw in the previous chapter, some methods use mesh refinement to modify the mesh in order to follow the propagation of the information in the computational domain. In practice it can be as easy as a modification of the coordinates of some vertices, or it can require a complete reconstruction of a sub-mesh in the domain. The most complicated approach to refinement involves a modification of the topology of the graph associated with the mesh. The refinement process can greatly reduce computational cost by focusing the resources on the area where the information is propagating and by reducing the precision of the computations on the areas where nothing is happening.

2.4.2 Proposed architecture

Based on the above with respect to the separation of concerns and to the different needs for our platform, Figure 2.1 exhibits the retained architecture for the `AeroSol` platform.
The platform will be built according to the following design principles :

- the mesh object has the knowledge of the geometric data distribution. It will provide a set of methods to iterate over the mesh ;
- the time scheme objects are high-level objects simulating the time stepping ; These objects will manipulate opaque objects embedding solutions and data on the mesh, as well as an abstract matrix. They will do basic manipulations on opaque data objects and pass them to the underlying spatial scheme ;
- the spatial scheme knows the data distribution of the solution over the mesh. Using the iterators provided by the mesh objects, it will be able to iterate over the mesh to reconstruct elements. The knowledge of the data distribution makes this object a good candidate to interact with the linear solver interface. It will manipulate an abstract object representing a matrix and call methods like fill-in prediction, assembly, inversion, or solve to complete each step of the resolution of the linear solver. It will use the the integration objects to compute quantities on elements ;
- the integrator objects are low-level classes for computing the integrals over certain entities. The integrator uses its collection of finite element to compute integrals in a reference

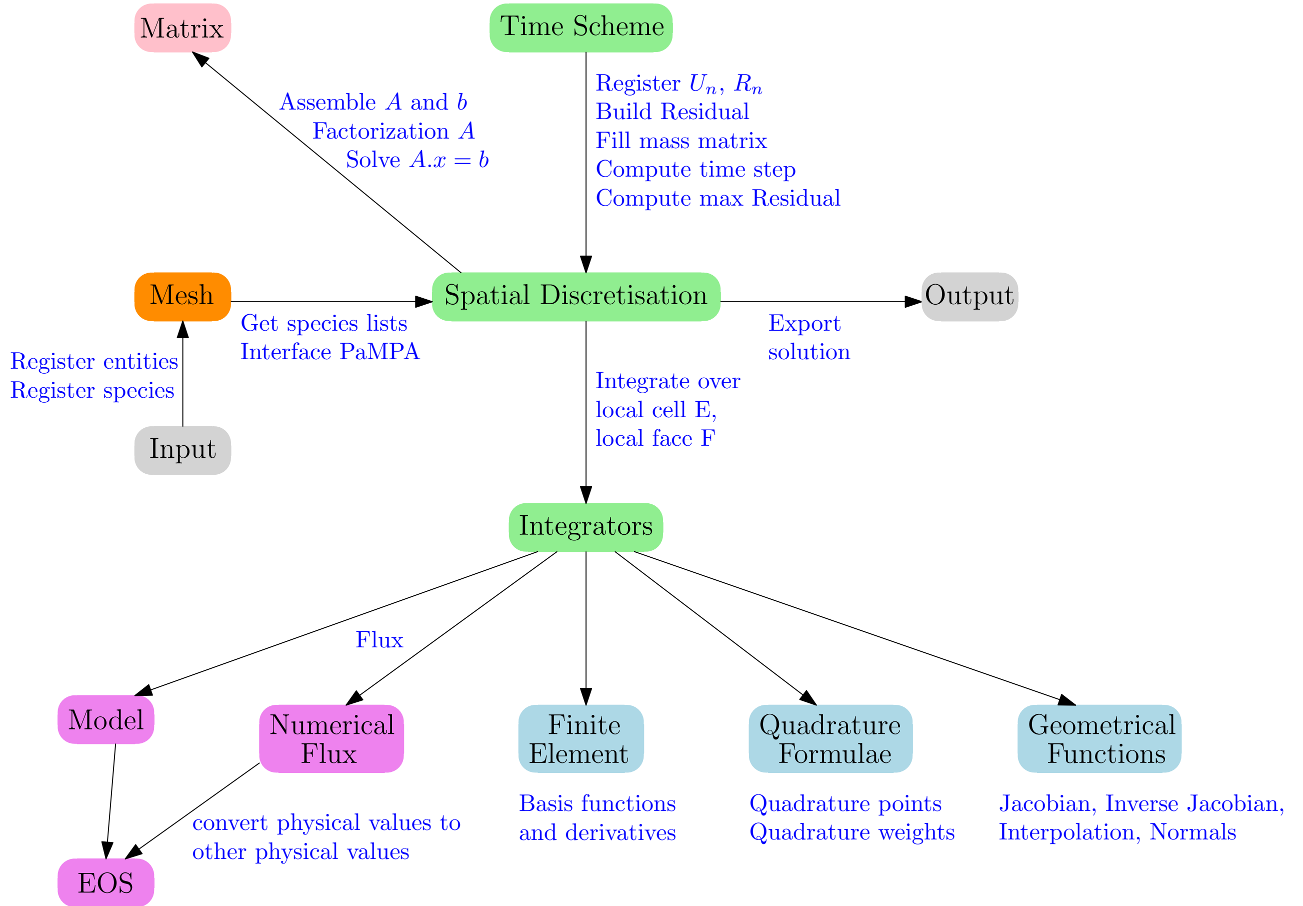


FIGURE 2.1 – Detailed architecture of the `AeroSol` platform.

element ;

— finite element, quadrature formulae, and geometrical functions are a collection of classes defining the finite element, and the geometric methods used by the integrators. Quadrature formulae will provide the coordinates of the integration points and a weight associated to this point, on a reference element. Finite element object will use these coordinates to evaluate polynomial functions of shape functions and gradient of the shape functions. Geometrical function objects will provide methods to transform the element in the physical space to the reference space ;

— two modules to interact with the end-user will be added. An input object setting the mesh, the solution and some constant of the domain, and an output module used by the spatial scheme object. It will be used to export the solution in various formats.

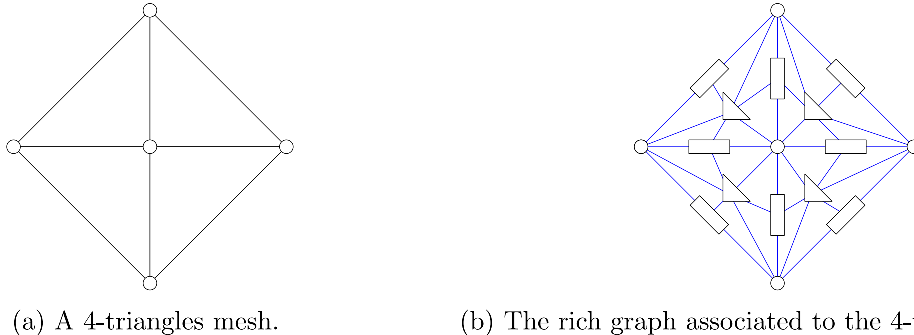
2.4.3 Technical requirements

Modern architectures used in high performance computing exhibit a complex structure of memory and processing units. Supercomputers are composed of multiple nodes, which can have accelerators. They result in the availability of a lot of processing units of different speeds, and of multiple levels of memory with different access times. In order to support the heterogeneity of the modern and future architectures, we need to find a way to abstract the underlying architecture.

2.5 Delegating the functionalities

Developing the whole software would have taken a lot of time, and this thesis had a limited time to frame. Moreover, some of the functionalities required are provided by tools developed either in our team or in collaborating teams. The development of this platform was a good opportunity to build an end-user application making a set of existing tools collaborate. Among these tools, we here present those that answer some of the technical requirements we have for our platform.

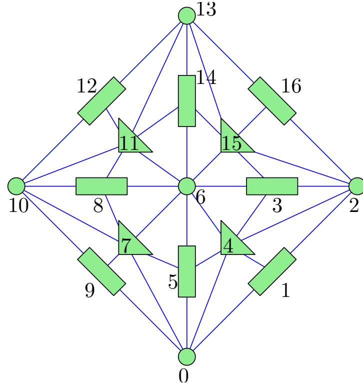
2.5.1 PaMPA



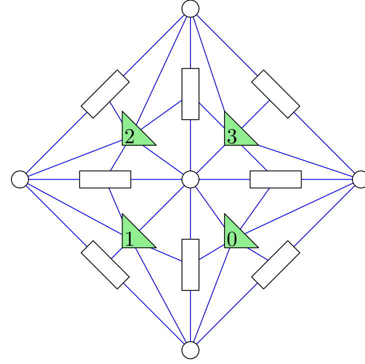
(a) A 4-triangles mesh. (b) The rich graph associated to the 4-triangles mesh.

FIGURE 2.2 – Transformation from a simple 4 triangles mesh (on the left) to a rich graph (on the right).

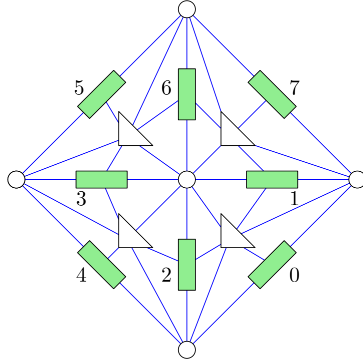
PaMPA [52, 53], standing for Parallel Mesh Partitioning and Adaptation, is a library developed at Inria Bordeaux Sud-Ouest. PaMPA manages unstructured meshes across the processors of a parallel machine. The idea is to relieve the developer from writing again and again the same error-prone functionalities of his applications : the mesh module, the data communication layer, the re-meshing module. PaMPA manipulates meshes as graphs. As depicted in Figure 2.2, vertices of this graph are entities of the mesh (Figure 2.2a), namely elements, faces, edges, or nodes, and the edges of the graph represent the link between entities. Entities are associated with an integer which represents the type of the entity. It can give simple information on the type of the entity (element, face, or vertex), but it can also provide rich information like the shape of the entity, or the order of the solution on this entity. We will discuss later (in section 2.6.1) the way we use this integer. In Figure 2.2b we can see that an element, such as a triangle, is connected to all of its faces (depicted as rectangles), and points (depicted as circles). A face is connected to its edges and its points, etc. Elements can be connected so as to characterize the neighborhood.
To each type of entity, can be “attached” some data. One can select an entity type, such as the element, and attach to each element a piece of memory of an arbitrary size. The memory area must be associated with a MPI_Datatype. The piece of memory will be divided into three segments. The first corresponds to local entities of the domain ; the second stores boundary entities, and the third corresponds to the ghost entities in the overlap.
PaMPA will do a parallel graph partitioning using the parallel library Pt-Scotch [33] on the input graph. The partitioning will ensure a good load-balancing of the domains, as depicted in figures 2.4a and 2.4b, and inside a domain, a good numbering of entities, thus enhancing the locality of data accesses when manipulating data attached to the entities. In addition to the domain decomposition, PaMPA will manage the overlapping parts of the graph : the boundaries, boundary entities will be identified and replicated on remote domains to ensure the capacity to do all the computations on the remote domains. The data attached to these ghost entities will



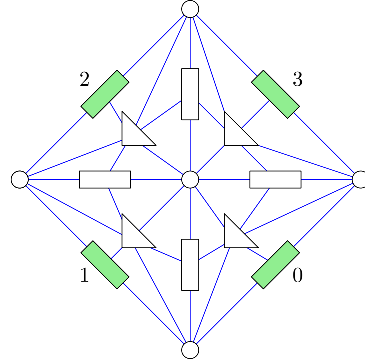
(a) All the entities of the graph with a global index.



(b) The elements with their per-entity numbering.



(c) All the faces of the graph with a per-entity index.



(d) A sub-set of faces corresponding to the border faces, with their per-sub-entity numbering.

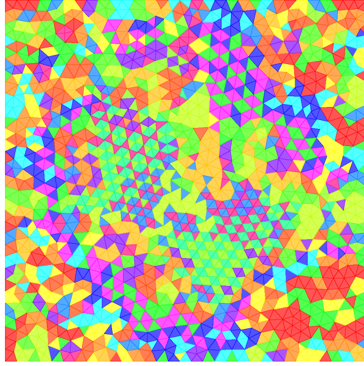
FIGURE 2.3 – Different examples of iterators provided by `PaMPA`.

be duplicated on both domains, and `PaMPA` will handle communications to synchronize the data when the developer asks for it. Finally, The depth of the overlap is a user-defined parameter.

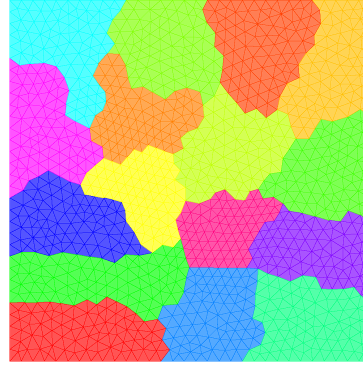
`PaMPA` can also do a sequential re-meshing of the graph. It uses the `MMG3D` library [23].

`PaMPA` will provide two levels of iterators to the application. With the first level of iterator, the developer will be able to visit every entity of a certain type. When visiting all the entities of a given type, `PaMPA` will provide a numbering starting from 0 to N_{ent}. In Figure 2.3 we can see different sub-sets of entities. In Figure 2.3a, all the entities are visited, so we have a global indexing for the entities. In Figure 2.3b, we restricted the iterator to the element entity type. We have elements numbered from 0 to 3. In Figure 2.3c, the chosen sub-set of entities is the faces. As in the previous iterator example, we will visit all the faces of the mesh, with a restricted numbering. Finally, Figure 2.3d shows an iterator corresponding to border faces. The interesting feature here is that `PaMPA` allows to create a 2-level hierarchy between entity types. It means that at the top level hierarchy, one can put the element type, the face type, and the vertex type. At the lower level, one will put all the variants of elements like triangle, quadrangular, or the variants of faces like border faces and inner faces. This will be useful when boundary conditions are needed. The programmer has an easy access to border faces. The interest in differencing the entities is to allocate the exact amount of memory needed. The top-level entity type will be referred to as the group type, and the lower-level type will be denoted as the entity type. We will see later how we manage memory on top of this graph.

When visiting an entity, the developer can use the second level of iterators to focus on the entity and visit entities linked to it. Thus, we can access data to reconstruct a cell, or compute the



(a) Mesh after the distributed reading phase and before the partitioning and redistribution phases



(b) Distributed mesh after repartitioning and redistribution.

FIGURE 2.4 – The partitioning and the redistribution of the mesh entities.

isobarycenter of an element.

2.5.2 Runtime systems

As we saw in the previous subsection, the `PaMPA` library handles our mesh, our data, and gives us a domain decomposition for the distributed memory layer. What we need now is a platform for handling the shared memory layout. The main difficulty here is the heterogeneous aspects of a computer. Multi-threaded programming for a computer with multiple GPUs of possibly different generations, or vendors is complicated. This is why we chose to rely on a library hiding the specificities of the underlying architectures and to submit work to this library using an insert-task programming model. The first step to use these libraries is to re-write our algorithm as tasks, working on data. Our algorithm becomes a directed acyclic graph where the tasks are the computation elementary operations, and the edges between tasks characterize the data dependencies between these tasks.
Among our collaborators and the teams working at `Inria`, we have two runtime systems available. The first one is the `StarPU` library, developed within the RUNTIME team. This library uses an insert-task model and a global vision of the of the directed acyclic graph to take task scheduling decisions based on the reduction of the critical path to reduce the overall completion time. The second library we chose to use is `PaRSEC`. Instead of submitting tasks to `PaRSEC`, the developer will submit an entry point to the directed acyclic graph, and the platform will discover the graph dynamically when a task is completed and its dependent tasks become executable. This dynamical discovery of the graph will allow `PaRSEC` to scale when graph size increases. The specificities of each platform will be discussed in the next chapter (see Section 3.1).
To summarize, by re-writing our algorithm as tasks, and by providing implementations of the tasks for each hardware device, the runtime system will be able to perform our computations on an heterogeneous computer. Thanks to the fact that all the data consistency mechanisms, and the scheduling strategies are delegated to the runtime system, we can focus on the implementation of our tasks and kernels. Theorically, performance portability is ensured at a reduced cost ; when a new hardware is released, only tan adapted optimized implementation of our tasks will be required.

2.5.3 Linear Solver

In many numerical applications, methods manipulate data to build matrices and, vectors and, using either an internal module or an external library, they invert a linear system to update

their solutions so as to progress in time. The idea of our work was not to develop our own linear solver library. This is a large field of ongoing research, and it seems wise to rely on existing tools developed by experts of this domain rather than trying to re-invent the wheel. There are multiple teams working in this field, and some of them have on-going collaborations with our team. This is why we chose to rely on their library for our platform, in addition to generic toolboxes like `PETSc`. The `MUMPS` library has been linked with our platform. A generic interface for sparse matrix has been designed to interface each library with our platform. Hence, the developer of an `AeroSol` application only manipulates an opaque object, and does calls generic methods to progress through the resolution of the linear system.

2.6 Functional implementation

In this section, we will present some of the solutions we implemented to solve the problems presented in Section 2.4.

2.6.1 Specializing the entities

A central object in the platform is the `EntType` object. It is necessary to present this key element of the platform. As we saw before, `PaMPA` handles a hierarchy of types, and each entity will have a group type and an entity type. The entity-type attribute is the parameter that characterizes our entity. Being restricted to use only one integer to represent the entity, we will have to embed all the informations in this integer. To do so, we used available bits to encode the needed informations :

	Degree	Domain	Shape	Curvature
Bits interval	[0;4]	[5;6]	[7;10]	[11;13]
Degree 5	10100	00	0000	000
Hexahedron	00000	00	1001	000
Hexahedron 5	10100	00	1001	000

Inner Hexahedron of degree 5 = entity type 1189

FIGURE 2.5 – Decomposition of the EntType integer. Example given for building a straight hexahedron of order 5.

Figure 2.5 presents a table showing how the `EntType` attribute is decomposed. From bit 0 to bit 4, we encode the degree of the entity. Bits 5 and 6 encode the domain information which answer is the following questions : 1) Is this entity inside the domain ? 2) Is it a border face ? Is it a periodic border face ? Then bits 7 to 10 characterize the shape of the entity, bending point, point, line, triangle, quadrangular, tetrahedron, pyramid, prism, hexahedron ? The last interval of bits, from 11 to 13, encodes the curvature degree of the element. It starts from 0 for a straight element. The remaining bits are available for future use.
When loading the mesh, all the `EntType` are stored in the mesh object. This object keeps a set of lists of `EntType`, each list containing the `EntType`'s of a given dimension. These lists can be accessed either by the dimension of the elements in it, or by their codimension. The codimension is defined as follows :

$$codimension = dimension - dimension_{entity}, \quad (2.1)$$

Shape table	
Shape	Bit values [7 :10]
Bending point	1000
Point	0100
Line	1100
Triangle	0010
Quadrangular	1010
Tetrahedron	0110
Pyramid	1110
Prism	0001
Hexahedron	1001

TABLE 2.2 – Values corresponding to element shapes.

Domain table	
Domain	Bit values [5 :6]
Border	00
Inner	10
Periodic	01

TABLE 2.3 – Values corresponding to domain conditions.

where *dimension* is the dimension of the system, and $dimension_{entity}$ is the dimension required to represent the entity. A codimension of 0 represents all the elements of the mesh while a codimension of 1 will represent all the faces.

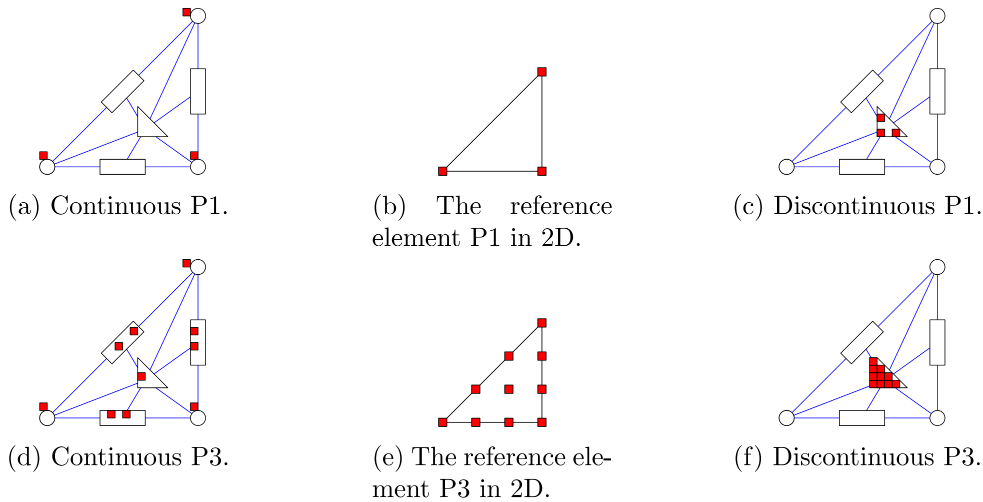
Entity E = 2756	00100	01	1010	100	
Degree_Mask	11111	00	0000	000	
E & Degree_Mask	00100	00	0000	000	Degree 4
Domain_Mask	00000	11	0000	000	
E & Domain_Mask	00000	01	0000	000	Border
Shape_Mask	00000	00	1111	000	
E & Shape_Mask	00000	00	1010	000	Quadrangle
Curvature_Mask	00000	00	0000	111	
E & Curvature_Mask	00000	00	0000	100	Curvature degree 1(+1)

FIGURE 2.6 – Use of binary masks to decompose an `EntType`.

The `EntType` will be used as a key when manipulating our data containers. Figure 2.6 shows an `EntType` of value 2756, and we see how the use of binary mask allows us to extract information about this entity. This entity type indeed corresponds to a curved (degree 2) quadrangle with a solution of degree 4, located on the border of the mathematical domain.

2.6.2 Solution Object

This object is user-defined. The platform provides a default one. Its function is to return the number of unknowns for an entity type. For example, in Figure 2.7, we show two examples of decompositions of triangles. In Figure 2.7b and 2.7e, we provide two triangles with the corresponding distribution of unknowns. Triangle P1 has 3 unknowns, while triangle P3 has 10.



(a) Continuous P1.

(b) The reference element P1 in 2D.

(c) Discontinuous P1.

(d) Continuous P3.

(e) The reference element P3 in 2D.

(f) Discontinuous P3.

FIGURE 2.7 – Hierarchical decomposition of a triangle at different orders with a continuous or a discontinuous solution.

In Figure 2.7a, we see that the 3 unknowns are localized on the entity of type point corresponding to circles in the figure. In Figure 2.7c, all the unknowns are stored on the element entity, the triangle. When increasing the order of the solution, in Figure 2.7f, the 10 unknowns are still stored on the element entity. On the contrary, in Figure 2.7d, the unknowns are distributed across several entities. The element entity receives 1 unknown, each face receives 2 unknowns, and each point receives 1 unknown. The information related to this distribution is given by the `Solution` object. This illustrates how flexible our platform is when dealing with the repartition of the unknowns on the different elements.

2.6.3 Cell reconstruction

To do all the computations required by the mathematical method, we need a method to gather the scattered information. As we saw previously, when the order of the solution increases, the data are scattered on a larger number of entities. Gathering these data, and ordering them in order to perform computations correctly, can be an issue. This is a key method of the spatial scheme object as its responsibility to provide a valid vector of information to its integrator objects.

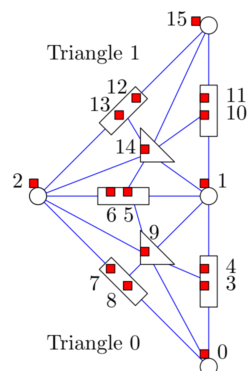


FIGURE 2.8 – Example of the orientation issue with two triangles.

First, let us start with an example in 2 dimensions, in Figure 2.8. In this example, the entities

have unknowns linked to them, and these unknowns received a global index.

Triangle 0 Degrees of Freedom :

0, 1, 2,	3, 4, 5, 6, 7, 8,	9
Points	Faces	Element

Triangle 1 Degrees of Freedom :

1, 15, 2,	10, 11, 12, 13, 5, 6,	14
Points	Faces -1	Element
1, 15, 2,	10, 11, 12, 13, 6, 5,	14
Points	Faces	Element

FIGURE 2.9 – The corresponding lists of DoFs.

If we try to build the list of DoFs with just the informations of the graph, we obtain an error as depicted in Figure 2.9. The list for triangle 0 is valid, all the DoFs are well sorted, but for triangle 1, an error appears on the face shared with triangle 0. The DoFs are inverted, because we are missing some information regarding orientation. This information will be added to the graph in the form of an integer vector of a size equal to the number of faces of the element, that will be attached to the element. It will give us the orientation information required to sort the DoFs of the face when merging the list with the others.

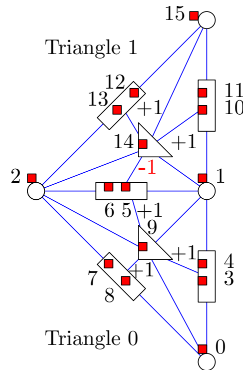


FIGURE 2.10 – Construction of the orientation.

When we load the mesh, we will decide that the first element referring to a face will be the owner of this face, as depicted in Figure 2.10. The link between them will have the label +1. A face can be connected to 1 or 2 elements. So, when we will build the other element, the relation between this element and the face will have the label -1.

This problem is slightly more complex when we deal with a three dimensional mesh. In Figure 2.11, we have, on the left, a simple high order hexahedron. We focus on face number 2. This face contains four degrees of freedom : $\{1, 2, 3, 4\}$. This face is shared with another hexahedron, on the right part. This hexahedron knows this face as its face 0. It could be anything between 0 and 5. Yet, depending on the order of the vertices that where found in the mesh file, face 0 can be registered in 4 different ways. When registering an hexahedron, we read 8 integers. The first four integers represent face 0. So, if this hexahedron is composed of vertices $\{10, 11, 12, 13, 14, 15, 16, 17\}$, face 0 is defined by its 4 vertices $\{10, 11, 12, 13\}$. But it can be rotated three times, and face 0 may become $\{11, 12, 13, 10\}$, $\{12, 13, 10, 11\}$, $\{13, 10, 11, 12\}$. And,

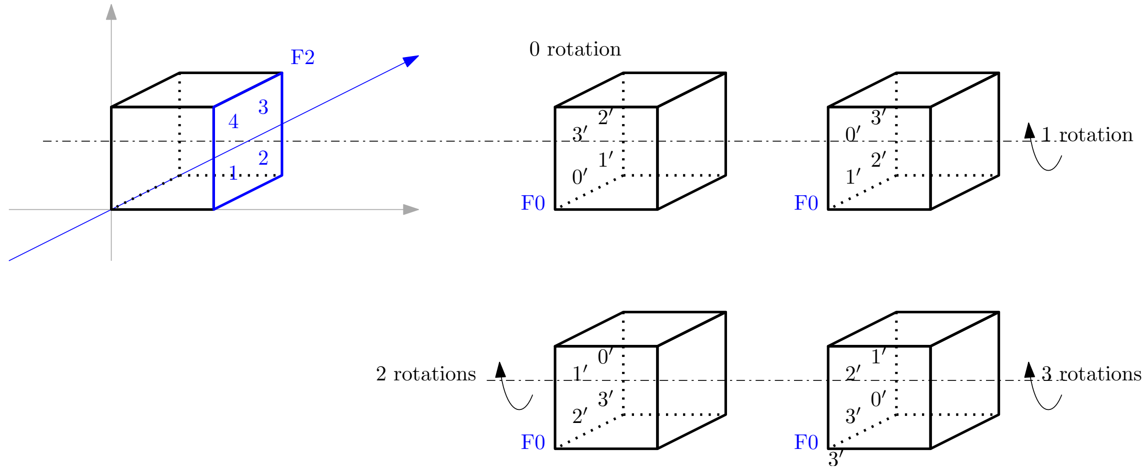


FIGURE 2.11 – Orientation and Rotation issue.

for each rotation, the order of the merging of the degrees of freedom list will be different. In the figure, when there is no rotation, the degrees of freedom list $\{0', 1', 2', 3'\}$ will correspond to $\{1, 2, 3, 4\}$. In the second case, with one rotation, the list becomes $\{2, 3, 4, 1\}$. In the third case, two rotations, the list to merge is $\{3, 4, 1, 2\}$. And finally, in the fourth case, the list to merge is $\{4, 1, 2, 3\}$. To avoid this issue, we added another information to the relation between an element and its faces : the rotation. The element which has an orientation of $+1$ will have a rotation of 0. The other element (orientation -1) will compare the indices of the face it has with the indices of the owner (the element with orientation $+1$), and deduce the rotation to apply. The rotation will be an integer ranging from 0 to $(number_of_indices - 1)$. With these additional data, the reconstruction method can operate and provide a flat vector of data to the integrator objects.

2.7 Computational implementation

As we saw in the proposed architecture, our platform has a vertical hierarchy of classes, or objects. A class asks the class(es) below to work for it, uses the result to contribute to its own computations and sends its result to the above class. We will start our description of the different classes from the bottom of this hierarchy, where there are no dependencies in computation, and see what is possible to compute when considering to the upper classes.

2.7.1 Finite elements collection

As we saw, the lower levels of our hierarchy are populated with our finite element classes. The Quadrature Formulae classes provide weights and coordinates of the integration points that will be used to compute integrals. The number of weights and points grows along with the order of the polynomial we want to represent.
The Finite Element classes are another collection of methods. There are two important methods in a Finite Element class. Two methods evaluate polynomials. The first one evaluates the shape function polynomials of each degree of freedom on the given coordinate. It returns a small vector of the size of the number of degrees of freedom. The second evaluates the gradient of the shape function polynomials of each degree of freedom and returns a small matrix of the size of the degrees of freedom times the dimension of the system.
The Geometrical functions are a collection of methods that compute geometrical values, ba-

rycenter coordinates, inscribed circle center and radius. Also, they provide the transformation from the physical space to the reference space, and its inverse.

2.7.2 Equations of state, models and, numerical fluxes

The basis class is the Equation of State class (EOS). This class first defines the number of equations for one degree of freedom. As the number of equations increases, the size of the global problem is multiplied by the number of equations. This class also provides methods to convert physical values into other physical values. Such methods are used by the model and numerical flux classes to compute quantities used by the integrator classes

2.7.3 Integrator classes

The Integrator classes are the tools used to compute quantities over different entities. Specialized by the Equations of state, the Models and the Numerical fluxes, they compute what is required by the Spatial scheme.
Like we saw previously, the Finite element polynomials are evaluated on a reference element using the coordinates provided by the Quadrature formulae. This information is stored in the Integrator and will be used when needed with a faster access than by re-evaluating a function. We saw that the Geometrical functions classes are a collection of methods. It means that they inherit from the same interface. Consequently, calling these virtual methods in the inner loops of our platform is expensive. This is why to be faster, we stored a function pointer in the integrator so as to speed-up the process of finding the function to execute, depending on the entry entity.

2.7.4 Spatial scheme

The Space Scheme is the central object of our platform. It has the knowledge of the mesh and of its geometrical data. It can iterate over the mesh, visit the entities in any order, use the connectivity of each entity to move to a neighboring entity. With the cell reconstruction method, it can gather scattered information in a flat vector and send it for computation to an Integrator. It comprises a toolbox of integrators to compute quantities on elements or faces, depending on user requirements. Every contribution computed can be assembled in the linear system using the Matrix interface. Using its knowledge of the data distribution of the solution, it can export the solution to the end-user.
More importantly, this object hides the parallel implementation of the method. It can compute a domain decomposition, and the methods of the scheme can process a sub-domain. In each sub-domain, to handle the shared memory problem, we can either we rely on graph coloring to avoid race conditions when scheduling computations, or else we can use our runtime systems and submit tasks. The graph coloring approach is discussed in Section 3.6 and will not be developed here.

2.7.5 Time scheme

This object simulates the time stepping of our simulations. Among its features,

- It can create opaque data objects and provide them to the Spatial scheme object for initialization.
- It has access only to the Spatial scheme interface. It provides it data objects for update, and can perform simple operation on these data.
- It can manipulate the linear system through the Spatial scheme interface.

2.8 Experiments and results

We compared our platform to the `Aghora` platform to illustrate our design choices about our platform.
Measurement were done on the AVAKAS cluster [4], which is composed of 264 nodes of bi-pro hexa-core INTEL Xeon x5675 CPUs running at 3,06 GHz. Each node has 48GB of RAM.

2.8.1 Numerical tests

Due to the fact that both the Aerosol and `Aghora` libraries are not yet mature, a simple test was chosen for comparing libraries. In `Aghora`, only three dimensional computations are possible. One and two dimensional computations are also possible, but by extruding one layer of cells in the z direction (and also in the y direction in one dimension). In `AeroSol`, true one and two dimensional computations can be considered, but geometrical functions, finite elements functions, and mesh I/O were not yet available for three dimensional shapes. For performing fair comparisons, the `Aghora` point of view was adopted, and single layer, hexaedral meshes were used for performing a two dimensional test. `AeroSol` later was extended to three dimensions.

2.8.2 Yee vortex

We consider the convection of an isentropic vortex in a 2D uniform and inviscid flow [75] with conditions $\rho_\infty = 1$, $\mathbf{u}_\infty = \mathbf{e_x}$ and $T_\infty = 1$. The domain is the unit square $\Omega = [0,1]^2$ with periodic boundary conditions. The initial condition consists in a perturbation of the uniform flow which reads in primitive variables :

$$
\begin{aligned}
\rho(\mathbf{x},0) &= \left(1-(\gamma-1)\left(\frac{M_\infty M_v r_c}{2}\exp(1-\tfrac{r^2}{r_c^2})\right)^2\right)^{\frac{1}{\gamma-1}} ,\\
u(\mathbf{x},0) &= 1 - M_v y \exp(1-\tfrac{r^2}{r_c^2}) ,\\
v(\mathbf{x},0) &= M_v x \exp(1-\tfrac{r^2}{r_c^2}) ,\\
\mathrm{p}(\mathbf{x},0) &= \frac{1}{\gamma M_\infty^2}\rho(\mathbf{x},0)^\gamma ,
\end{aligned}
$$

where $r^2 = (x-x_c)^2 + (y-y_c)^2$ denotes the distance to the vortex center $(x_c, y_c)^\top$, r_c and M_v are the radius and strength of the vortex, and M_∞ is the Mach number of the freestream flow. The exact solution of this problem is a pure convection of the vortex at velocity $\mathbf{u}_\infty$.
Numerical results are obtained for physical parameters $M_\infty = 0.5$, $M_v = 4$, $r_c = 0.1$, and a final time $T = 1$.
As the periodic boundary conditions were not yet available in the `AeroSol` library, the test was slightly modified as follows : instead of adding a mean flow to the vortex, the initial state was taken as the state at infinity, without any velocity. On the boundaries, the values of the vortex are weakly applied. From a computational point of view, the workload is nearly the same, except for the communications that are needed for periodic boundary conditions in the case of the unsteady vortex.
Table 2.4 presents a weak scalability analysis, in which we evaluate the elapsed time observed for the global computation and for a fixed number of 24×24 hexaedral elements per computing core. Results are shown as a function of the number of cores. The relative increase, E, of the elapsed time with respect to the single core is also indicated, on Figure 2.12. Note that the

4. https ://redmine.mcia.univ-bordeaux.fr/projects/cluster-avakas/wiki

Table 2.4 – Yee vortex problem : time/proc. [s] with Aerosol and Aghora for 3000 Runge-Kutta sub-steps.

	# cores	1	4	16	64	256	1024	2048
$p = 0$	Aerosol	202.52	205.32	209.91	216.68	222.69	224.01	225.52
$p = 1$	Aghora	12.67	14.02	15.69	15.88	16.42	18.43	17.97
	Aerosol	977.3	977.1	999.3	1035.6	1074.5	1074.3	1083.9
$p = 2$	Aghora	94.12	96.90	100.08	100.72	103.28	112.91	116.75
	Aerosol	5938.6	5923.2	6047.1	6307.7	6504.6	6501.0	6640.5
$p = 4$	Aghora	1401.5	1421.3	1430.2	1435.5	1439.1	1451.4	1370.8

memory requirement increases with the polynomial degree : the number of degrees of freedom per core is 4608, 15552, and 36864 for $p = 1$, 2, and 3, respectively. We observe that the solver scales nearly perfectly for $p = 4$, while results deteriorate for lower polynomial degrees $p = 1$ and 2. The relative part of communications compared to the numerical scheme decreases as the polynomial degree increases : from 1.1% for p =1 to 0.1% for p=4, as we can see in Table 2.5. As a consequence, the high frequency of communications between processes for a large number of cores becomes insensitive for large p-values. `AeroSol` has a stronger work load for boundary sides because, in `Aghora`, the boundary sides of the top and the bottom are ignored in the boundary side loop, whereas in `AeroSol`, a freestream boundary condition was used.

We observe in Table 2.4 that `Aghora` has lower execution time for a same number of RK sub-steps. Using an analyzer, we found the following reasons for the lower efficiency of `AeroSol`.

— Firstly, `AeroSol` is a finite element code that can both use continuous and discontinuous finite elements. In continuous finite elements, a degree of freedom is shared by many elements, if it is located on edges, faces, or points. Consequently, a connectivity table must be built for knowing which degrees of freedom are shared by which elements. In `AeroSol`, this connectivity table is never stored, and its computation is irrespective of continuous and discontinuous finite elements (thus, it was not optimized for discontinuous finite elements). We found that the computation of this connectivity table is nearly three times more costly that the computation of the local residual. We are currently developing a version in which the connectivity table is stored.

— Secondly, in `AeroSol`, the geometry (e.g., the local Jacobian of elements) is never stored. This means that, at each time step, and on all quadrature points, all the needed geometry is computed. In the computation of the local residual, we evaluated that 35% of the time elapsed in the computation of the local residual is for computing geometrical functions. We will develop in the near future a version in which the geometry is stored.

— Lastly, we did not use any optimization option for compiling `AeroSol`.

Table 2.5 gives a detailed analysis of the relative costs of different stages of the numerical algorithm. Results are given for one core. The implementation of surface and volume integrals for fluxes clearly represent the most expensive stage and its relative cost increases with p. This result is in agreement with the high scalability observed for high p-values as local operations strongly dominate communications.

Finally, we wanted to compare the results obtained with different MPI libraries. However, `AeroSol` needs an MPI library for which the `MPI_THREAD_MULTIPLE` is supported (i.e., if the process is multi-threaded, multiple threads may call MPI at once with no restrictions), and such a functionality is only available with MVAPICH2 on Avakas. That is why we show the results of this comparison only for `Aghora` and not for `AeroSol`, see Table 2.6. Note that an important part of this execution time can be due to the initialization of the MPI universe, see Table 2.7.

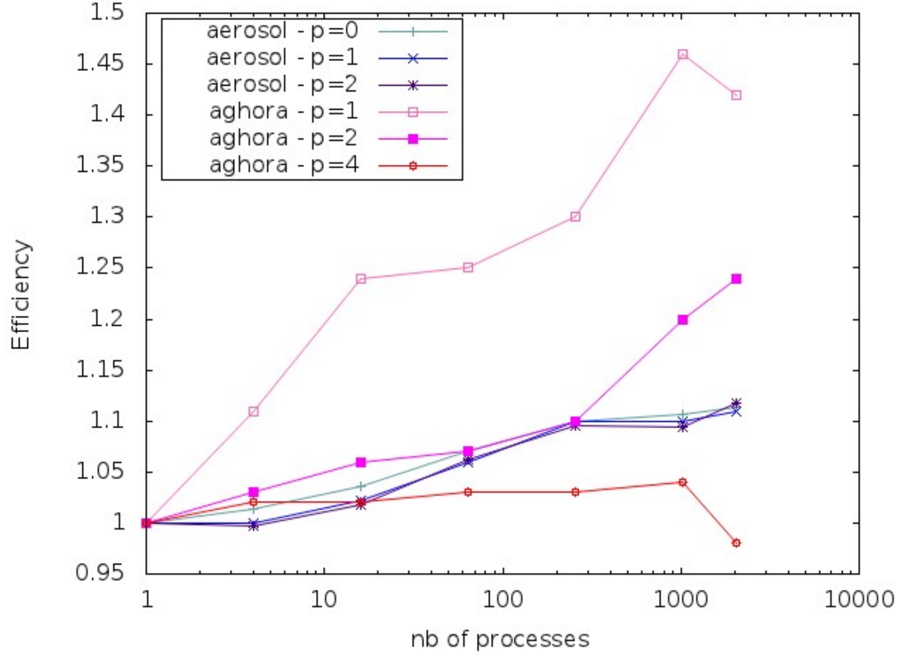


FIGURE 2.12 – Comparison of the weak scalability obtained for `AeroSol` and `Aghora` with the library `mvapich2`. All the results show good scalability. The least scalable case is `Aghora` for $p = 1$, and the most efficient is `Aghora` for $p = 4$. Nevertheless, if we consider the results given in Table 2.4, we see that `AeroSol` is much more costly if execution time is considered. This explains that weak scalability is better, as `AeroSol` has more computations for overlapping communications.

TABLE 2.5 – Yee vortex problem : relative costs in percent of different stages of the numerical algorithm per physical time step.

p	0	1		2		4
	`AeroSol`	`AeroSol`	`Aghora`	`AeroSol`	`Aghora`	`Aghora`
boundary surface integral	31.1	25.0	4.2	16.5	3.2	2.2
internal surface integral	51.3	36.4	57.5	21.2	46.8	34.6
volume integral	17.3	37.6	36.5	61.4	49.3	63.0
mass matrix inversion	0.3	1.0	1.8	0.8	0.7	0.2

	# cores	1	4	16	64	256	1024	2048
IntelMPI	time/proc. [s]	282.78	288.55	299.80	303.38	309.13	343.27	427.47
	E [-]	1	1.02	1.06	1.07	1.09	1.21	1.51
OpenMPI	time/proc. [s]	282.92	291.50	299.53	302.27	309.67	334.44	341.93
	E [-]	1	1.03	1.06	1.07	1.10	1.18	1.21
MVAPICH2	time/proc. [s]	282.37	290.71	300.25	302.17	309.85	338.73	350.26
	E [-]	1	1.03	1.06	1.07	1.10	1.20	1.24

TABLE 2.6 – Comparison of execution time by core for a polynomial degree equal to 2 (9000 RK sub-steps). We observe that the most efficient execution is obtained for the OpenMPI library on AVAKAS.

# core	256	1024	2048
IntelMPI	15	122	5001
OpenMPI	05	010	0020
MVAPICH2	33	122	0536

TABLE 2.7 – Average elapsed time [s] required for initialization of the MPI universe.

2.9 Conclusion

In this chapter, we presented a state of the art of software platforms that provide features close to the ones that we wanted to implemented. Our goals were the accessibility of the platform and, the ability for anyone to contribute to the platform. This is an ambitious goal, but the decomposition of the platform in modules interacting by simple interfaces, and a set of macros simplifying reading and writing will help to improve this point. The second goal was the flexibility of the platform. The platform uses modern tools for compilation and testing. All the basic classes, model, numerical fluxes, finite elements, and geometric functions are heavily tested. There are some scenari to test the time and spatial schemes although it requires an outside look to validate the test. The third objective was the mathematical abstraction of the methods. We hid the difference between the continuous and the discontinuous methods in the cell reconstruction methods. All our integrators work on local cells without knowing the methods. The results presented in the previous section are encouraging, and we are satisfied by the performance of the platform. Choosing to be generic on every aspect, is anti-performant by definition. We lose many easy optimizations and yet, the platform exhibits a good behavior.
In the next chapter, we will discuss the ways to parallelize finite element methods in a shared memory environment, and we will present the second contribution of this thesis regarding the assembly operation over a runtime system.

Chapitre 3

The assembly operation

Contents

3.1 Introduction . . . 41
3.2 Related Works . . . 42
3.3 The StarPU runtime system . . . 45
3.4 PaRSEC . . . 46
3.5 Finite Element Method . . . 47
3.5.1 Computation of contributions . . . 48
3.5.2 Taskified Finite Element Method using StarPU . . . 49
3.5.3 Taskified Finite Element Method using PaRSEC . . . 50
3.6 The assembly operation . . . 50
3.6.1 A new taskified Assembly Operation . . . 56
3.6.2 Scheduling strategies for taskified assembly operations . . . 58
3.7 Experimental results . . . 59
3.7.1 Experimental setup . . . 59
3.8 Conclusion . . . 61

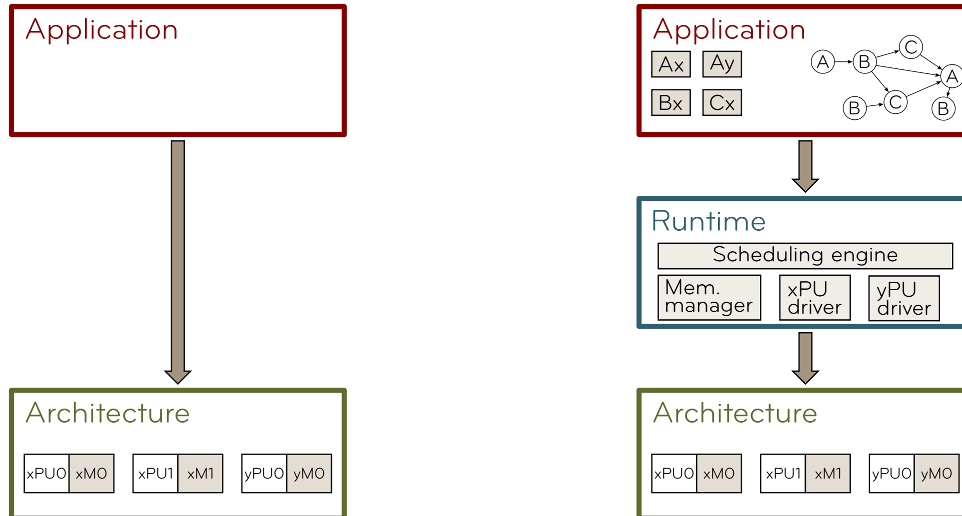
3.1 Introduction

Ten or twenty years ago, the increase in performance relied exclusively on the augmentation of clock frequency, and on a higher degree of instruction level parallelism. Sadly, one attained a physical limit, the outcomes are not worth the power needed and the heat dissipation required to sustain the augmentation of performance. So, rather than increasing the single-thread performance, one started to miniaturize and aggregate multiple processing units on the same chip. Multi-core processors are nowadays ubiquitous, and the evolution of computers is driven by a run towards higher and higher numbers of cores per chip. This trend was largely anticipated by Graphical Processing Units (GPUs).
The GPUs, well known for their use in graphics applications, comprise hundreds of slow cores, and a fast memory, and have the best ratio of performance with respect to cost. The first attempts to use GPU in the context of scientific computing emerged in the 90's [64]. Recent GPUs are relatively powerful and can outperform CPUs in term of data processing and memory bandwidth. By comparison, the latest NVIDIA Tesla card, the K40, has a peak performance of 1.43×10^{12} floating point operations per second for double precision operations and a memory bandwidth of 288 GBytes/s. The GPU seems to be the perfect candidate for scientific

computing,. Yet developing an application that makes efficient use of a GPU is more complicated. Interfaces like `CUDA`, or `OpenCL` have quickly evolved to help the developers. The GPGPU, General Purpose GPU, gathers all the work in the domain to ease the use of GPUs. Inspired by all the performance of these devices working for the scientific community, INTEL released a new co-processor device as under the INTEL MIC brand, the Xeon Phi. It comprises 60 optimized cores, handling 240 threads, for a peak performance of 10^{12} floating operations per second. Each core uses the x86 instruction set, and its design is based on the old and heavily optimized P54C, one of the first Pentium processors from the mid '90s. The main advantage of these devices is their low consumption, and their cheap price, for a high parallelism. More recently, INTEL is trying to make the high performance accelerators and regular cores converge into complex processors such as the Ivy Bridge processor where the graphical processor is integrated into the same chip as the general purpose cores.
The increasing parallelism and complexity of hardware architectures requires the High Performance Computing (HPC) community to develop more and more complex software that combine multiple technologies in order to exploit all the available computational resources. To achieve a high level of optimization and exploit this potential, not only the related codes are heavily tuned for the considered architecture, but the software is often designed as a single entity that aims at simultaneously coping with both the algorithmic and architectural needs. If this approach may indeed lead to extremely high performance, it is at the price of a tremendous development effort, a lesser portability and a poor maintainability.
Alternatively, a more modular approach can be employed. In this case, the numerical algorithm is described at a high level, independently of the hardware architecture, as a Directed Acyclic Graph (DAG) of tasks in which a vertex represents a task and an oriented edge represents a dependency between tasks. The DAG describes explicit dependency between tasks or a data dependency, that tasks have to be performed in some order.
A second layer is in charge of taking the scheduling decisions. Based on these decisions, a runtime system will perform the actual execution of the tasks, maintaining data consistency and ensuring that dependencies are satisfied. The fourth layer consists of the optimized code for the related tasks on the underlying architectures. This approach is starting to give successful results in various domains going from very regular applications [66, 3, 19, 74] to very irregular ones [54, 2, 4]. However, building such complex applications on top of task-based runtime systems requires algorithmic modifications of some core kernels of the application so that the flexibility offered by the runtime system can be fully exploited. More precisely, these operations need to be expressed as a task graph that should express enough parallelism to allow the runtime system to overcome all the synchronizations/race conditions which can be met with classical implementations of these kernels.
In this chapter we will present different approaches to parallelize a numerical simulation application in a shared memory context and using a task-based paradigm. Firstly, we will present the state-of-the-art approaches, based on graph coloring methods coupled with `OpenMP`. We will then show how we adapted this method to our problem. We will then introduce runtime systems and see how we can use these middleware to solve the problem of the assembly operation. Finally, we will present some results along with a discussion to illustrate our ideas.

3.2 Related Works

Considering the increasing complexity of modern high performance computing platforms, the need for a portable layer that will insulate the algorithms and their developers from the rapid hardware changes becomes critical. Each new accelerating hardware leaves programmers to decide whether to make the application dependent of this feature, breaking compatibility



(a) Classical approach to build an application. The application targets directly the hardware.

(b) The runtime acts as a middleware, hiding the architecture specifities to the application.

FIGURE 3.1 – Comparison of the classical approach to develop applications with respect to the runtime system approach.

with other hardware, or not, and so, misses the opportunity to increase the performance of the platform. This common problem is known as the performance portability issue. Recently, this portability layer appeared under the name of task-based runtime. There are two purposes of runtime systems. First, to provide abstraction, runtime systems offer a uniform programming interface for a specific subset of hardware. For example, OpenGL and DirectX are runtime systems dedicated to hardware-accelerated graphics. 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 and hardware specificities are hidden inside the runtime system. The abstraction provided by runtime systems thus enables portability. Yet, this is not enough to provide portability of performance, as it does nothing to leverage low-level-specific features to get increased performance. That's why the second objective of runtime systems is to optimize the actual application mapping onto resources by dynamically scheduling the tasks onto resources as efficiently as possible. This resource selection process is based on :

— scheduling algorithms and heuristics to decide which processor will process which task in order to optimize a given metric (makespan, memory-usage, etc) ;
— hints provided by the application (task priorities, critical path, etc) ;
— performance counters provided by the runtime system (performance of the kernels, etc). By relying on such a layered approach, applications can benefit architecture's underlying low-level capabilities, leading to enhanced performance without breaking their portability.

Runtime systems provide higher-level software layers (as opposed to low-level development kits like CUDA [26] or OpenCL [62]) with convenient abstractions which permit to design portable algorithms without having to deal with low-level concerns. The main objective of these frameworks is to provide simultaneously support for both data management and scheduling altogether.

Runtime systems can be classified in two main groups, namely, generic or domain specific, depending on their area of use. In the remainder of this section, we will focus only on runtime systems with high level interfaces and implementing mainly a task-based paradigm.

A lot of 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 whether or not they manage data movements between computational resources and to which extent they focus on task scheduling.

Qilin [57] provides an interface to submit kernels that operate on arrays which are automatically dispatched across the different processing units of an heterogeneous machine. Moreover, Qilin dynamically compiles parallel code for both CPUs (by relying on the Intel TBB [68] technology) and for GPUs, using CUDA. Another relevant framework is Charm++ [43] 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 [50]. Other runtime like UPC [17] rely on the PGAS programming model to express the parallelism.

Many runtime systems propose a task-based programming paradigm. Runtime systems like `StarPU` [8], KAAPI/XKAAPI [39], APC+ [37], or Legion [73] offer support for hybrid platforms comprising CPUs and accelerators. 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 is based on dynamic strategies like work-stealing, *e.g.* for KAAPI. 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 [10, 16, 9]. 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. Finally, the `PaRSEC` [18] runtime system dynamically schedules tasks within a node using a rather simple strategy based on work-stealing. 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 different types of tasks, only a great number of instance of these tasks) to represent the task dependency graph in an algebraic fashion. A more detailed presentation of the `StarPU` and the `PaRSEC` runtime systems will be given in the following sections.

In parallel to the efforts mentioned in the previous section, many domain specific runtime systems (mainly targeting linear algebra) have been developed in the past years. The TBLAS runtime system [70] follows a linear algebra specific approach. It automates data transfers and provides a simple interface to create dense linear algebra applications. 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 [51] 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 to other dedicated runtime systems. Finally, the SuperMatrix runtime system [25] follows nearly the same idea, as it represents the matrix hierarchically : the matrix is viewed as blocks that serve as data units, while operations on those blocks are treated as units of computation. The implementation transparently enqueues the required operations, internally tracking dependencies, and then executes the operations utilizing out-of-order execution techniques.

The main differences between all the approaches concern whether or not they manage data movements between computational resources, to which extent they focus on task scheduling, and how task dependencies are expressed. These task-based runtime systems aim at performing the actual execution of the tasks, both ensuring that DAG dependencies are satisfied at execution time and maintaining data consistency. Most of them are designed to allow writing a

program independently of the architecture and thus require a strict separation of the different software layers : high-level algorithm, scheduling, runtime system, and actual task implementation. Among these frameworks, we will focus on the `StarPU` and the `PaRSEC` runtime systems.

Regarding the use of task-based runtime systems for the design of high-performance applications, a lot effort has been made in the past five years. The dense linear algebra community has strongly adopted such a modular approach [66, 3, 19] and delivered subsequent production-grade solvers. As a result, performance portability is achieved thanks to the hardware abstraction layer introduced by runtime systems. More recently, this approach was considered in more complex/irregular applications : sparse direct solvers [54, 2], fast multipole methods [4], etc. The obtained results are promising and illustrate the interest of such a layered approach.

From the numerical simulation point of view, and more precisely regarding finite element methods, a lot of effort has been made recently to exploit modern heterogeneous architectures (i.e. multi-core systems equipped with accelerators) [41, 34]. The main idea is to be able to have efficient implementations of the core kernels needed by the numerical simulation, namely, assembly operation, and linear systems solution, on these architectures. We believe that this effort is necessary to understand the bottlenecks so as to achieve a good performance on such heterogeneous architectures. However, we believe, as stated above, that the modular approach proposed in this chapter, coupled with a fine grain task-based expression of the application, will ensure performance portability.

3.3 The `StarPU` runtime system

As most modern task-based runtime system, `StarPU` aims at performing the actual execution of the tasks, both ensuring that the DAG dependencies are satisfied at execution time and maintaining data consistency. The particularity of `StarPU` is that it was initially designed to write a program independently of the architecture and thus requires a strict separation between the different software layers : high-level algorithm, scheduling, runtime system, and actual code of the tasks. We refer to Augonnet *et al.* [8] for the details and present here a simple example containing only the features relevant to this work. Assume we aim at executing the sequence $fun_1(\underline{x}, y)$; $fun_2(\ x)$; $fun_1(\underline{z}, w)$, where $fun_{i, i\in\{1,2\}}$ are functions applied to w, x, y, z data ; the arguments corresponding to data which are modified by a function are underlined.

A task is defined as an instance of a function on a specific set of data. *Codelets* are abstractions of tasks that can be executed either on a core or on an accelerator. Developers provide implementation of the codelets for each architecture that can execute them, using their specific programming languages. *Codelets* embed a high level description of the data manipulated by the function, and their access mode.The set of data and tasks are asynchronously declared using a **submit_task** instruction. This is a non blocking call that allows one to add a task to the current DAG and postpone its actual execution up to the moment when its dependencies are satisfied. Although the API of a runtime system can be virtually reduced to this single instruction, it may be convenient in certain cases to explicitly define extra dependencies. To do so, identification tags can be attached to the tasks at submission time and dependencies are declared between the related tags with the **declare_dependency** instruction. The resolution of dependencies using tags are faster than the classical data dependencies.

Several preexisting task scheduling policies can be applied to any high-level algorithm. Moreover, the API offers facilities to implement new scheduling strategies. One can create as many queues as desired, one for the the GPUs and one for the CPUs, for example, and execute the functions triggered by the runtime layer every time a new task has its dependencies satisfied.

3.4 PaRSEC

As described in [20], PaRSEC is a dataflow programming environment supported by a dynamic runtime. It allows the programmer to handle easily the challenges of the ongoing changes at the hardware level. The underlying runtime is a generic framework for architecture-aware scheduling and management of micro-tasks on distributed many-core heterogeneous architectures. The dynamic runtime is only one side of the necessary abstraction, as it must be able to discover concurrency in the application so as to feed all the computing units. To reach the desired level of flexibility, it supports the runtime with a symbolic representation of the algorithm, able to expose more of the available parallelism than traditional programming paradigms. The runtime is capable of exploiting this parallelism to increase the opportunities for useful computation, predict future algorithm behaviors and increase the usage of the computing units.

Algorithm 2 Ping-Pong algorithm expressed in the PaRSEC dataflow description

```
1: PING(k) : k = 0 .. N
2: RW A ← (k == 0) ? A(k) : A PONG(k-1)
3:          → A PONG(k)

4: PONG(k) : k = 0 .. N
5: RW A ← A PING(k)
6:          → (k == N) ? A(k) : A PING(k+1)
```

Algorithm 2 represents a concise dataflow description of a ping-pong application, where some data A(k) is altered by two tasks, PING and PONG, before being written back into the original location A(k). Line 1 defines the task PING(k) and its valid execution space, $\forall k \in [0..N]$. Line 2 depicts the input value A for the task PING(k) : if k is 0, the data is read from an array A(), otherwise it is the output A of a previous task PONG(k-1). Line 3 describes the output flow of the task PING, where the locally modified data A is transmitted to a task PONG(k). This task PONG(k) can be executed in the context of the same process as PING(k), or remotely. The runtime will automatically infer the communications depending on the locations of the source and target task. Lines 4 to 6 similarly depict the complementary task PONG.

Each task includes, in addition to the dataflow definition depicted in the above algorithm, several possible implementations of the code to be executed on the data, the so-called codelets. Each codelet is targeted toward a specific hardware device (CPU, Xeon Phi, GPU) or a specific language or framework (Open CL). The decision of which of the possible codelets is to be executed is controlled by a dynamic scheduling that is aware of the state of all local computing resources. Once the scheduling decision taken, the runtime provides the input data located on the specific resource where the task is to be executed and, upon completion, it will make the resulting data available for any potential successors. As the task flow definition includes a description of the type of use made by a task for each data (read, write or read/write) the runtime can minimize the data movements while respecting the correct data versioning. Not depicted in this short description are other types of collective communication patterns that can be either described, or automatically inferred from the dataflow description.

Overall, this symbolic description of the algorithms allows for a separation of concerns between the different parties involved : the user, the compiler and the programming environment. The user focuses on a high level description of the algorithm in terms of data-flows, the compiler focuses on optimizing the codelets, while the runtime focuses on maximizing the occupancy of computing resources (scheduling), improving data reuse and minimizing data movements. Moreover, a significant difference between this approach and what other task-based runtimes typically do is to be mentioned. In other task-based runtimes, the execution flow is directly

derived from the sequential execution of the target application ; discovering the task-graph in a scalable way in such cases is a challenge in distributed environments. In contrast, `PaRSEC` 's parametrized task graph provides a concise symbolic task representation that allows for a scalable task discovery and scheduling in distributed, heterogeneous environments.

3.5 Finite Element Method

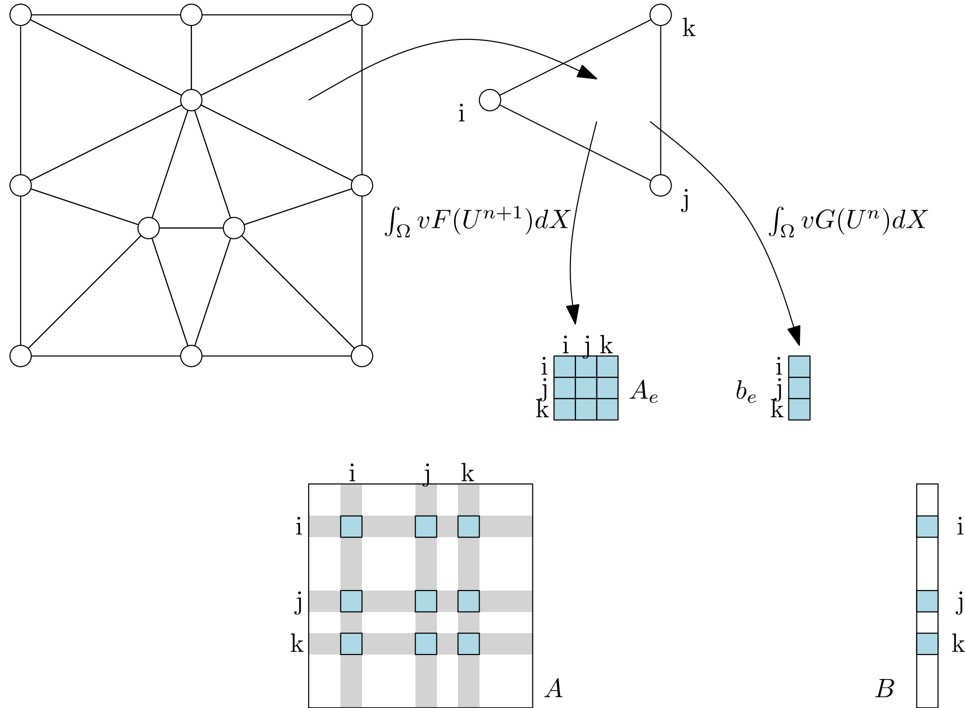


FIGURE 3.2 – A basic representation of a Finite Element Method. The method visits each element, builds contributions and assembles them in the global linear system.

As shown in the previous chapter, to follow the evolution of a physical phenomenon, finite element-based are often used . As mention in algorithm 1 (section 2.1) At each time step, we assemble a linear system, solve it, and update the solution, then iterate over the next time step. Focusing on the mathematical aspect is not our concern in this chapter. Rather, our aim is to address the algorithmic challenges related to an efficient implementatio of these methods. We will study two families of finite element methods : those that operate in the continuous domain, referred as (CG, Continuous Galerkin), and those that operate in the discontinuous domain (DG, Dicontinuous Galerkin). These methods apply to a mesh of elements. The mesh will be be visited during algorithm 1 in various ways :

- loop over all the elements. For each element visited, we compute at least one contribution block for the global matrix, and at least one contribution for the global right hand side ;
- loop over the border faces. The boundary conditions will be applied to the matrix and to the right hand side ;
- loop over all the faces in the mesh. In the DG case, a numerical flux needs be computed between elements sharing a face. This phase can update the matrix and/or the right hand side, depending wether the method is explicit or implicit.

Each time we visit an entity like an element, we gather information coming from the entities (its faces, edges, and points) around this element, and build a contribution, which will be

scattered in the linear system. We distinguish two stages : first, the embarrassingly parallel computation of each contribution block, or vector ; then the assembly of these contributions within the global linear system. Figure 3.2 shows a small example of a method visiting the elements of the mesh. For each element, two contributions are computed : one goes into the global sparse matrix, and the second one into the right hand size. Both use an indirection array that gives the correspondence between the local number of the equation in local memory and the global numbering in the whole system. The way each contribution is computed depends on the method and on the physics. It can be as easy as a vector product to generate a matrix, but it can be as difficult as an inversion of a function using an iterative method.

3.5.1 Computation of contributions

In this phase of computation, each entity, element or face, can be processed in parallel. The entity will gather local information in the case of the element, the face will gather informations coming from the two elements sharing it. Using some reference information such as the shape functions, the quadrature formula, and the geometry of the local entities, the method can compute the contribution. Our first strategy is to pack elements, that we call a `macro element`. All elements of the `macro element` are of the same kind, so the finite element is fixed. Knowing this, when treating a `macro element`, the shape functions, the quadrature formula, the geometric functions are the same for all the elements. Each operation on an elements is rewritten to work on a set of element instead of a single element.
The ideal number of element in a `macro element` is determined dynamically, based on the type of the element, the order of the solution and some parameters coming from the hardware, like memory size of the GPU. To avoid costly indirections on GPU some information is duplicated, such as the geometrical coordinates, and the list of degrees of freedom. This information and that linked to the shape functions, allows us to compute the values on our elements. The laplacian DAG is indicated in figure 3.3. In this figure, the DAG depicted is applied to an element E_i. First, two tasks compute the contribution : the $\nabla U \nabla V$ and the Mass Integrator. Both these tasks uses a temporary buffer, called B, in which the contribution is stored after computation and a set of reference data related to the element, like the shape functions, the quadrature formulae and, its geometry. Finally, an assembly task add the contribution in the whole linear system using the

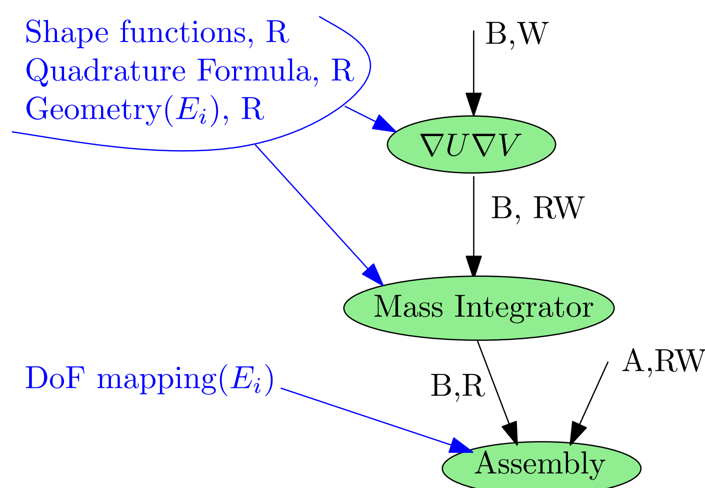


FIGURE 3.3 – Full directed acyclic graph of the computation of the contribution and assembly for the laplacian problem.

3.5.2 Taskified Finite Element Method using `StarPU`

To submit tasks to `StarPU`, we describe here what the developer should define. The first step is to re-express the target application as directed acyclic graph of tasks taking input data and producing output data. Let us take for example, the computation of the laplacian,

$$-\nabla U + U = f \tag{3.1}$$

We have explained in Chapter 1 that this operation involves two integrators to compute the contribution coming for each element : The mass integrator and the gradient ($\nabla U \nabla V$).

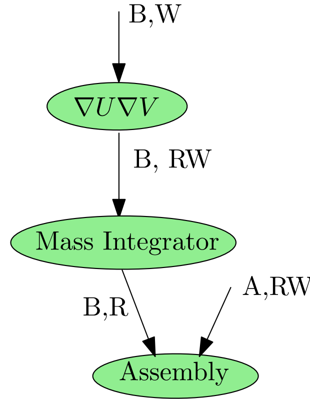


Figure 3.4 – Directed acyclic graph of the assembly for the laplacian problem.

In figure 3.4, we provide a DAG representation of the operations that need to be applied to each element of the mesh. In green ellipses, one can see the operations, or tasks, and between them, the data dependencies associated with an access mode. As said previously, by providing the access mode for each data, `StarPU` will infer the dependencies either at runtime or at the submission phase. Please note that the order of the operations has no importance, each contribution will be summed in the global matrix, and the sum being commutative, one can do either solutions. One can merge the two operations ($\nabla U \nabla V$ and the Mass Integrator) in one, but we wanted to built a set of elementary blocks, and have avoided complex operations. We have indicated in figure 3.4 the buffers B in which we will accumulate the contribution, and A, the matrix in which we assemble the total contribution of the element. The complete DAG for this operation is given in figure 3.5.
Each "instance" of this DAG will work on a certain element. Some informations are independent from the elements and more related to the kind of element, like the shape functions and the number of degrees of freedom. The informations linked to the quadrature formula are defined by the finite element. Geom(E_i) contains the coordinates of the real element in the global space. As explained in the previous chapter, our methods work on a reference element, so we have to keep the real coordinates during the computations to be able to project the contributions in the global space. Finally, the degrees of freedom mapping contains the indirections arrays needed to assemble each local entry of a contribution in the global matrix. Once the DAG of the method has been defined, The programmer needs to describe the data to `StarPU`. This is done by registering all the data. Secondly, one will describe each type of task as a *c*odelet. This structure embeds the different implementations of our task. One can provide one implementation for each generation of GPU, each with its own optimizations. We will provide the number of registered data this function will manipulate, and then the access mode for each of this data. This is the information `StarPU` will use to infer the dependencies between the submitted tasks.

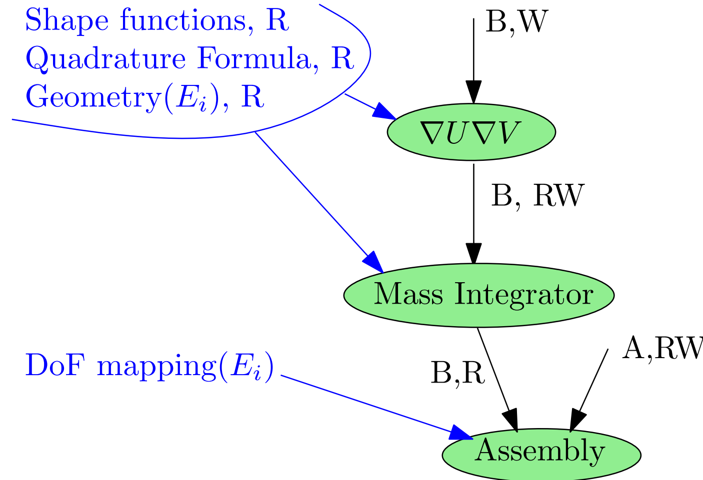


FIGURE 3.5 – Full directed acyclic graph of the resolution of the laplacian problem.

3.5.3 Taskified Finite Element Method using PaRSEC

The PaRSEC runtime is different from StarPU. Among the differences, we can talk about the visible ones from the programmer's perspective. First of all, in StarPU, the developer will use an insert_task method to submit tasks, dependencies will be inferred from the order tasks are submitted : StarPU relies on the Sequential Task Flow model for the description of the task graph. In PaRSEC, when the developer uses the JDF (an intermediate language to describe the task graph) file method data dependencies will be explicitly described and no tasks will be explicitly submitted. The runtime will build on its own the list of executable tasks based on the description of the algorithm provided in the JDF : PaRSEC uses the Parametrized Task Graph paradigm for describing the DAG. Secondly, StarPU will use its full knowledge of the directed acyclic graph to schedule the tasks using cost models to decide the where and when to execute a ready task. In PaRSEC, the runtime has only a local view of the DAG ; when completing a task, it will discover the released tasks and add them to the executable tasks set. We used priorities in our methods, it works well with StarPU, but the main scheduling criterion of PaRSEC is not is the locality of data, the defined priorities are only used as hints. We will give a description on how PaRSEC was used once the taskified assembly operation is introduced.
In the following section, we will present in more details the assembly operation of the contribution blocks and present the taskified approach we propose.

3.6 The assembly operation

From a general point of view, assembly operations can be viewed as scatter-add operations of each contribution on the global matrix following the scheme depicted in Algorithm3. Each contribution block after being built is summed with the corresponding entry of the matrix A. The association between entries of the contribution blocks and the entries of the global matrix A are determined during an initialization phase where two indirection arrays *rmap* and *cmap* store the correspondence between local indices of the contribution block and global indices of the matrix A.
In Figure 3.6, let us consider this small mesh composed of three quadrangles, for each quadrangular one will build a contribution block of size 4x4. The global matrix A will be 8x8 : there are 8 equations in this system. If we look at the quadrangular E_2 (vertices 3, 4, 5, 6), $rmap(c_2, 1)$ (resp. $cmap(c_2, 1)$) will be equal to 3 while $rmap(c_2, 2)$ (resp. $cmap(c_2, 2)$) will be equal to 4.

1: Initialize matrix A.
2: **for each** contribution C_i **do**
3:     **for each** entry $c[i][j]$ of C_i **do**
4:         $A[rmap[C_i, i]][cmap[C_i, j]] += c[i][j]$
5:     **end for**
6: **end for**

Algorithm 3 – Sequential assembly operation.

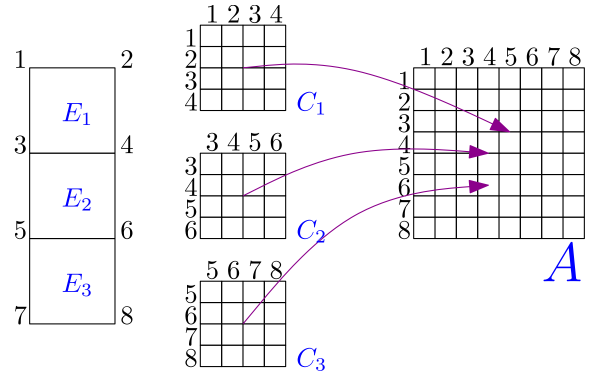


FIGURE 3.6 – Sequential assembly operation on a simple mesh.

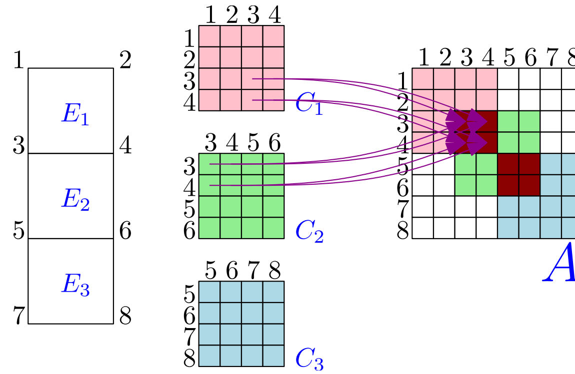


FIGURE 3.7 – Two elements overlap. Their contributions overlap too.

Recently, a lot of work has targeted the implementation of efficient assembly operations for finite element methods running on multi-core architectures which may be enhanced with accelerators. The main issue with the parallelization of assembly operations comes from the race conditions which occur when two different contribution blocks need to update the same entry of the global matrix A. In Figure 3.7, one can see that the unknown number 4 will receive two contributions (corresponding to C_1 and C_2) that must be treated sequentially.
A first approach, a naive one, is to process the contribution blocks in a sequential way using a parallel implementation of the assembly of one block. In the figure 3.8, one can see for each block a 1D distribution of work. Each color represents a thread, each thread working on a row of the contribution block. One will use a fork-join model to exploit the parallelism inside the block. This strategy requires the contribution block to be large enough to ensure performance. Moreover, this approach suffers from the lack of scalability.
More recently, in [24] Cecka *et al* adapted a parallelization approach based on a coloring of the contribution blocks to multi-core architectures : each block of the same color can be treated in parallel. This property is guaranteed, at construction, by the fact that each element has a different color than its neighbours. If two elements share at least one equation, those two elements will receive two different colors. The worst case we have to consider is when elements share equations on the vertices. Basically, it means that an equations is owned by one to n elements. In Figure 3.9 we can see six triangles sharing an equation. Each element will contribute

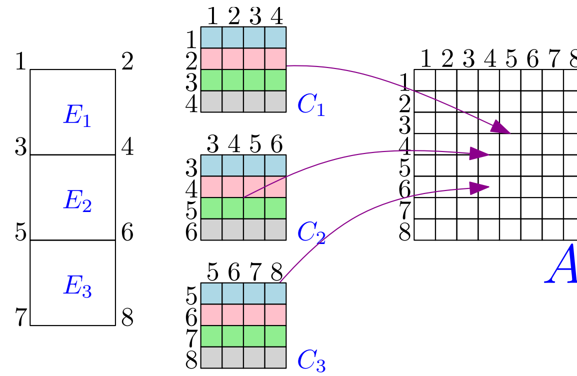


FIGURE 3.8 – Each contribution block is divided in rows. And each rows is treated in parallel.

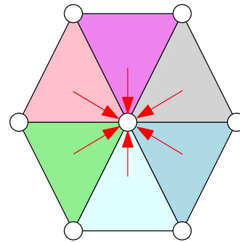


FIGURE 3.9 – 6 triangles share an equation. Each element receives a different color to be treated in a sequential way.

to this equation, so we have to treat them sequentially in order to avoid the race conditions. Each element shares a sub-set of equations with his neighbours. So, to avoid race conditions, we just give to the element a different color than its neighbours. We then put every element of the same color in a set. In this set, we know that we can treat all the elements in parallel. The coloration algorithm we implemented is as follow :
In algorithm 4, we give a detailed description of the coloration algorithm. From line 1 to 3, we initialize each element with a NULL color. Line 4 initializes the color counter. On line 5, we keep the information of the number of element colored since the beginning, so we know the number of elements remaining. Between line 6 and 8, we pick a random number for each element and store it in the element. Then comes the election phase. We traverse the mesh, and for each non colored element, we compare its random value to the one of its neighbours. If the neighbour has neither a color, nor being colored during this round, we compare the value. If the current element has a superior value, we keep tracking this information in a boolean. When we finished visiting the neighbours, we can decide if the element is a local maximum and then paint it int the current color C. Finally, during the final phase, we search for the elements forgotten during the process. Between line 24 and 36, the idea is to color elements whose neighbours have not been colored this round although they are not local maximum.
This approach allows to do the two phases in one time. While iterating over the elements in parallel, using `OpenMP` for example, one can compute the contributions and assemble them immediately. This will keep the memory consumption at a low level.

```
For c = 0 .. Ncolor-1
  For i = 0 .. Nelement[c] // OpenMP loop
    ComputeBlock(i, C_i)
    ComputeRHS(i, B_i)
    AssembleBlock(A, i, C_i)
    AssembleRHS(B, i, B_i)
```

Algorithm 4 Coloration algorithm

```
 1: for each element E of the mesh M do
 2:   E.color ← -1
 3: end for
 4: C ← 0
 5: while there are elements to color in M do
 6:   for each element E of the mesh M do
 7:     E.random ← random number from 0 to N
 8:   end for
      {first step to elect local maximum}
 9:   for each element E of the mesh M do
10:     if E.color = -1 then
11:       boolean local_max ← true
12:       for each element F, neighbour of element E do
13:         if F.color = -1 OR F.color = C then
14:           if if F.random > E.random then
15:             local_max ← false
16:           end if
17:         end if
18:       end for
19:       if local_max then
20:         E.color ← C
21:       end if
22:     end if
23:   end for
      {second step to color some forgotten vertices}
24:   for each element E of the mesh M do
25:     if E.color = -1 then
26:       boolean E_to_color = true
27:       for each element F, neighbour of element E do
28:         if F.color = C then
29:           E_to_color ← false
30:         end if
31:       end for
32:       if E_to_color then
33:         E.color = C
34:       end if
35:     end if
36:   end for
37:   C = C + 1
38: end while
```

One can make two observation about the coloring approach. First, deciding of the coloration is complicated. Droux in [30] proposed an algorithm to build an optimal coloration, but this algorithm does not ensure the existence of this coloration. Droux understood that reducing the number of colors must not be the main objective. A small number of colors only means a larger number of parallel tasks in a color. But if you have a limited number of computational resources, you do not need more parallelism (the number of parallel tasks) than the number of computational resources. That's why he focused on building sets containing exactly the number of threads you can execute simultaneously.
When coming to coloration, people barely talks about their method to color the elements. We can assume that an approach where elements receive a random number, and the local maxima are elected in the same color is a decent solution. It will not produce an optimal coloration, the latest color lists can be under-populated, leading even to an under-utilization of the computational resources.
This idea has been pushed further by Markall *et al* in [58] by improving the coloring scheme in a way such that the number of colors used is reduced. Cecka was coloring the elements, Markall is coloring the equations. It means that he divides the contribution block in rows. We know that a row will overlap with a row of the matrix, and that elements overlap each other on rows. By going at a finer grain, he is able to exhibit more parallelism with smaller tasks.
By doing this, Markall proposed a way for coloring the contribution rows. Let A be the global matrix, let A_E be the block diagonal matrix of all the contribution blocks. This matrix has a size of $(\sum_{e \in Elements} n_{dofs}(e))^2$, and let M be a transformation matrix working as an indirection array between the local index of the contribution to the global matrix. $m_{i,j} = 1$ if the i-th row of A_E contributes to the j-th equation of the system, else $m_{i,j} = 0$. Let M^{T} denotes the transposed of matrix M. The formula for building the matrix A becomes :

$$A = M^{\mathrm{T}}.A_E.M$$

Markall's approach is to use matrix M to color the equation contributions. The algorithm picks the first column of M, it looks for the first non-zero term, gives it the first color, then proceed to next term, gives it the second color, and goes on until there is no more 1 term in the column. It repeats this operation for each column, starting back to the first color each time it changes the column. When the algorithm ends, every contribution received a color, and every contribution overlapping the same equation, i.e. each contribution in the same column, has a different color.

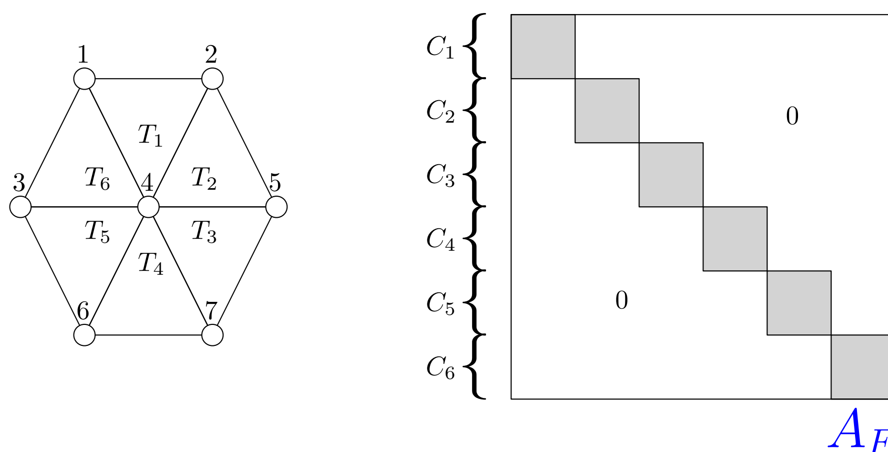


FIGURE 3.10 – On the left, a mesh with 6 triangles. On the right, the block diagonal matrix A_E, each block correspond to the contribution C_i built from the triangle T_i.

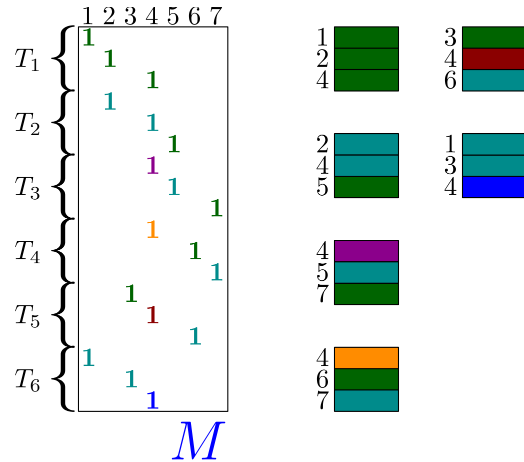


FIGURE 3.11 – The matrix M, with its coefficient colored. On the right, The contributions are divided in rows, and each row is colored according to the coloration of M.

In Figure 3.10 we have an example with six triangles. The matrix A_E, block diagonal contains the three contributions on its diagonal. And the matrix M is built as explained before (see Figure 3.11). A 1 is placed where a contribution row overlap an equation in the system. Then, the 1 are colored by exploring each column, and coloring them. When the matrix M is fully colored, we deduce the coloring of the contribution block rows, and we can treat them without worrying about the race conditions like in figure 3.12.

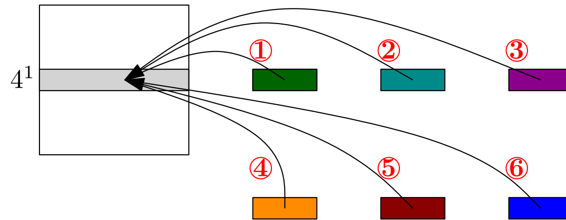


FIGURE 3.12 – Focus on the row 4, and the scheduling for summing the contributions.

This approach proposed by Markall *et al* has many advantages. With a finer grain than the Cecka approach, they can build a coloration which is optimal in the number of colors. However, this method requires the application to build all the contribution blocks before starting the assembly phase. This implies a synchronization between the two phases, and a higher memory consumption due to the necessity to keep the blocks in memory. Nonetheless this is a very good approach exhibiting a lot of parallelism, and with a finer grain, the amount of time where resources are idle, waiting for the completion of the last tasks of the current color, is less important.
Lately, Hanzlikova *et al* proposed in [36] an approach which extends the work from Cecka by using extra storage to avoid synchronizations needed to prevent race conditions.
In Figure 3.13 we focus on the fourth row of the system. As we saw in Figure 3.12, we had six steps to sum all he contributions. By introducing a temporary buffer and splitting the contributions between the two buffers, the assembly can be done in four steps, three for summing the contributions in the buffers, and one step for the sum of the two buffers. It is easy to extend this approach with a binary tree of buffers when the number of contributions is important. The advantage of this method against the Markall approach is too exhibit more parallelism at the cost of memory.

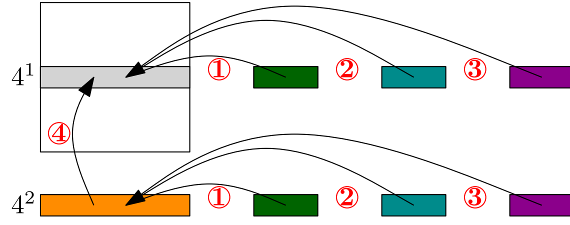


FIGURE 3.13 – Focus on the row 4, with a temporary buffer. The contributions are splitted between the original row 4 and the buffer. A last step will assemble the buffer in the row 4.

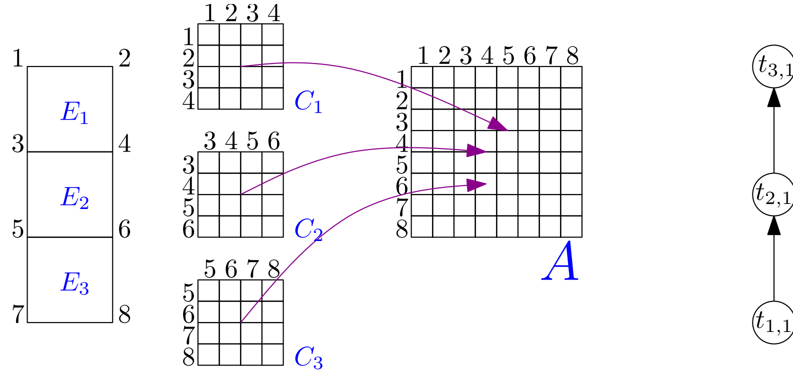
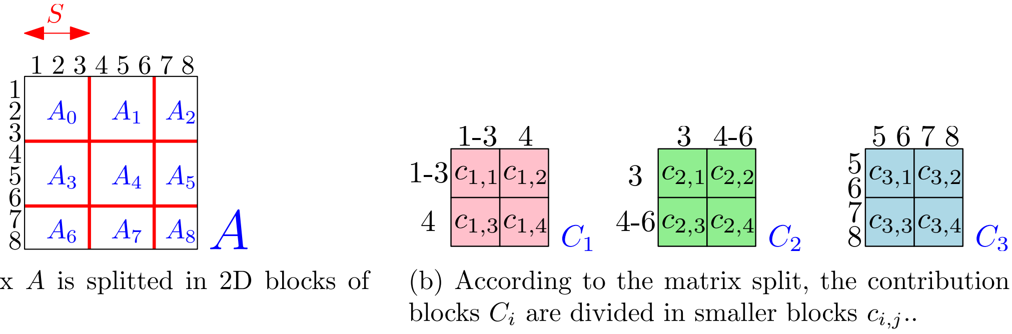


FIGURE 3.14 – Naive assembly of three contribution blocks, with the associated task graph.

In the next Section, we propose a new taskified assembly operation scheme which exhibits high level of concurrency while leaving the management of the race conditions to the underlying runtime system.

3.6.1 A new taskified Assembly Operation

We introduce in this section a taskified assembly operation where the objective is to enhance parallelism while leaving the management of race conditions which may be met to the underlying runtime system. The main phenomenon which limits the amount of parallelism is the serialization of the assembly of contribution blocks updating the same block leading to a race condition. To increase the amount of parallelism, computations must be organized such that conflicting write operations are minimized. A naive approach to express the global assembly operation would be to associate a task to the assembly operation of each contribution block C_i (see Figure 3.14). In this context, all the tasks will be serialized because of the write conflicts on the global matrix. For example, if we consider the assembly operation presented in Figure 3.14 where this naive scheme is used, the dependency task graph contains three tasks (namely $t_{1,1}$, $t_{2,1}$, and $t_{3,1}$) which have a write conflict on the global matrix. $t_{1,1}$ (resp. $t_{2,1}$, and $t_{3,1}$) has as arguments its contribution block C_1 (resp. C_2, and $t_{3,1}$) and the global matrix. Note that since the summation operator used during the assembly operation is commutative and associative, the task graph where $t_{3,1}$ is the predecessor of $t_{2,1}$ is also valid. Any ordering of those tasks is valid.
In order to exhibit more parallelism, one could partition the global matrix into blocks (see Figure 3.15a) and associate a task to the assembly operation of each contribution block into each tile of the global matrix (see Figure 3.16). Of course, if a contribution block does not update a tile of the global matrix, the corresponding empty task is not considered. By doing so, the amount of non-conflicting tasks is increased leading to higher degree of parallelism. For example, if we consider now the assembly operation described in Figure 3.16 where this tile-



(a) Matrix A is splitted in 2D blocks of size SxS.

(b) According to the matrix split, the contribution blocks C_i are divided in smaller blocks $c_{i,j}$..

FIGURE 3.15 – Splitting of the data.

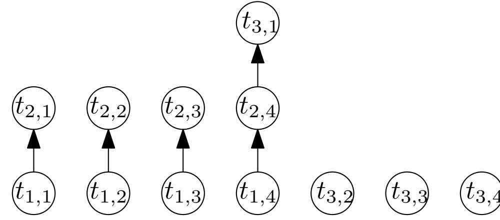


FIGURE 3.16 – Taskified version of the assembly of three contribution blocks, with the associated task graph.

based scheme is used, we can see that the task graph contains now 12 tasks for which the number of conflicts are reduced. For example, the task $t_{1,4}$, $t_{2,4}$, and $t_{3,1}$. Note that in opposition to the naive case, in this situation each task has arguments its contribution sub-block $c_{i,j}$ and a tile of the global matrix A_k. When using this scheme, the number of tasks and subsequently the degree of parallelism is strongly linked to blocking factor used for the global matrix. A tradeoff needs thus to be found between the needed parallelism and the management overhead induced in the runtime system.

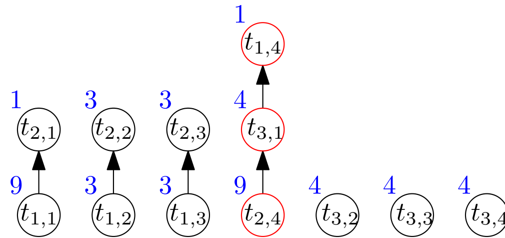


FIGURE 3.17 – Taskified version of the assembly with the associated weight.

The approach to taskify the assembly operation that we propose is a tiled approach where the serialized tasks are sorted according to their computational cost in each chain : the most costly tasks are treated first. The task graph is thus composed by a set of independent chains of tasks. This is illustrated in Figure 3.17 (the weight in blue of the task is linked to its computational cost). We can see that the task graph is composed of a set of independent sorted chains of tasks. This will allow to maximize the concurrency and enhance performance. This scheme will be referred to as the *flat assembly operation scheme.* The chain in red denotes the possibility to sort the tasks in a chain because the assembly operation is associative and commutative.
To overcome the overhead due the management of the large number of tasks, one could decrease the number of tasks for a fixed tile size by merging the chains of the flat assembly scheme into a single tasks. This will produce a fixed number of tasks corresponding to the number of

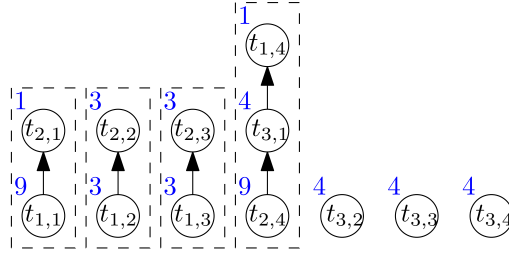


FIGURE 3.18 – Tasks in a chain are gathered in a single task.

tiles of the global matrix. This approach is similar to [24], in the sense that it builds a set of completely independent tasks preventing all race conditions from occurring. This is illustrated in Figure 3.18, where the chain is replaced by the dashed box surrounding it. This scheme will be referred to as *no-chain assembly operation scheme*. We will introduce in the next section, scheduling simple heuristics which aim to ensure a good behavior in terms of performance.

3.6.2 Scheduling strategies for taskified assembly operations

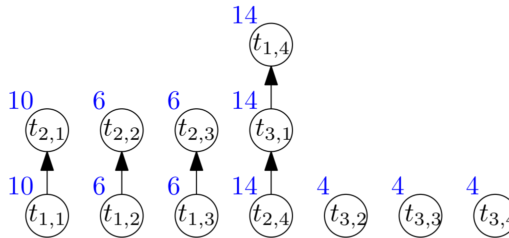


FIGURE 3.19 – Fixed priorities for each chain.

Taskified assembly operations are expressed using task dependency graphs composed of independent chains of tasks (an example is given in Figure 3.16). We consider in this work dynamic on-line scheduling strategies which are commonly used in various runtime systems. In order to efficiently assign tasks to the computational resources it is important to take into account the weight of each task in terms of workload and give priority to the largest ones (the larger the contribution the higher its priority is). This strategy is used on the set of ready tasks (i.e. tasks for which the corresponding dependencies are satisfied) and each idle processing unit picks the task with highest priority from the set of ready tasks. By doing so, the processing units are constantly working on the critical path of the execution.
Varying the tasks priorities allow for further improvement of the scheduling strategy. Even if the proposed strategy is simple, there is still room for improvement considering how the priorities are assigned to ready tasks. A first approach to express the critical aspect of a task regarding the length of the chain it belongs to, is to associate a priority related to the cost of the entire chain. This illustrates in Figure 3.19, where the priorities of the entire chain (blue labels) are constant, and are computed based on the cost of the entire chain. We will refer to this priority scheme as *fixed priority scheme*.
This priority management can be pushed further so that the priorities, not only take into account the absolute length of the critical path but its current length at the moment where the scheduling decision is taken. Thus, the priority of a task is computed based on the remaining workload on the chain it belongs to. This allows the working units to select the tasks that are currently the most critical. Figure 3.20 depicts the same example as before using this new

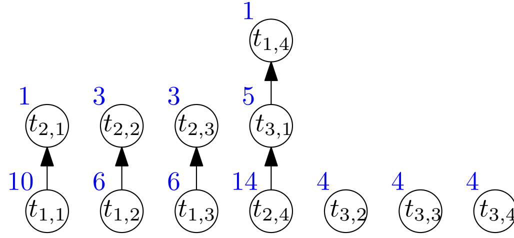


FIGURE 3.20 – Adaptive priorities for each chain.

priority assignment scheme. This time the tasks belonging to a chain have a priority linked to the length of the remaining part of the chain. We will refer to this priority scheme as *adaptive priority scheme*.
One of the major differences between `StarPU` and `PaRSEC` is the way the list of ready tasks is managed. In `StarPU`, the user divides data, precomputes a list of tasks working on those data, and submits, in advance, all the tasks. This sequential submission of tasks creates implicit dependencies between the tasks. In `PaRSEC`, the dependencies are explicitly specified by the user, and the tasks are dynamically discovered by the runtime based on completed dependencies and the symbolic description of the algorithm. From the scheduling point of view, `StarPU` gives the opportunity to the user to write his own scheduler while in `PaRSEC`, a highly optimized scheduler is provided, where priorities are secondary to enforcing a data locality policy.

3.7 Experimental results

In this section we evaluate the behavior of our taskified assembly operation approaches and illustrate how increasing concurrency leads to performance improvements. This will be done on a set of configurations arising from FEM applications on top of `PaRSEC` and `StarPU`. Moreover, the taskified assembly operation scheme will be evaluated using irregular configurations which can be met within the multifrontal method [31] for the factorization of sparse matrices.

3.7.1 Experimental setup

We evaluate the behavior of our task-based approach on the *riri* platform, composed by 4 Intel E7-4870 processors having 10 cores clocked at 2,40 GHz and having 30 MB of L3 cache. The platform has uniform memory access (UMA) to it's 1 TB of RAM.
We have chosen to illustrate the behavior of our approaches on two different classes of problems. The first class correspond to assembly operations met in finite element methods. We consider in the following study both 2D and 3D finite element continued method applied on structured meshes. The difference between the two cases resides in the connectivity between elements. While on a 2D grid, each element has at most 8 neighbors, in 3D, each hexahedron has 26 neighbors leading to higher overlapping between contribution blocks for the 3D case. The contribution blocks are dense whereas the matrix is sparse and is built using the connectivity of the mesh. The second class correspond to a less structured assembly operations met in a sparse direct method (namely the multifrontal method [31]). Here, the contribution blocks are dense and the father in which the assembly is done is dense too. The considered configuration has been generated using the `MUMPS` sparse direct solver [5, 6] using input problems coming from the University of Florida Sparse Matrix Collection [5]. To be more precise, we extracted configurations met the assembly phases needed by the sparse LU factorization. The contribution blocks for

5. `http://www.cise.ufl.edu/research/sparse/matrices`

these configurations are very irregular with sizes varying from 0.01% to 100% of the father's size. Finally, two parameters will vary in our experiments, the size of the tile and the number of computational resources. The bigger the tile size, the lesser parallelism one will be able to exhibit. And the smaller the blocks, the smaller the tasks, and we will see the consequences on the behavior of the platform. Thus, one shall find an acceptable value in sync with the second parameter, the number of computing resources units available.

Figure 3.21 depicts the performance of the assembly operation when used in the context of a finite element method application in a 2D mesh case. This corresponds to a case where the overlapping between contribution blocks is small. First of all, we can observe that, by increasing the concurrency (figures on the right, with blue plots), the taskified assembly operation obtains a very good behavior with `PaRSEC` with all strategies. Moreover, we observe that the `StarPU` implementation (left column, red plots) has a good behavior on a small number of processing units but seems less efficient when the number of resources increases. As shown in Figure 3.24 this is mainly due to the overhead induced by the management of the tasks, the tasks are not compute intensive enough to amortize the overhead of the `StarPU` runtime system (which is mainly due to the inference of task dependencies). Similarly, in Figures 3.24a and 3.24b we can notice that independently from this observation, the *no-chain assembly operation scheme* behaves well in both runtime systems mainly because there are no race conditions in this strategy, the tasks are more compute intensive and there are less tasks to schedule. We can see also, that this strategy gives performance equivalent to the one obtained with the coloring strategy described in [24] and outperforms it in certain configurations (typically when there the global matrix is tiled using fine grain blocks). This illustrates the interest of our taskified assembly scheme on this simple scenario.

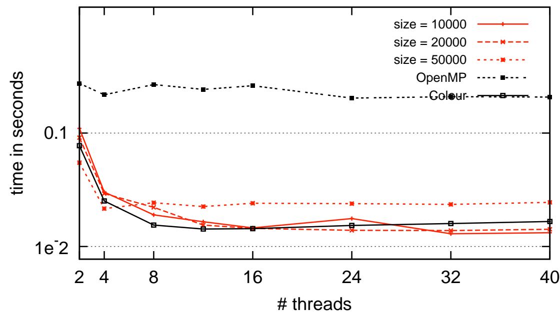
In Figure 3.22 we investigate the behavior of the taskified assembly operation on the two runtime systems in the context of a finite element method application in a 3D mesh case. This time both the size of the contribution blocks and their overlapping increased in comparison with the 2D case. We can observe that the functioning of our taskified schemes have a good behavior for all tile sizes. Moreover, we can observe that the overhead of the runtime system is negligible compared with the computational cost of the tasks and allow all the strategies to expose a scalable behavior. Concerning the coloring scheme, it is outperformed by all the strategies when the number of computational resources increased. Finally, once again, the *no-chain assembly operation scheme* is the best efficient variant for both runtime systems. Note that the performance in the rightmost plot is limited for both runtime systems because of a small parallelism degree (there are at most 10 concurrent ready tasks at a time).

Finally, Figure 3.23 reports the results gathered in the context of the most irregular and complex case : assembly operations arising in the sparse LU factorization using the multifrontal method. First of all, note that in this case, it is not possible to use the coloring heuristic since the overlapping between contributions blocks may be arbitrarily large (the cost of the coloring heuristic is prohibitive in this case). We can observe that `PaRSEC` has a good performance with all tiling strategies and all scheduling policies. We can also see that the *adaptive priority* scheduling policy is the one with the most scalable behavior. Finally, we can observe in Figure 3.26 that the overhead induced by the runtime is minimal with `PaRSEC`. Concerning `StarPU`, when the granularity of the tiles is small, we measure that the overhead of the runtime system tends to increase with the number of resources leading to a significant performance loss. However, increasing the granularity allows to overcome the runtime overhead and the behavior of `StarPU` becomes equivalent to the one obtained with `PaRSEC`. Once again, the *no-chain assembly operation scheme* is the most efficient variant for both runtime systems. Finally, we report also, the behavior of the naive implementation using based on OpenMP where all the global matrix is not tiled and the contribution blocks are treated sequentially using as many threads as provided

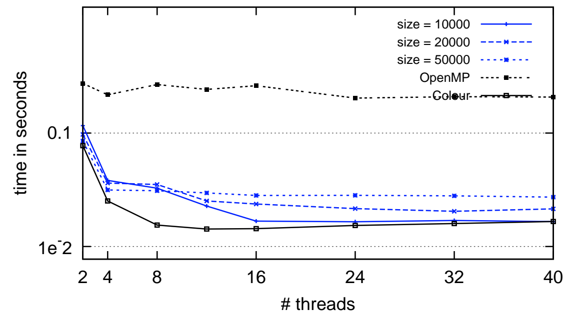
by the user for each contribution block. We can see, that our taskified assembly scheme is much more stable in terms of behavior and outperforms the OpenMP implementation for most cases. These experimental results illustrate the interest of our taskified scheme and show we do not sacrifice performance by relying on task-based runtime systems compared with a more simplistic fork-join scheme. From a broader point of view and regarding the behavior of the application, we can see that when confronted to this memory-bound operation, the overhead of the runtime system may be prohibitive with large number of cores and very small tasks (see Figures 3.24, 3.25 and 3.26). This issue needs to be tackled from both ends : The application and the runtime system. From the application point of view, a tradeoff needs to be found between the amount of parallelism and the cost of the tasks. Moreover for applications having very low complexity operations, using explicit data dependencies will be more suited. From the runtime system side (mainly on `StarPU` which infers the dependencies), the cost the data management engine needs to be further evaluated to try to reduce the global overhead.

3.8 Conclusion

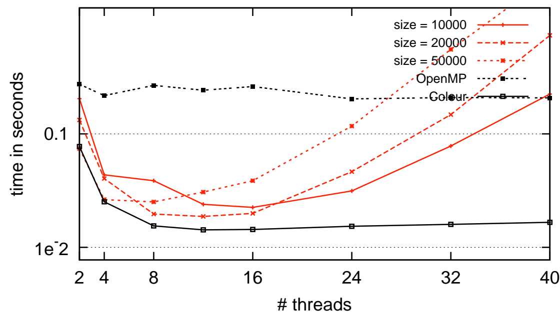
The main objective of this work was to evaluate the usability and effectiveness of general-purpose task-based runtime systems for parallelizing the assembly operation, which is a main operation in several application fields. This was done by expressing the assembly operation in taskified way and giving the resulting task graph to the underlying runtime systems. Several taskification approaches which aim at enhancing the concurrency while trying to reduce the amont of race conditions have been proposed. Overall, the results clearly indicates that for both runtime systems, namely `PaRSEC` and `StarPU`, our approaches exhibit encouraging performance, especially when the computational cost of the tasks increases. The results also illustrates that on small test cases, the overhead of the runtime systems may limit the performance.
In the near future, we plan to further extend this work by using accelerators (GPU, Intel Xeon-Phi, etc) to minimize the time-to-solution. This will be done by relying on existing assembly kernels for the different accelerators and leave the data management and scheduling decisions to the runtime systems (the scheduling policies need to be adapted to the heterogeneous context). Moreover, it could be of interest to consider intra-task parallelism which may offer more flexibility to enhance concurrency. In a longer term, this work represents a necessary kernel which will be used to design complex numerical simulation applications on top of modern runtime systems. This will allow the application to run in a more asynchronous way without relying on the classical fork-join paradigm.



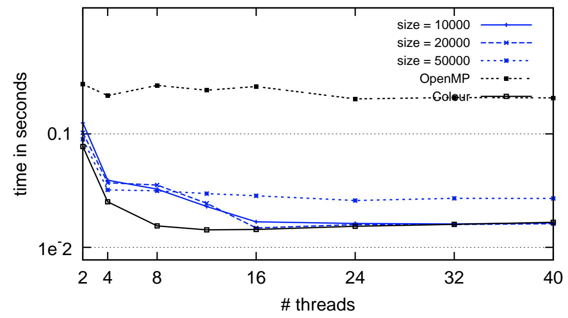
(a) No-chain strategy with StarPU.



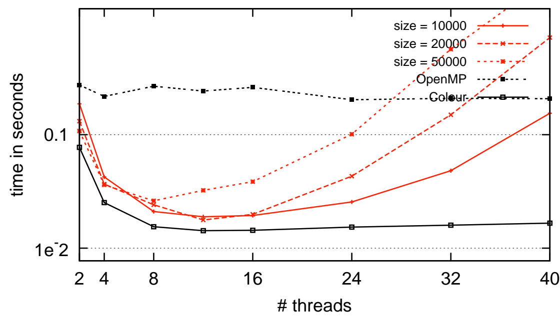
(b) No-chain strategy with PaRSEC.



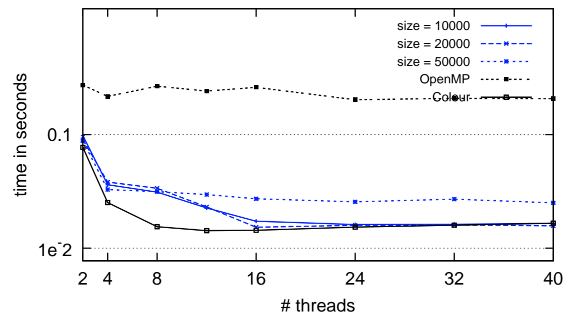
(c) Flat strategy with StarPU.



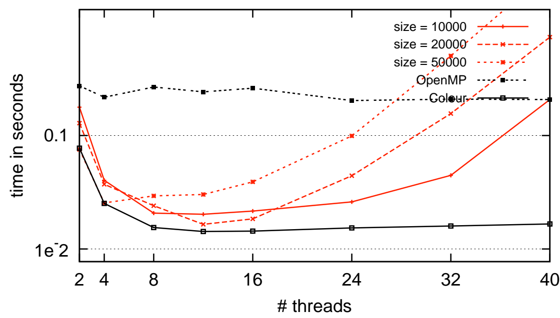
(d) Flat strategy with PaRSEC.



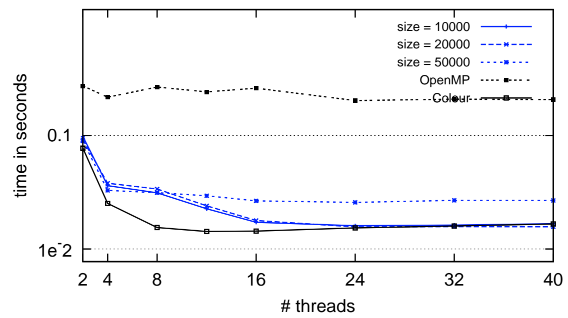
(e) Fixed strategy with StarPU.



(f) Fixed strategy with PaRSEC.

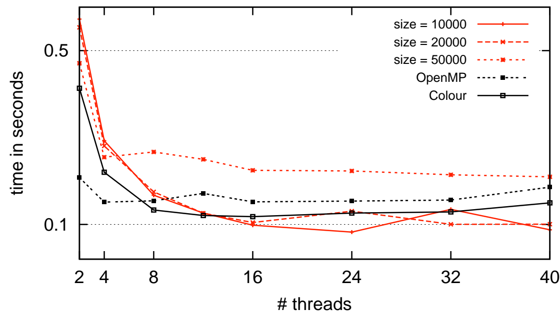


(g) Adaptive strategy with StarPU.

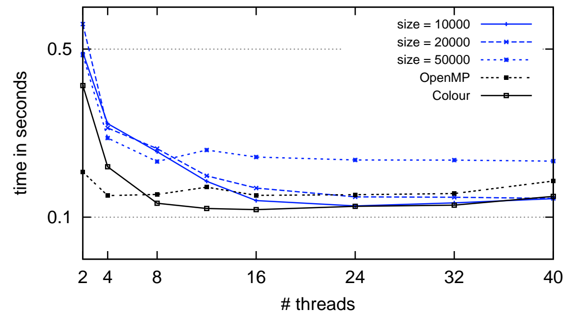


(h) Adaptive strategy with PaRSEC.

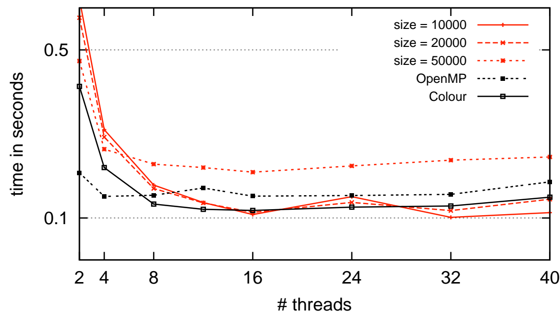
FIGURE 3.21 – Second test case, 2D grid with blocks of size 121x121, and a matrix of 203k entries. Influence of the tile size for the different strategies with the two runtimes.



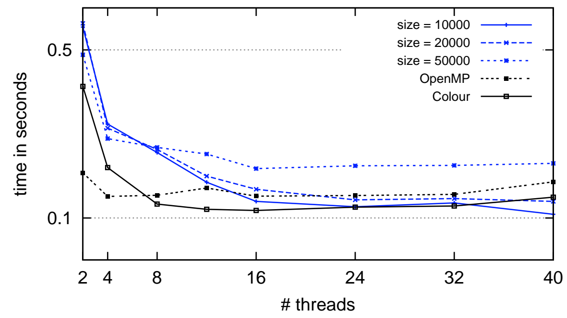
(a) No-chain strategy with `StarPU`.



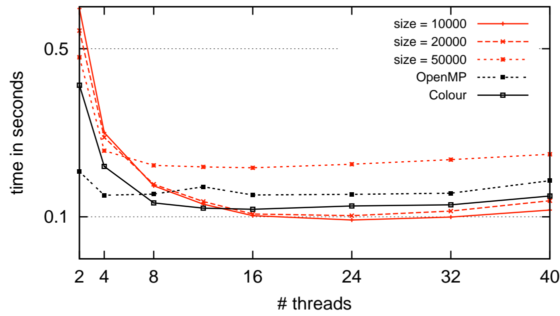
(b) No-chain strategy with `PaRSEC`.



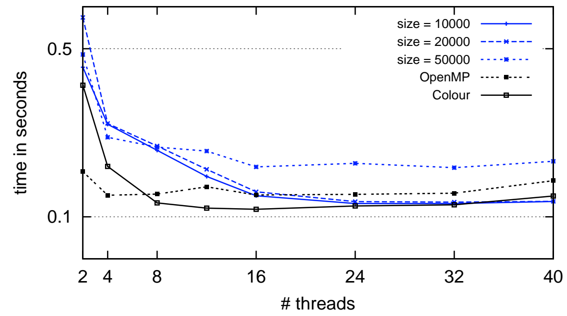
(c) Flat strategy with `StarPU`.



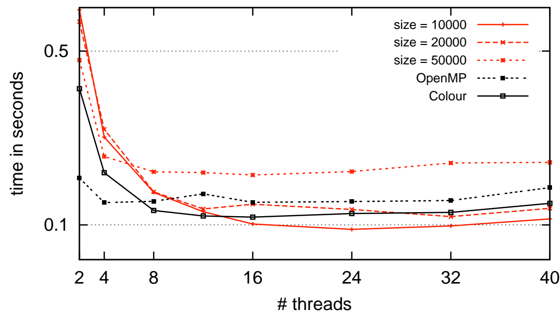
(d) Flat strategy with `PaRSEC`.



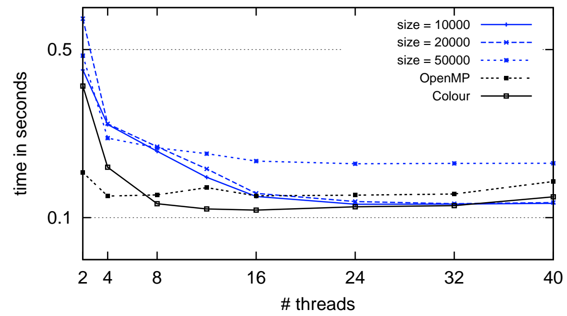
(e) Fixed strategy with `StarPU`.



(f) Fixed strategy with `PaRSEC`.

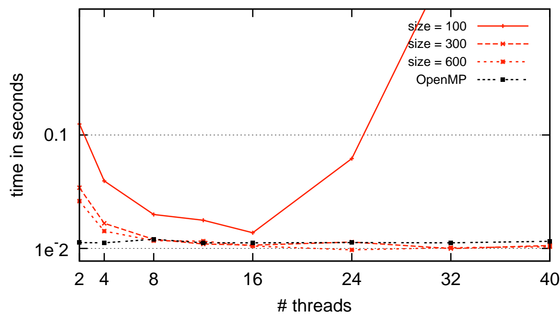


(g) Adaptive strategy with `StarPU`.

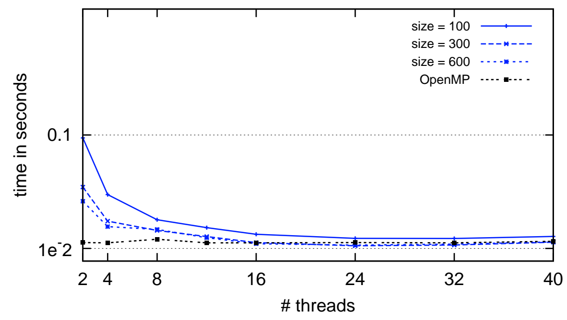


(h) Adaptive strategy with `PaRSEC`.

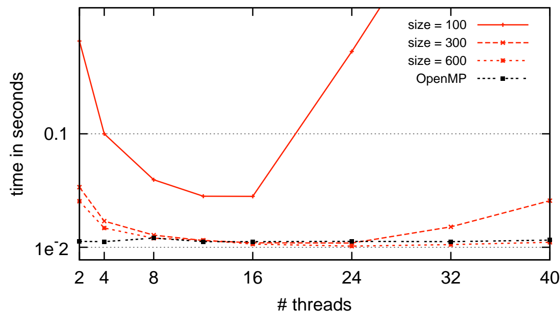
FIGURE 3.22 – Third test case, 3D grid with 512 blocks of size 512x512 and a matrix of 185k entries. Influence of the tile size for the different strategies with the two runtimes.



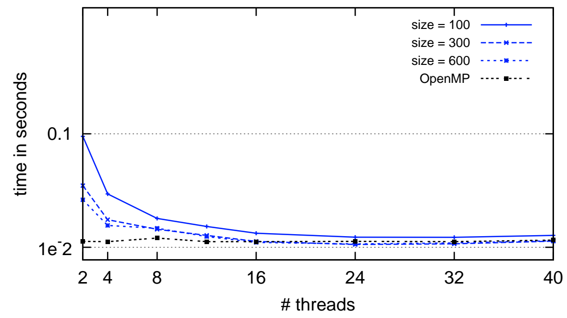
(a) No-chain strategy with StarPU.



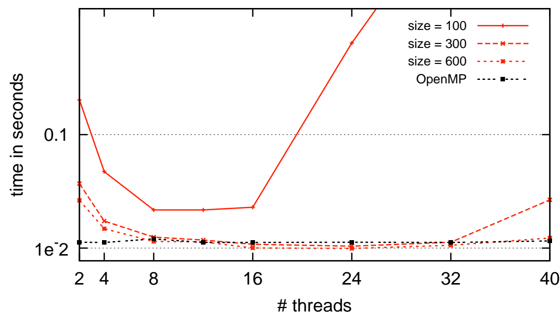
(b) No-chain strategy with PaRSEC.



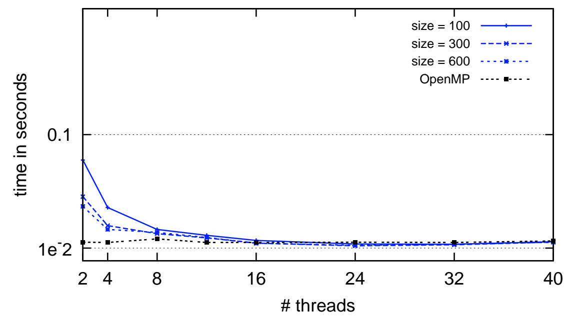
(c) Flat strategy with StarPU.



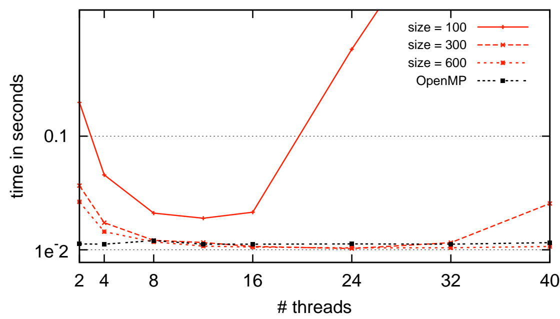
(d) Flat strategy with PaRSEC.



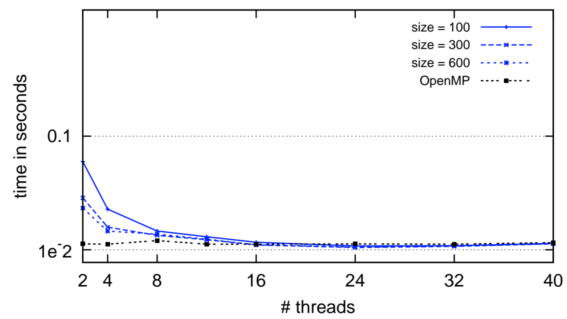
(e) Fixed strategy with StarPU.



(f) Fixed strategy with PaRSEC.

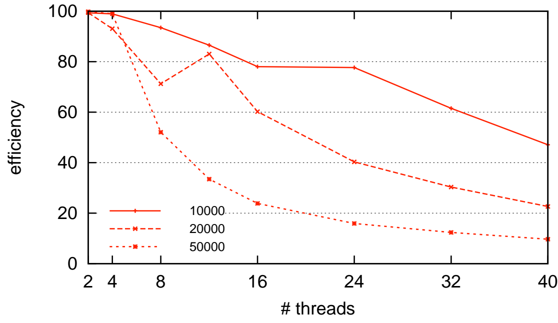


(g) Adaptive strategy with StarPU.

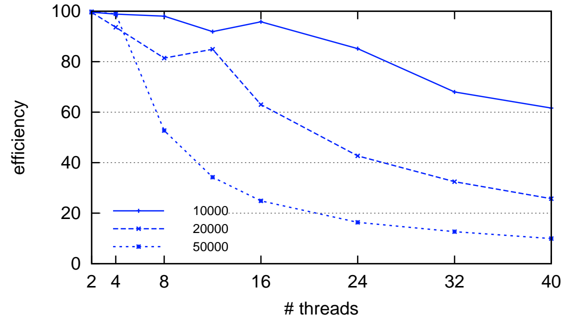


(h) Adaptive strategy with PaRSEC.

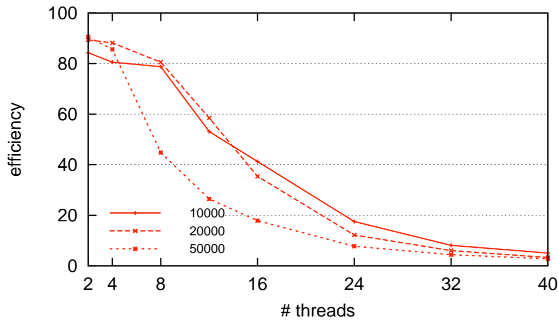
FIGURE 3.23 – First test case, MUMPS reduction. The assembly is done in a dense block (the father) of size 3061x3061. Contribution blocks sizes vary from 1 to the size of the father. Influence of the tile size for the different strategies with the two runtimes.



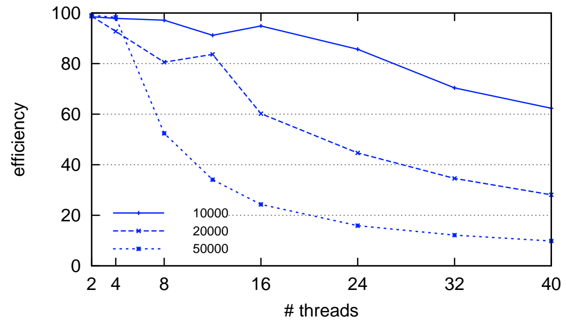
(a) No-chain strategy with `StarPU`.



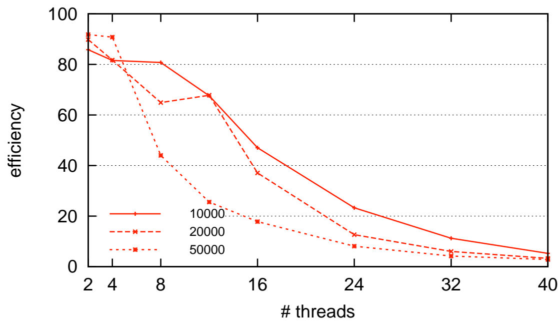
(b) No-chain strategy with `PaRSEC`.



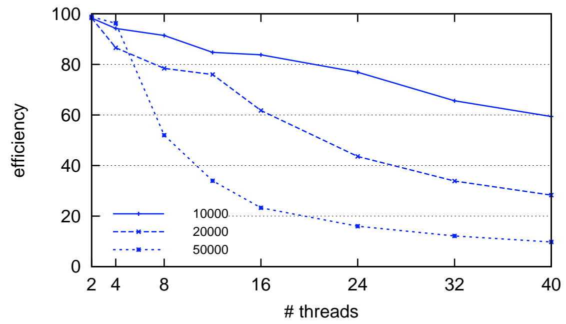
(c) Flat strategy with `StarPU`.



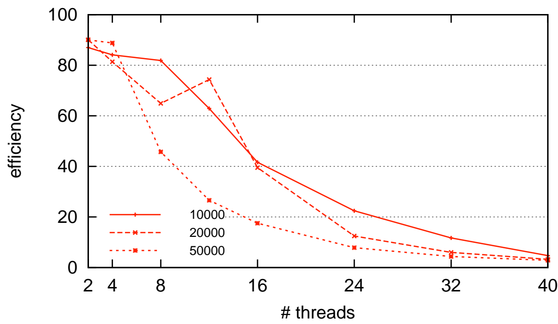
(d) Flat strategy with `PaRSEC`.



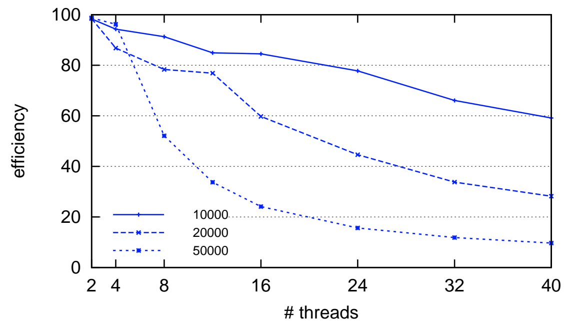
(e) Fixed strategy with `StarPU`.



(f) Fixed strategy with `PaRSEC`.

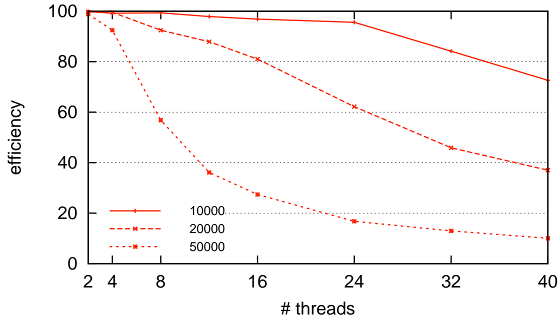


(g) Adaptive strategy with `StarPU`.

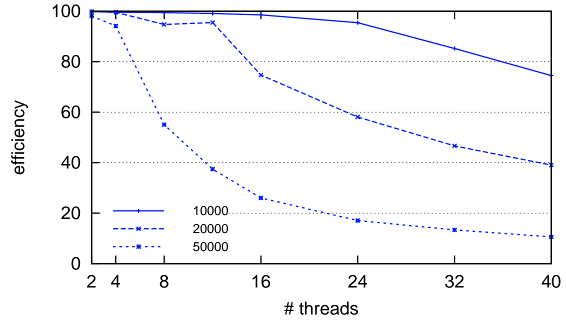


(h) Adaptive strategy with `PaRSEC`.

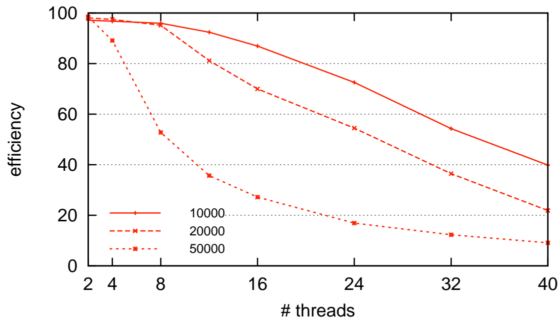
FIGURE 3.24 – Efficiency for the second test case, the 2d grid.



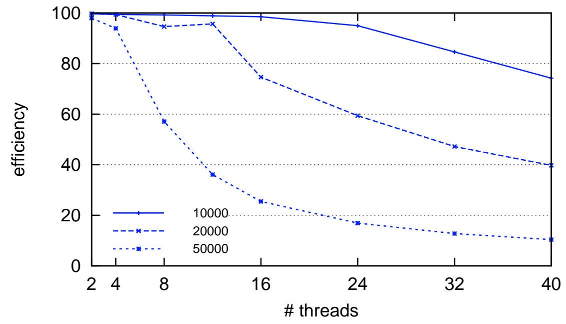
(a) No-chain strategy with `StarPU`.



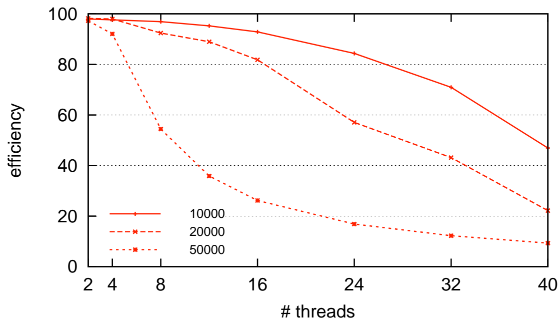
(b) No-chain strategy with `PaRSEC`.



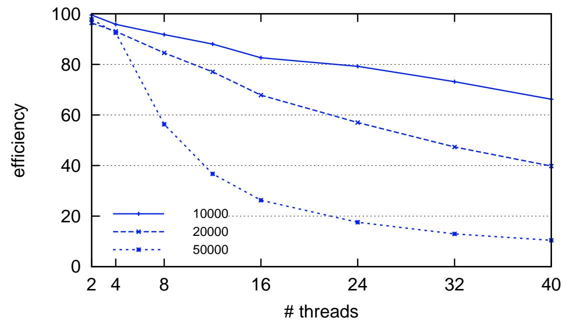
(c) Flat strategy with `StarPU`.



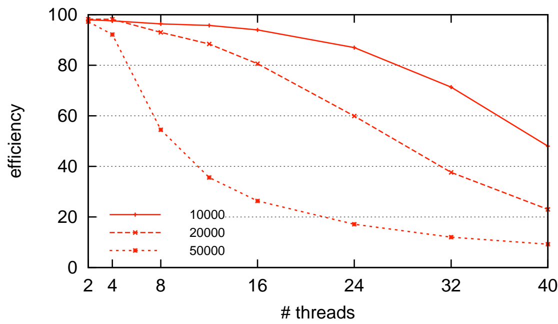
(d) Flat strategy with `PaRSEC`.



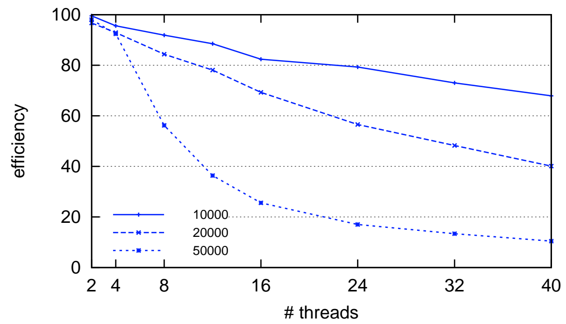
(e) Fixed strategy with `StarPU`.



(f) Fixed strategy with `PaRSEC`.

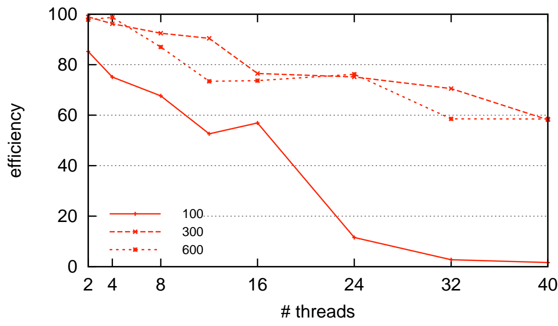


(g) Adaptive strategy with `StarPU`.

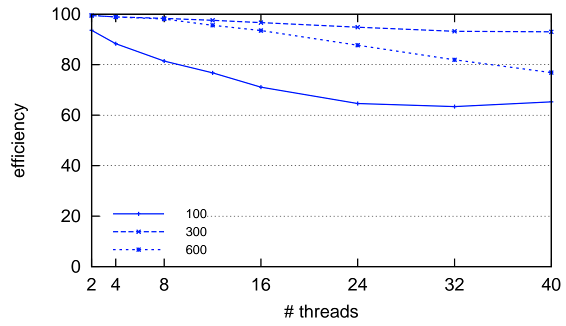


(h) Adaptive strategy with `PaRSEC`.

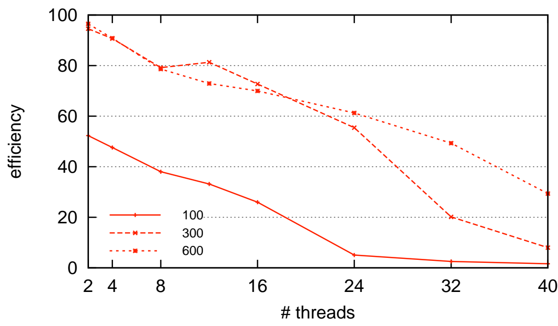
FIGURE 3.25 – Efficiency for the third test case, the 3d grid.



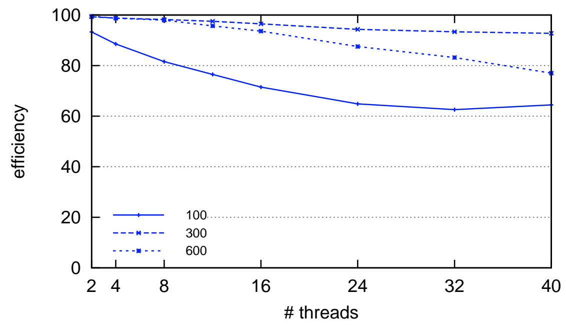
(a) No-chain strategy with StarPU.



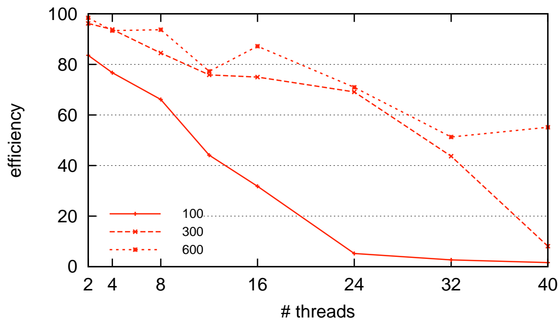
(b) No-chain strategy with PaRSEC.



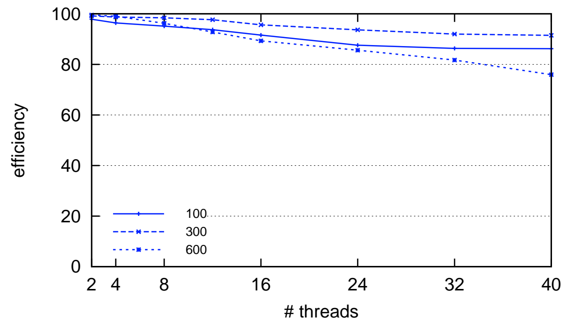
(c) Flat strategy with StarPU.



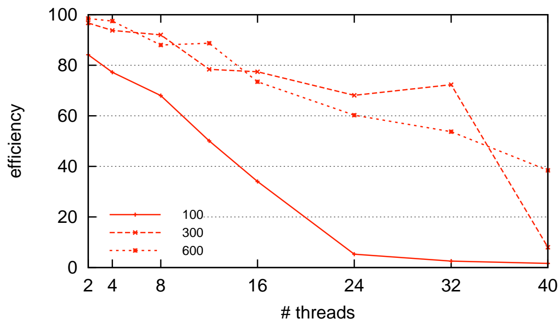
(d) Flat strategy with PaRSEC.



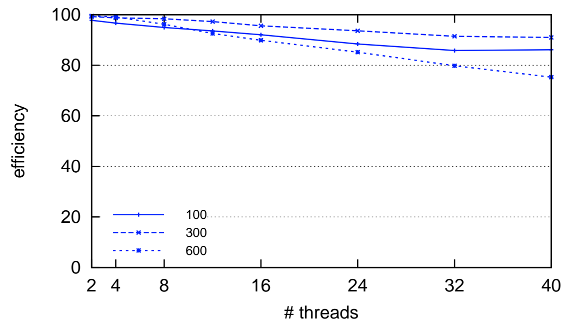
(e) Fixed strategy with StarPU.



(f) Fixed strategy with PaRSEC.



(g) Adaptive strategy with StarPU.



(h) Adaptive strategy with PaRSEC.

FIGURE 3.26 – Efficiency for the first test case, extracted from the direct linear solver MUMPS.

General Conclusion and perspectives

Conclusion

The work of this thesis was directed towards providing a new parallel platform in which contributors from diffrent backgrounds could contribute without the full knowledge of the platform. The target platform needs to be fully parallel, able to exploit the new heterogeneous architecture, with complex hierarchy of processing units and memory. Moreover, this platform has to be evolutive, and as easy as possible to maintain, by using modern tools of development and test.
In this thesis, we presented `AeroSol`, a new platform for solving PDE problems using different finite element methods. This platform can handle different kind of finite elements, either continuous or discontinuous, thus enabling us to implement two types of finite element methods, the continuous Galerkin, and the discontinuous Galerkin. `AeroSol` is able to work on hybrid meshes, composed of elements of a variable geometry, straight or curved. We proposed a hierarchical decomposition of the elements in order to abstract the memory differences between the two kinds of finite elements, and thus hide the differences within the platform.
In term of parallelism, our platform works on many levels. Between the nodes, a classical domain decomposition is computed in order to distribute the work and ensure a good load balancing between processing nodes. An overlap is computed on each domain to ensure that computations on boundary elements can be done by duplicating the cells at the frontier. Inside a computing node (at the shared memory level), many approaches has been tested to parallelize the assembly operation in shared memory. The proposed approach was to rely on modern task-based runtime systems to manage to fully exploit the available parallelism. In comparison with the classical fork-join paradigm, the task-based approach provides a fine grain management of the dependencies of the computations which allows a better pipeline of the computations inducing a better performance. A complete study has been proposed for the assembly operation which enlightens the interest of the task-based approach : The performance was comparable to the best known algorithm namely the colouring scheme. This study was done done on top of two different task-based runtime systems.

Perspectives

The concepts proposed in this thesis provide some solutions to build a robust and portable solver for PDE problems running on modern high performance computing architectures. Meanwhile, the approach proposed in this thesis needs to be extended in several directions. First of all from the modelisation point of view, it clearly will be interesting to support other numerical methods and simulation schemes. Secondly, the study proposed in this thesis provided a first insight of the performance granted by the use of modern runtime systems in such numerical simulation softwares. This work has to be pushed further by increasing the interaction of the software, the external tools and the runtime system. For example, `AeroSol` can rely on existing

sparse solvers based on modern runtime system. This will allow the global iteration to be done without explicit synchronization as soon as the structure of the sparse matrix does not change and that the solver is able to use an elemental input format instead of the assembled sparse matrix format. All on in all, this research direction is focussed around an increased interaction between a numerical simulation tool (`AeroSol`) and external tools (sparse solvers) which share the same runtime system. Another research direction which has to be considered is related to the use of modern heterogeneous architectures : modern multicore systems equipped with accelerators. We believe that a viable approach to tackle the problem is to rely on existing kernels or kernel generators for these devices and to consider only the scheduling issues related to the use of accelerators. Thus, the approach which will be considered consists in designing high-level scheduling engines adapted to the needs of the numerical simulation. In this context, another important issue is related to the granularity of computations which needs to be considered to ensure a good performance on all the types of computing resources. Finally, we believe that a third research axis will be the design of high level domain specific languages (DSL) to describe the numerical simulation scheme. This will make the project easily usable for non-expert users. Moreover, this topic opens a lot of collaborative research activities with both the compilers and runtime systems communities. From our point of view, the main challenge is to be able to generate the task-graph of the numerical simulation together with the actual high performance implementation of the tasks inpendently from the numerical scheme considered.

Bibliography

[1] FEniCS home page. http://fenicsproject.org/.

[2] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. Multifrontal qr factorization for multicore architectures over runtime systems. In *Euro-Par 2013 Parallel Processing - 19th International Conference*, pages 521–532, 2013.

[3] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures : The PLASMA and MAGMA projects. *Journal of Physics : Conference Series*, 180(1) :012037, 2009.

[4] 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.

[5] 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.

[6] 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.

[7] Hartwig Anzt, Werner Augustin, Martin Baumann, Hendryk Bockelmann, Thomas Gengenbach, Tobias Hahn, Vincent Heuveline, Eva Ketelaer, Dimitar Lukarski, Andrea Otzen, Sebastian Ritterbusch, Björn Rocker, Staffan Ronnas, Michael Schick, Chandramowli Subramanian, Jan-Philipp Weiss, and Florian Wilhelm. Hiflow3 – a flexible and hardware-aware parallel finite element package. *Preprint Series of the Engineering Mathematics and Computing Lab*, 0(06), 2013.

[8] C. Augonnet, S. Thibault, R. Namyst, and P.-A. 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.

[9] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí. An extension of the starss programming model for platforms with multiple GPUs. In H. J. Sips, D. H. J. Epema, and H.-X. Lin, editors, *Euro-Par*, volume 5704 of *Lecture Notes in Computer Science*, pages 851–862. Springer, 2009.

[10] R. M. Badia, J. R. Herrero, J. Labarta, J. M. Pérez, E. S. Quintana-Ortí, and G. Quintana-Ortí. Parallelizing dense and banded linear algebra libraries using SMPSs. *Concurrency and Computation : Practice and Experience*, 21(18) :2438–2456, 2009.

[11] W. Bangerth, R. Hartmann, and G. Kanschat. deal.II – a general purpose object oriented finite element library. *ACM Trans. Math. Softw.*, 33(4) :24/1–24/27, 2007.

[12] W. Bangerth, R. Hartmann, and G. Kanschat. Deal.ii&mdash ;a general-purpose object-oriented finite element library. *ACM Trans. Math. Softw.*, 33(4), August 2007.

[13] W. Bangerth, T. Heister, L. Heltai, G. Kanschat, M. Kronbichler, M. Maier, B. Turcksin, and T. D. Young. The deal.ii library, version 8.1. *arXiv preprint http://arxiv.org/abs/1312.2266v4*, 2013.

[14] Wolfgang Bangerth and Guido Kanschat. Concepts for object-oriented finite element software – the deal.ii. In *Preprint 43, SFB 359*, 1999.

[15] F Bassi, A Crivellini, S Rebay, and M Savini. Discontinuous Galerkin solution of the Reynolds-averaged Navier-Stokes and $k-\omega$ turbulence model equations. *Computers & Fluids*, 34(4-5) :507–540, MAY-JUN 2005. Workshop on Residual Distribution Schemes, Discontinuous Galerkin Schemes, Multidimensional Schemes and Mesh Adaptation, Univ Bordeaux I, Inst Math, Talence, FRANCE, JUN 23-25, 2002.

[16] P. Bellens, J. M. Pérez, F. Cabarcas, A. Ramírez, R. M. Badia, and J. Labarta. CellSs : Scheduling techniques to better exploit memory hierarchy. *Scientific Programming*, 17(1-2) :77–95, 2009.

[17] Filip Blagojević, Paul Hargrove, Costin Iancu, and Katherine Yelick. Hybrid pgas runtime support for multicore nodes. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 3 :1–3 :10, New York, NY, USA, 2010. ACM.

[18] G. Bosilca, A. Bouteiller, A. Danalis, T. Hérault, P. Lemarinier, and J. Dongarra. DAGuE : A generic distributed DAG engine for high performance computing. *Parallel Computing*, 38(1-2) :37–51, 2012.

[19] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Luszczek, and J. Dongarra. Dense linear algebra on distributed heterogeneous hardware with a symbolic dag approach. *Scalable Computing and Communications : Theory and Practice*, 2013.

[20] 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.

[21] D.A. Burgess, P.I. Crumpton, and M.B. Giles. A parallel framework for unstructured grid solvers. In *Programming Environments for Massively Parallel Distributed Systems*, Monte Verita, pages 97–106. 1994.

[22] E. Burman, A. Quarteroni, and B. Stamm. Interior penalty continuous and discontinuous finite element approximation of hyperbolic equations. *J.Sci.Comp.*, 43 :293–312, 2010.

[23] C.Dobrzynski. MMG3D home page. http://www.math.u-bordeaux1.fr/~dobrzyns/logiciels/mmg3d.php.

[24] Cris Cecka, Adrian J. Lew, and E. Darve. Assembly of finite element methods on graphics processors. *International Journal for Numerical Methods in Engineering*, 85(5) :640–669, 2011.

[25] 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 S. Chatterjee and M. L. Scott, editors, *PPOPP*, pages 123–132. ACM, 2008.

[26] NVIDIA Corporation. NVIDIA CUDA best practices guide version 4.0, 2011.

[27] Marta de la Llave Plata, Florent Renac, Emeric Martin, Jean-Baptiste Chapelier, and Vincent Couaillier. Application of a discontinuous galerkin method for the simulation of turbulent flow configurations on hybrid meshes.

[28] Andreas Dedner, Robert Klöfkorn, Martin Nolte, and Mario Ohlberger. A generic interface for parallel and adaptive discretization schemes : abstraction principles and the dune-fem module. *Computing*, 90(3-4) :165–196, 2010.

[29] D.A. DiPietro and A. Ern. Springer-Verlag, Berlin Heidelberg, 2012.

[30] Jean-Jacques Droux. An algorithm to optimally color a mesh. *Computer Methods in Applied Mechanics and Engineering*, 104(2) :249 – 260, 1993.

[31] 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.

[32] A. Ern and J-L. Guermond. Springer-Verlag, New-York LCC, 2004.

[33] F.Pellegrini. `Pt-Scotch` home page. http://www.labri.fr/perso/pelegrin/scotch/.

[34] Zhisong Fu, T. James Lewis, Robert M. Kirby, and Ross T. Whitaker. Architecting the finite element method pipeline for the GPU. *Journal of Computational and Applied Mathematics*, 257(0) :195 – 211, 2014.

[35] M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, and P. H.J. Kelly. Performance analysis of the op2 framework on many-core architectures. *SIGMETRICS Perform. Eval. Rev.*, 38(4) :9–15, March 2011.

[36] Nina Hanzlikova and Eduardo Rocha Rodrigues. A novel finite element method assembler for co-processors and accelerators. In *Proceedings of the 3rd Workshop on Irregular Applications : Architectures and Algorithms*, IA^3 '13, pages 1 :1–1 :8, New York, NY, USA, 2013. ACM.

[37] 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.

[38] Frédéric Hecht. New development in freefem++. *Journal of Numerical Mathematics*, 20(3-4) :251–266, 2012.

[39] E. Hermann, B. Raffin, F. Faure, T. Gautier, and Jérémie Allard. Multi-GPU and multi-CPU parallelization for interactive physics simulations. In P. D'Ambra, M. R. Guarracino, and D. Talia, editors, *Euro-Par (2)*, volume 6272 of *Lecture Notes in Computer Science*, pages 235–246. Springer, 2010.

[40] Vincent Heuveline. Hiflow3 : A flexible and hardware-aware parallel finite element package. In *Proceedings of the 9th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, POOSC '10, pages 4 :1–4 :6, New York, NY, USA, 2010. ACM.

[41] Peter Huthwaite. Accelerated finite element elastodynamic simulations using the GPU. *Journal of Computational Physics*, 257, Part A(0) :687 – 707, 2014.

[42] Hrvoje Jasak, Aleksandar Jemcov, and Zeljko Tukovic. Openfoam : A c++ library for complex physics simulations. In *International workshop on coupled methods in numerical dynamics*, volume 1000, pages 1–20, 2007.

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

[44] Dries Kimpe, Andrea Lani, Tiago Quintino, Stefaan Poedts, and Stefan Vandewalle. The coolfluid parallel architecture. In *Proceedings of the 12th European PVM/MPI Users' Group Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, PVM/MPI'05, pages 520–527, Berlin, Heidelberg, 2005. Springer-Verlag.

[45] Andreas Klöckner, Nicolas Pinto, Bryan C. Catanzaro, Yunsup Lee, Paul Ivanov, and Ahmed Fasih. Gpu scripting and code generation with pycuda. *CoRR*, abs/1304.5553, 2013.

[46] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. Pycuda and pyopencl : A scripting-based approach to gpu run-time code generation. *Parallel Comput.*, 38(3) :157–174, March 2012.

[47] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan C. Catanzaro, Paul Ivanov, and Ahmed Fasih. Pycuda : Gpu run-time code generation for high-performance computing. *CoRR*, abs/0911.3456, 2009.

[48] Andreas Klöckner, Timothy Warburton, and Jan S. Hesthaven. High-order discontinuous galerkin methods by gpu metaprogramming. *CoRR*, abs/1211.0582, 2012.

[49] Norbert Kroll, Heribert Bieler, Herman Deconinck, Vincent Couaillier, Harmen van der Ven, and Kaare Sørensen, editors. *ADIGMA - A European Initiative on the Development of Adaptive Higher-Order Variational Methods for Aerospace Applications*, volume 113 of *Notes on Numerical Fluid Mechanics and Multidisciplinary Design Volume*. Springer, 2010.

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

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

[52] Cédric Lachat, François Pellegrini, and Cécile Dobrzynski. PaMPA : Parallel Mesh Partitioning and Adaptation. In *21st International Conference on Domain Decomposition Methods (DD21)*, Rennes, France, June 2012. INRIA Rennes-Bretagne-Atlantique.

[53] Cédric Lachat, François Pellegrini, and Cecile Dobrzynski. PaMPA : Parallel Mesh Partitioning and Adaptation. In *DD22 - 22nd International Conference on Domain Decomposition Methods*, Lugano, Switzerland, September 2013. Institute of Computational Science (ICS), Università della Svizzera italiana (USI).

[54] Xavier Lacoste, Mathieu Faverge, Pierre Ramet, Samuel Thibault, and George Bosilca. Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes. Rapport de recherche RR-8446, INRIA, January 2014.

[55] Anders Logg, Kent-Andre Mardal, Garth N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012.

[56] Anders Logg and Garth N. Wells. Dolfin : Automated finite element computing. *ACM Transactions on Mathematical Software*, 37(2), 2010.

[57] C.-K. Luk, S. Hong, and H. Kim. Qilin : exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In D. H. Albonesi, M. Martonosi, D. I. August, and J. F. Martínez, editors, *MICRO*, pages 45–55. ACM, 2009.

[58] G. R. Markall, A. Slemmer, D. A. Ham, P. H. J. Kelly, C. D. Cantwell, and S. J. Sherwin. Finite element assembly strategies on multi-core and many-core architectures. *International Journal for Numerical Methods in Fluids*, 71(1) :80–97, 2013.

[59] E Martin and F Renac. Hybrid mpi/openmp parallel strategies for a high order discontinuous galerkin solver in aerodynamics.

[60] Kazimierz Michalik, Krzysztof Banaś, Przemysław Płaszewski, and Paweł Cybulka. Modular fem framework" modfem" for generic scientific parallel simulations. *Computer Science*, 14(3) :513, 2013.

[61] G.R. Mudalige, M.B. Giles, I. Reguly, C. Bertolli, and P.H.J. Kelly. Op2 : An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. In *Innovative Parallel Computing (InPar), 2012*, pages 1–12, May 2012.

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

[63] Asim Onder and Johan Meyers. Hpc realization of a controlled turbulent round jet using openfoam®. *status : accepted*, 2013.

[64] 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.

[65] Lars Pesch, Alexander Bell, Henk Sollie, Vijaya R. Ambati, Onno Bokhove, and Jaap J. W. Van Der Vegt. hpgem—a software framework for discontinuous galerkin finite element methods. *ACM Trans. Math. Softw.*, 33(4), August 2007.

[66] G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. van de Geijn, F. G. Van Zee, and E. Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Trans. Math. Softw.*, 36(3), 2009.

[67] Tiago Quintino. *A Component Environment for High-Performance Scientific Computing : Design and Implementation.* PhD thesis, Katholike Universiteit Leuven, von Karman Institute for Fluid Dynamics, Leuven, december 2008.

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

[69] W Smigaj, S Arridge, T Betcke, J Phillips, and M Schweiger. Solving boundary integral problems with bem++. *ACM Transactions on Mathematical Software*, 2012.

[70] F. Song, A. YarKhan, and J. 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.

[71] Christian Terboven, Alexander Spiegel, Dieter an Mey, Sven Gross, and Volker Reichelt. Experiences with the openmp parallelization of drops, a navier-stokes solver written in c++. In MatthiasS. Mueller, BarbaraM. Chapman, BronisR. de Supinski, AllenD. Malony, and Michael Voss, editors, *OpenMP Shared Memory Parallel Programming*, volume 4315 of *Lecture Notes in Computer Science*, pages 95–106. Springer Berlin Heidelberg, 2008.

[72] J.F. Thompson, B.K. Soni, and N.P. Weatherill. CRC Press LLC, 1998.

[73] Sean Treichler, Michael Bauer, and Alex Aiken. Realm : an event-based low-level runtime for distributed memory architectures. In José Nelson Amaral and Josep Torrellas, editors, *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, pages 263–276. ACM, 2014.

[74] F. Van Zee, E. Chan, R. van de Geijn, E. Quintana, and G. Quintana-Orti. Introducing : The Libflame library for dense matrix computations. *Computing in Science Engineering*, PP(99) :1, 2009.

[75] H. C. Yee, N. D. Sandham, and M. J. Djomehri. Low-dissipative high-order shock-capturing methods using characteristic-based filters. *J. Comput. Phys.*, 150(1) :199–238, 1999.

Personal Publications

[1] Damien Genet, Abdou Guermouche, and George Bosilca. Assembly Operations for Multicore Architectures using Task-Based Runtime Systems. In *Europar2014 : Parallel Processing Workshops*, 2014.

[2] Dragan Mbengoue, Damien Genet, Cedric Lachat, Emeric Martin, Maxime Mogé, Vincent Perrier, Florent Renac, Mario Ricchiuto, and François Rue. Comparison of high order algorithms in Aerosol and Aghora for compressible flows. *ESAIM : Proceedings*, 43 :1–16, December 2013.