HAL
open science

Scheduling and memory optimizations for sparse direct solver on multi-core/multi-gpu duster systems

Xavier Lacoste

To cite this version:

Xavier Lacoste. Scheduling and memory optimizations for sparse direct solver on multi-core/multi-gpu duster systems. Distributed, Parallel, and Cluster Computing [cs.DC]. Université de Bordeaux, 2015. English. NNT : 2015BORD0016 . tel-01222565

HAL Id: tel-01222565

https://theses.hal.science/tel-01222565v1

Submitted on 30 Oct 2015

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

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

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Xavier LACOSTE**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Scheduling and memory optimizations for sparse direct solver on multi-core/multi-gpu cluster systems

---

Soutenue le : 18 Février 2015

Après avis des rapporteurs :

Timothy A. Davis ..	Professor, Texas A&M University
Xiaoye Sherry Li	Senior Scientist, Lawrence Berkeley
.....	National Laboratory

Devant la commission d'examen composée de :

Fredéric Desprez ...	Senior researcher, Inria, Grenoble	Président du Jury
Alfredo Buttari ...	Researcher, CNRS IRIT, Toulouse	Membre du Jury
Iain S. Duff	Professor, Rutherford Appleton	Membre du Jury
.....	Laboratory, Oxfordshire	
Guillaume Latu	Research scientist, CEA Cadarache	Membre du Jury
Boniface Nkonga ..	Professor, Université de Nice, ...	Membre du Jury
François Pellegrini	Professor, Université de Bordeaux	Directeur de Thèse
Pierre Ramet	Assistant professor, Université de Bordeaux	Directeur de Thèse

– 2015 –

Acknowledgements

I would not have produced such a Ph.D. Thesis without many people I would like to thank here.

First, I would like to thank Sherry Li and Timothy Davis for accepting reviewing and evaluating my manuscript and allowing me to defend my Ph.D. thesis. They both provided a very useful and encouraging reports. Particularly, Tim who took the time to fix my often approximated English and to point the missing or not clear enough information in the manuscript.

I also want to thank the members of the committee that considered both my manuscript and my Ph.D. defense enough to promote me "Doctor". Frédéric Desprez, the president of the committee, who came to find a very interesting question while the other members had stolen all the ones he prepared ;). Alfredo Buttari, with whom I enjoyed discussing and explaining our method more in detail while he was reviewing my manuscript with attention. Iain Duff, who kindely spoke slowly and with some French words in its question so that I can understand him easily and be in good condition for the following of the questions. Guillaume Latu, for its touching words and with whom I enjoyed to work very much during my Ph.D. Thesis and before it. And Boniface Nkonga, with whom its always enjoyable to work thanks to its unbreakable good mood.

I couldn't have done much during this thesis without my directors. François Pellegrini, "l'homme de paille", who taught me how to appreciate both Scotch and Scotch ;). Pierre Ramet, for these 7 years of work together that where full of good memories, its valuable advices, and help. I also enjoyed sharing many fun times: squash, skiing weekends, barbecue... And Mathieu Faverge, "l'homme de l'ombre", for all the work we have done together, from the research, through the code, to the manuscript.

I am also grateful to my colleagues, from ScalApplix, Bacchus, and HiePACS for the good working atmosphere and the after work times. My work with the StarPU and PaRSEC developers team was always rewarding, and I appreciate their high reactivity. I also liked working in Cadarache in the CEA team as in Bidart with the Algo'Tech PME's colleagues where I received a warm welcome each time I visited one of the coworkers.

I would also thank all my family and friends, with whom I have so many happy memories, and that support me during and before this Ph.D. thesis.

Last but not least, I also thank my Darling, Algiane, loving better half and future mother of my child(ren? ;p), who fulfill my life with so much love since nearly five years. I'm so happy thinking to all the good moment we had together and the ones that are coming that my smile cannot leave my face while I am finishing writing this!

Ordonnancement et optimisations mémoire pour un solveur creux par méthodes directes sur des machines hétérogènes

Résumé : L'évolution courante des machines montre une croissance importante dans le nombre et l'hétérogénéité des unités de calcul. Les développeurs doivent alors trouver des alternatives aux modèles de programmation habituels permettant de produire des codes de calcul à la fois performants et portables. PaStiX est un solveur parallèle de système linéaire creux par méthodes directe. Il utilise un ordonnanceur de tâche dynamique pour être efficaces sur les machines modernes multi-cœurs à mémoires hiérarchiques. Dans cette thèse, nous étudions les bénéfices et les limites que peut nous apporter le remplacement de l'ordonnanceur interne, très spécialisé, du solveur PaStiX par deux systèmes d'exécution génériques : PaRSEC et StarPU. Pour cela l'algorithme doit être décrit sous la forme d'un graphe de tâches qui est fournit aux systèmes d'exécution qui peuvent alors calculer une exécution optimisée de celui-ci pour maximiser l'efficacité de l'algorithme sur la machine de calcul visée. Une étude comparative des performances de PaStiX utilisant ordonnanceur interne, PaRSEC, et StarPU a été menée sur différentes machines et est présentée ici. L'analyse met en évidence les performances comparables des versions utilisant les systèmes d'exécution par rapport à l'ordonnanceur embarqué optimisé pour PaStiX. De plus ces implémentations permettent d'obtenir une accélération notable sur les machines hétérogènes en utilisant les accélérateurs tout en masquant la complexité de leur utilisation au développeur. Dans cette thèse nous étudions également la possibilité d'obtenir un solveur distribué de système linéaire creux par méthodes directes efficace sur les machines parallèles hétérogènes en utilisant les systèmes d'exécution à base de tâche. Afin de pouvoir utiliser ces travaux de manière efficace dans des codes parallèles de simulations, nous présentons également une interface distribuée, orientée éléments finis, permettant d'obtenir un assemblage optimisé de la matrice distribuée tout en masquant la complexité liée à la distribution des données à l'utilisateur.

Mots clés : Résolution de systèmes linéaires creux, ordonnanceur à base de tâches, MPI, multi-cœur, GPU.

Discipline : Informatique

LaBRI (UMR CNRS 5800)
Université de Bordeaux,
351, cours de la libération
33405 Talence Cedex, FRANCE

Equipe Projet InriaHiePACS[1],
Inria Bordeaux – Sud-Ouest-200818243Z,
200, avenue de la vieille tour,
33405 Talence Cedex, FRANCE

[1]HiePACS: High-End Parallel Algorithms for Challenging Numerical Simulations, `https://team.inria.fr/hiepacs/`.

Scheduling and memory optimizations for sparse direct solver on multi-core/multi-gpu cluster systems

Abstract: The ongoing hardware evolution exhibits an escalation in the number, as well as in the heterogeneity, of computing resources. The pressure to maintain reasonable levels of performance and portability forces application developers to leave the traditional programming paradigms and explore alternative solutions. PaStiX is a parallel sparse direct solver, based on a dynamic scheduler for modern hierarchical manycore architectures. In this thesis, we study the benefits and the limits of replacing the highly specialized internal scheduler of the PaStiX solver by two generic runtime systems: PaRSEC and StarPU. Thus, we have to describe the factorization algorithm as a tasks graph that we provide to the runtime system. Then it can decide how to process and optimize the graph traversal in order to maximize the algorithm efficiency for the targeted hardware platform. A comparative study of the performance of the PaStiX solver on top of its original internal scheduler, PaRSEC, and StarPU frameworks is performed. The analysis highlights that these generic task-based runtimes achieve comparable results to the application-optimized embedded scheduler on homogeneous platforms. Furthermore, they are able to significantly speed up the solver on heterogeneous environments by taking advantage of the accelerators while hiding the complexity of their efficient manipulation from the programmer. In this thesis, we also study the possibilities to build a distributed sparse linear solver on top of task-based runtime systems to target heterogeneous clusters. To permit an efficient and easy usage of these developments in parallel simulations, we also present an optimized distributed interface aiming at hiding the complexity of the construction of a distributed matrix to the user.

Keywords: Sparse direct solver, Tasks based runtime systems, MPI, multi-core, GPU.

Discipline : Computer science

LaBRI (UMR CNRS 5800)
Université Bordeaux 1,
351, cours de la libération
33405 Talence Cedex, FRANCE

Equipe Projet Inria HiePACS[2],
Inria Bordeaux – Sud-Ouest-200818243Z,
200, avenue de la vieille tour,
33405 Talence Cedex, FRANCE

[2]HiePACS: High-End Parallel Algorithms for Challenging Numerical Simulations, `https://team.inria.fr/hiepacs/`.

Afin d'obtenir une connaissance toujours plus précise de leur environnement, les scientifiques n'ont cessé de mettre en place des expériences permettant l'explication et la prédiction des évènements. Avec l'avènement des ordinateurs, une nouvelle discipline a permis l'accélération de ce processus : la science informatique. En effet, les ordinateurs sont capables de simuler des expériences pour valider de nouvelles idées ou conceptions avec une précision jamais atteinte auparavant. L'utilisation de simulations numériques permet de réduire le coût, de simplifier, d'accélérer et de rendre plus sûr les expérimentations. Ces simulations peuvent intervenir dans des domaines d'applications très variés comme l'aérodynamique, les prospections pétrolières, la météorologie, la biologie, la finance... Cette liste n'est pas exhaustive, mais met en évidence le fait que les simulations numériques peuvent intervenir dans de très nombreux domaines. Ainsi, la modélisation numérique est un domaine de recherche très actif, les simulations doivent être précises et rapides. En particulier, la fusion nucléaire est un domaine de recherche prometteur pour pourvoir à la consommation énergétique de notre futur et requiert des simulations très précises. Le prototype de réacteur à fusion ITER (International Thermonuclear Experimental Reactor) en construction à Cadarache, en France, en est un bon exemple. Le code JOREK, qui sera évoqué dans cette thèse, est développé pour aider à l'élaboration et à la calibration des pièces qui vont le composer.

En général, le problème considéré ne peut pas être résolu de manière continue sur le domaine complet, mais doit être étudié en de nombreux points. La séparation du domaine d'étude en multiples points s'appelle la discrétisation. Les équations discrètes correspondent à une restriction des équations continues mathématiques sur la discrétisation du domaine. Ces équations discrètes forment un système d'équations qui peuvent également être représentées sous la forme d'une matrice dont les entrées correspondent aux coefficients du système. Ce système peut soit être dense si tous les points interagissent entre eux, la matrice correspondante ne contient alors quasiment pas de termes nuls et est appelée *dense*, soit creux si seuls les points proches interagissent les uns sur les autres, la matrice contient alors de nombreux termes nuls et est appelée *creuse*. Lorsque l'on veut obtenir une solution plus précise ou que le domaine d'étude devient plus grand, le nombre de point de discrétisation augmente et avec lui la taille du système à résoudre. On peut alors obtenir des problèmes avec des dizaines de milliers d'inconnues. Par exemple, en météorologie, pour capturer tous les phénomènes présent dans l'atmosphère, la simulation doit être très précise, avec une discrétisation la plus fine possible. Cependant, on veut également obtenir la solution en un temps raisonnable (e.g. une prédiction météorologique n'a d'utilité que si elle est disponible avant qu'elle ne soit passée) alors que le temps requis pour obtenir la solution augmente avec le nombre d'inconnues (de

manière linéaire à cubique en fonction de la méthode). Il faut alors soit réduire la précision en espaçant les points de discrétisation, soit accélérer la simulation numérique pour obtenir la solution en temps voulu. Les scientifiques ont donc développé des techniques de calcul parallèle permettant de bénéficier de la puissance de plusieurs ordinateurs pour une même simulation numérique. De plus, différente méthodes ont été développées pour résoudre les systèmes linéaires produits par les simulations numériques. Ces méthodes diffèrent par leur complexité et la précision de la solution obtenue. Grâce à ces techniques et à des processeurs toujours plus puissants les scientifiques peuvent augmenter à la fois la taille des domaines d'étude et la finesse de discrétisation et résoudre alors des problèmes de grande taille dans le même temps de calcul.

Les constructeurs de machine ont malheureusement atteint une limite empêchant l'augmentation de la puissance d'un cœur de calcul. En effet, l'augmentation de la dissipation thermique [Sut05] a stoppé la croissance de la fréquence d'horloge des puces depuis 2004. Durant les 10 dernières années, la puissance de calcul des machines a pu augmenter grâce au puces dotées de multiples cœurs capable d'effectuer un grand nombre d'opérations par cycle. Les grappes de machine de calcul haute performance rassemblent aujourd'hui des nœuds de 16 à 64 cœurs inter-connectées par des réseaux haute performance. Ces machines offrant un accès hiérarchisé à la mémoire sont appelées nœuds NUMA[7]. La tendance actuelle dans le calcul haute performance est d'assister les cœurs de calcul traditionnels avec des accélérateurs permettant l'accélération d'une certaine catégorie d'opérations. Cette idée d'utiliser un accélérateur n'est pas nouvelle dans le domaine du calcul haute performance mais les cartes graphiques, détournées pour le calcul, ont l'avantage d'avoir un rapport puissance-prix très avantageux. Ces accélérateurs sont massivement parallèles et nécessitent des calculs très réguliers sur un vecteur de données (Ce sont de unités de calcul SPMD[8]). Intel produit également un accélérateur *many-core*, l'Intel Xeon Phi, pour lequel le programmeur doit également extraire beaucoup de parallélisme de ses algorithmes pour obtenir de la performance. Toutes ces puces requièrent des optimisations spécifiques pour être utilisent efficacement. Pour exploiter ces nouvelles machines hétérogènes de nouveaux paradigmes de programmation ont été proposés. Parmi ceux-ci, les supports d'exécution à base de tâches proposent de séparer l'algorithme de la gestion de l'architecture de la machine. Les supports d'exécution à base de tâche sont des intergiciels qui sont capables d'estimer la concurrence entre les tâches et de les exécuter en parallèle sur toutes les unités de calcul d'une machine. Ces supports d'exécution sont également capables d'effectuer les transferts de données entre les unités de calcul. Ainsi, l'application n'a plus à gérer la cohérence des données. Grâce à ces outils, le développeur peut produire un code plus facile à maintenir et la portabilité des performances est plus facile à obtenir grâce à la séparation de l'algorithme et de la gestion du matériel.

La communauté de l'algèbre linéaire dense a déjà mené de nombreuses études concernant le calcul sur machines hétérogènes, souvent en utilisant des supports d'exécution à base de tâches [Pla+09; Bos14]. Les problèmes d'algèbre linéaire dense sont bien

[7]Non-Uniform memory access
[8]Single Process Multiple Data

adaptés à l'utilisation des accélérateurs grâce à la régularité de leurs algorithmes et à la possibilité d'adapter la granularité de leur données au besoin d'un support d'exécution. Au contraire, les systèmes linéaires creux sont très irréguliers, avec de petites données de tailles irrégulières imposées par le problème considéré. Ainsi, les bénéfices que l'on pourrait tirer des GPUs sur des problèmes d'algèbre linéaire creuse ne sont pas évidents et le surcoût lié à l'utilisation d'un support d'exécution pourrait être trop important comparé à la granularité très fine des tâches produites. L'objectif principal de cette thèse est d'étudier les effets de l'utilisation d'un support d'exécution générique pour la résolution de systèmes linéaires creux par méthode directe comparée à l'utilisation d'un ordonnanceur dédié et optimisé pour ce problème. En utilisant ces supports d'exécution à base de tâche, nous pouvons séparer l'algorithme de la gestion de l'architecture et exploiter les accélérateurs dans un solveur creux par méthode directe. Les solveurs par méthode directe produisent un graphe de tâche direct et acyclique, il est donc naturel d'utiliser les supports d'exécution à base de tâche pour ordonnancer ces tâches sur des machines hétérogènes. Parmi les supports d'exécution à base de tâches existants, nous avons choisi deux candidats pour cette étude : StarPU et PaRSEC. Les solutions proposées ont été validées la bibliothèque PaStiX, développée par Inria Bordeaux - Sud-ouest, qui permet la résolution de systèmes linéaires creux par méthode directe. Cette thèse se situe dans le contexte du projet ANR[9] ANEMOS qui a pour but de développer des outils de simulation de plasma pour la fusion contrôlée dans les réacteurs nucléaires. Parmi ces outils, JOREK est un code de simulation utilisant des éléments finis pour discrétiser le domaine. Ces éléments produisent des matrices élémentaires qui sont utilisées pour calculer le système d'équation global. Un autre objectif de cette thèse est de proposer une solution simple et efficace permettant l'assemblage d'une matrice distribuée pour l'utilisation de solveurs creux dans des codes utilisant les éléments finis. Cette thèse propose une interface de programmation qui s'adapte à la décomposition originale du maillage pour le parallélisme et est responsable de toutes les communications à la frontière de chaque sous-domaine. Cette interface de programmation est générique et pourrait être utilisée dans la plupart des simulations numériques pour appeler n'importe quel bibliothèque de résolution de systèmes linéaires creux.

Dans le premier chapitre de cette thèse, nous décrivons le contexte général du calcul haute performance. Nous détaillons les différentes architectures qui ont été développées pour effectuer des calculs de manière efficace. Ensuite, nous expliquons les différentes techniques disponibles pour exploiter les nouvelles architectures hétérogènes. Après cela, nous décrivons les différentes méthodes permettant la résolution de systèmes linéaires creux sur ces machines en insistant particulièrement sur les solveurs de systèmes linéaires creux par méthode directe qui sont le sujet principal de cette thèse. Enfin, ce chapitre décrit les nouvelles machines hétérogènes et comment les bibliothèques d'algèbre linéaire ont été adaptées pour être efficaces sur celles-ci.

Le second chapitre présente les modifications que nous avons apportées au solveur de systèmes linéaires creux par méthode directe PaStiX pour l'adapter aux machines

[9] Agence National pour la Recherche

hétérogènes à mémoire partagée. Il introduit la bibliothèque de résolution de systèmes linéaires creux en détail et les deux supports d'exécution à base de tâches utilisés dans cette étude : StarPU développé par Inria Bordeaux - Sud-ouest, et PaRSEC développé par l'ICL[10] à l'université du Tennessee, à Knoxville. Puis, nous détaillons les changements apportés à PaStiX pour permettre l'utilisation des supports d'exécution à base de tâche et comparons les performances de la nouvelle implémentation avec celles de l'ordonnanceur statique dédié et optimisé pour les machines multicœur de type NUMA. Sur des nœuds à mémoire partagée nous présentons des résultats avec l'implémentation utilisant les supports d'exécution comparables à ce que l'on peut obtenir avec l'ordonnanceur d'origine. Puis, nous fournissons un nouveau noyau de calcul au support d'exécution pour déporter les calculs les plus intensifs (i.e. les produits matriciels) sur les GPUs. Les expériences sur machines hétérogènes montre une accélération jusqu'à 2,8 fois en utilisant trois GPUs en plus des douze cœurs classiques.

Dans le troisième chapitre, nous nous attaquons aux problèmes de l'utilisation de la version de PaStiX basée sur les supports d'exécution à base de tâche sur des plates-formes distribuées de machines hétérogènes. Nous présentons les nouveaux challenges qui apparaissent sur une machine à mémoire distribuée et étudions l'implémentation de deux algorithmes parallèles en utilisant des supports d'exécution génériques. Les expériences montrent des résultats encourageants sur une grappe de machines hétérogènes.

Le quatrième chapitre présente l'intégration de nos travaux dans JOREK, un code de production simulant la fusion de plasma contrôlée développé par le CEA Cadarache. Nous décrivons ici une interface de programmation générique permettant l'assemblage d'une matrice distribuée et le contrôle d'un solveur dans un code utilisant des éléments finis. Le but de cette interface de programmation est d'optimiser et de simplifier la construction d'une matrice distribuée qui sera fournie en entrée à la bibliothèque PaStiX. La distribution de cette matrice a pour but d'améliorer le passage à l'échelle de l'application en terme de mémoire. Les expériences montrent que par l'utilisation de cette interface, nous avons pu réduire la consommation mémoire en passant d'une matrice centralisée à une matrice distribuée en entrée du solveur. L'assemblage de la matrice a également pu être accéléré grâce à une optimisation du volume de communication.

Finalement, nous résumons l'ensemble des résultats obtenus durant cette thèse et proposons de nouvelles orientations pour les travaux de recherche qui feront suite à cette thèse.

[10]The Innovative Computing Laboratory

Contents

Introduction **1**

1 Linear algebra on modern architectures **5**

1.1 Parallel architectures 6

1.1.1 Multi-core machines 6

1.1.2 Accelerators 8

1.2 Addressing parallel machines 10

1.2.1 Parallel programming 11

1.2.2 Addressing accelerators 13

1.2.3 Task-based runtime systems 14

1.3 Sparse linear algebra methods 16

1.3.1 Prerequisite: Basic Linear Algebra Subroutines 17

1.3.2 Solving a sparse linear system 18

1.3.3 Description of the direct methods 19

1.3.4 Linear algebra on homogeneous clusters 29

1.3.5 Linear algebra on heterogeneous clusters 31

1.4 Discussion 34

2 Sparse factorization on shared-memory heterogeneous machines **37**

2.1 Framework 39

2.1.1 PaStiX original algorithm description 39

2.1.2 Elected runtime systems 41

2.2 Implementation on top of generic runtime systems 49

2.2.1 PaRSEC implementation 50

2.2.2 StarPU implementation 53

2.2.3 Multi-core Architecture experimentation 55

2.3 Heterogeneous systems . . . 57
2.3.1 Implementation of a specific `GEMM` kernel . . . 60
2.3.2 Data mapping over multiple GPUs . . . 66
2.3.3 Heterogeneous experiments . . . 69
2.3.4 Memory study . . . 69
2.4 Optimizations . . . 71
2.4.1 Task granularity adapted to the runtime . . . 71
2.4.2 Block splitting algorithm . . . 73
2.5 Discussion . . . 79

3 Sparse factorization on distributed heterogeneous systems 83
3.1 Solving distributed sparse linear system . . . 84
3.1.1 *Fan-out* implementation . . . 84
3.1.2 *Fan-in* implementation . . . 86
3.1.3 Data mapping . . . 89
3.2 Experiments . . . 92
3.2.1 Distributed implementation on homogeneous nodes . . . 92
3.2.2 Distributed implementation on heterogeneous nodes . . . 97
3.3 Discussion . . . 99

4 Integration in a controlled plasma fusion simulation code: JOREK 103
4.1 Description of the framework . . . 104
4.1.1 Set of equations . . . 105
4.1.2 Spatial discretization . . . 105
4.1.3 Time integration scheme . . . 106
4.1.4 Equilibrium . . . 106
4.1.5 Sparse solver and preconditioning . . . 107
4.2 Assembly step in JOREK . . . 108
4.3 Optimized distributed matrix assembly . . . 112
4.3.1 Generic distributed finite element assembly oriented API . . . 113
4.3.2 Comparison with PETSc . . . 115
4.4 Integration into JOREK . . . 118
4.4.1 Implementation . . . 118
4.4.2 Timing and memory scaling study . . . 119

4.5 Discussion 121

Conclusion and perspectives **123**

Appendixs **127**

Bibliography **127**

A Publications **141**

B Integration in Algo'Tech software **143**

B.1 Algo'Tech software simulation tool 143

B.2 Optimizations 144

B.2.1 PaStiX integration 144

B.2.2 Parallelization 144

B.2.3 Cloud computing 144

B.2.4 Numerical optimization 145

B.3 Conclusion 145

C Murge and Jorek code samples **147**

D Sparse matrix storage formats **153**

List of Figures

1.1 A four CPUs SMP machine. 7
1.2 A four CPUs NUMA machine. 7
1.3 Architecture of the NVIDIA Kepler GK110. 9
1.4 Architecture of the Intel Xeon Phi. 11
1.5 A Mirage node described using Hardware Locality (hwloc) tool. 12
1.6 Reordering of the unknowns. 21
1.7 A block structure factorized matrix and the associated elimination tree. . 23
1.8 Description of the notations used for the factorization algorithm. 24
1.9 Multifrontal algorithm. 25

2.1 Notations used in algorithms 42
2.2 Comparison of dense and sparse task decomposition. 51
2.3 CPU scaling study on 12 cores. 58
2.4 CPU scaling study on 48 cores. 59
2.5 Description of the panel update task. 61
2.6 Description of the GPU sparse `GEMM` update operation. 62
2.7 Performance study on the `DGEMM` kernel (sparse vs dense). 63
2.8 Performance study on the `DGEMM` kernel (sparse vs multiple CUDA calls). 65
2.9 Effects of commutable tasks on the graph. 66
2.10 Sort criteria study. 68
2.11 GPU scaling study. 70
2.12 Memory consumption comparison. 72
2.13 Comparison of panel splitting. 74
2.14 Flop/s during factorization depending on authorized variation in column block splits. 77
2.15 Block's size study. 78

3.1 Distributed example. 85
3.2 Distributed algorithm's task graph. 88
3.3 *Fan-In* distributed example. 90
3.4 Proportional mapping. 91
3.5 Simulated static scheduling. 92
3.6 Comparison of distributed implementations of PaStiX original scheduler. 94

3.7 Comparison of distributed implementations: original scheduler vs StarPU. 95
3.8 Comparison of distributed implementations: dynamic scheduling vs StarPU. 96
3.9 Comparison of distributed implementations on the 10 Millions test case. . 98
3.10 Distributed heterogeneous scaling study on 4 nodes. 100
3.11 Distributed heterogeneous scaling study on 16 nodes. 101

4.1 A tokamak. 104
4.2 Bezier 2D representation of a tokamak's plan. 106
4.3 JOREK main steps diagram. 109
4.4 Part of JOREK's elementary matrix. 110
4.5 JOREK's assembled matrix. 111
4.6 Different assembly methods. 115
4.7 Timing scaling study comparing PETSc and MURGE assembly. 117
4.8 Time and memory comparison on model 302 (2 harmonics). 121
4.9 Time and memory comparison on model 303 (2 harmonics). 122
4.10 Time and memory comparison on model 303 (4 harmonics). 122

D.1 An example of CSC matrix. 153
D.2 An example of CSCD matrix. 154

List of Tables

2.1 Descriptions of the matrices. 56
2.2 Comparison of the different splitting methods. 76

3.1 Comparison of the *fan-in* memory overhead. 90

List of Algorithms

1 Multi-frontal factorization: $A = LU$. 26
2 Right looking blocked sequential factorization: $A = LU$. 26
3 Left looking blocked sequential factorization: $A = LU$. 27
4 Parallel column-block factorization on thread t of process p. 43
5 StarPU tasks insertion algorithm. 54
6 Adapted splitting algorithm. 79
7 Choice of the split: largest first. 80
8 Choice of the split: average first. 81
9 *Fan-out* Cholesky implementation with StarPU. 87
10 *Fan-in* Cholesky implementation with StarPU. 89
11 Frequencies loop using a classical direct solver. 145
12 Frequencies loop with direct preconditionned iterative solver. 146
13 Assembly algorithm using original PaStiX interface. 150
14 Assembly algorithm using MURGE API (1/2). 151
15 Assembly algorithm using MURGE API (2/2). 152

Introduction

Scientists have always wanted to acquire more accurate knowledge of their surroundings. They realized experiments to explain and predict the universe. With the computers emergence, a new discipline that greatly accelerated this process appeared: computational science. Indeed, computers are capable of simulating experiments that can validate new concepts or designs with an accuracy never seen before. Indeed, using numerical simulations one can avoid setting up expensive, complicated, time-consuming, or dangerous experiments. Moreover, simulations can be used in various fields such as: aerodynamic field, oil prospecting, meteorology, biology, finance, etc. This list is not exhaustive but shows how simulations can affect many fields on a day to day basis environment. Thus, numerical simulation is a very active research field: simulations must be both accurate and fast. Particularly, nuclear fusion reactors are a promising research field for our future as a new source of energy, and require very accurate simulations. A good example is the nuclear fusion reactor prototype ITER (International Thermonuclear Experimental Reactor) built in Cadarache, France and for which a simulation code, JOREK, has been developed to design and calibrate its pieces.

Generally, the studied problem cannot be solved continuously on the whole domain, but it has to be studied in many points. The separation of the domain into many pieces is called the discretization. The discrete equations are a restriction of the mathematical continuous equations on the domain discretization. These discrete equations form a system of equations that can also be viewed as a matrix where the entries are the coefficients of the system. This system can be either dense, if we consider each point of the discretization has an influence to all other points. Then, a corresponding matrix that stores those influences contains nearly no nil values and is called *dense*. In some cases, some of the interactions can be neglected, which translates to nil entries in the matrix. When the number non zeros entries is of the same order of magnitude as its number of equations, the matrix is said to be *sparse*. As the accuracy of the expected solution and the size of the problem increase, the number of unknowns in the system gets larger, reaching several tens of millions unknowns in many simulations. For example, to capture all the phenomena in the atmosphere, a meteorologic simulation must be very sharp, with the smallest distance between discretization points as possible. However, one would want to obtain the solution as fast as possible (e.g. a weather forecast should be computed before it happened), but the time required to solve the system increases (linear to cubic increase depending on the method) with the number of unknowns. Ei-

ther we reduce the accuracy of the solution by spreading the discretization points, or we accelerate the numerical simulation to obtain the solution in a bounded time. Thus, scientists developed parallel techniques to benefit from multiple computers within one simulation. Furthermore, different methods have been developed to solve the linear systems induced by those simulations, varying in term of complexity and accuracy. Thanks to these techniques and increasingly more efficient chips, scientists can increase the domain size and use a more accurate representation to solve problems with a larger number of unknowns in the same computational time.

Nowadays, machine manufacturers have reached a limit in the performance of a single computational core. Indeed, the clock frequency of chips has not improved significantly since 2004 because of the heat dissipation increase [Sut05]. In the last decade, machines computational power has been increased using multi-core chips able to compute a larger number of operations per cycle. High performance computing clusters are now based of about nodes with 16 to 64 cores interconnected with high performance networks. These machines feature a hierarchical access to the memory and are called NUMA[3] nodes. The current trend in high performance computing is to assist the traditional cores with accelerators to speed-up certain categories of operation. This idea of using an accelerator is not new in high performance computing, but the new usage of Graphic Processing Units for computations brought to manufacturers affordable accelerators. These accelerators are massively parallel and require very regular computations on a vector of data (they are SPMD[4] computational units). Intel has also produced a many-core accelerator, the Intel Xeon Phi, which also requires the programmer to extract a lot of parallelism from its algorithms. All these different chips require specific programming optimizations to be used efficiently. To address these complex heterogeneous machines, new parallel programming models have been designed. Among them, the task-based runtime systems propose to isolate the algorithm from the architecture management. Task-based runtime systems are middle-wares that have been designed to estimate concurrency between tasks and execute them in parallel using all the computational devices of a machine. These runtime systems are also capable of moving data between the devices to relieve the application developer from taking care of data coherency. Using these systems, the developer can obtain more adaptability of the code, and the performance portability is more easily reachable thanks to the separation of the algorithm from the hardware management.

The dense linear algebra community has led multiple studies on heterogeneous computing, often using task-based runtime systems [Pla+09; Bos14]. Dense linear algebra problems are well suited to the usage of accelerators thanks to their regular pattern and the granularity of their data that can be easily adapted to the needs of a runtime system. On the contrary, sparse linear systems are very irregular, with small data of various sizes imposed by the considered problem. Thus, the benefit one can expect from the GPU on sparse linear system is not obvious and the overhead of the runtime system may be too large compared to the fine granularity of the tasks. The main objective of this thesis is to study the effect of these generic runtime systems in sparse linear algebra compared

[3]Non-Uniform memory access
[4]Single Process Multiple Data

to hand-tuned dedicated schedulers. Using these task-based runtime systems, we can decouple the algorithm from the architecture, and exploit accelerators in sparse direct solvers. Direct linear solvers produce a direct acyclic graph of tasks; thus, it is natural to use a task-based runtime to schedule these tasks on parallel heterogeneous clusters. Among the existing task-based runtime systems, we elected two different candidates to perform this study: StarPU and PaRSEC. The proposed solutions have been validated in the sparse linear direct solver library PaStiX developed at Inria Bordeaux - Sud-Ouest. This thesis takes place in the context of the ANR[5] ANEMOS project that aims at producing simulation tools for controlled plasma fusion in nuclear reactors. Among these tools, JOREK simulation code uses finite elements to discretize the domain. These elements produce elementary matrices that are used to build the global system of equation. Another objective of this thesis is to propose an efficient and easy way to assemble a distributed matrix for a sparse solver in finite element simulation codes. This thesis proposes an API that follows the original mesh parallel decomposition and is in charge of all required communications at the border of each parallel sub-domain. The API is generic and could be used above any linear solver, in any simulation code.

In the first chapter of this thesis, we describe the global framework of high performance computing. We detail the different architectures that have been developed to perform efficient computation. Then, we explain the different methods available to exploit the new heterogeneous parallel machines. Those methods that have been developed to address sparse linear systems are described, and we particularly focus on sparse direct algebra parallel solvers, which are the target of this thesis. Then, this chapter describes emerging heterogeneous architectures and how linear algebra libraries have been modified to fit these machines.

The second chapter presents how PaStiX direct solver has been modified to handle shared memory heterogeneous nodes. It introduces the PaStiX direct solver library in detail and the two task-based runtime systems used in this study: StarPU developed at Inria Bordeaux - Sud-Ouest and PaRSEC from ICL[6] at the University of Tennessee, Knoxville. Then, we detail the changes we have performed in PaStiX to use task-based runtime systems and compare the performance of the new implementation with the highly tuned original static scheduler on multi-core NUMA nodes. On shared-memory homogeneous nodes, we show performance results with the implementation of the sparse linear decomposition on top of generic runtime systems comparable to the original scheduler. Then, we provide a new kernel to the runtime system to offload the most compute intensive tasks (i.e. matrix-matrix products) on the GPUs. The heterogeneous experiments show up to 2.8 time speed up when adding three GPUs to the twelve basic cores.

In the third chapter, we tackle the problem of using the task-based runtime implementation of PaStiX on distributed platforms of heterogeneous nodes. We present the new difficulties that arise on a distributed cluster, and we study the implementation of

[5]Agence National pour la Recherche, a French agency that provides funding for project-based research
[6]The Innovative Computing Laboratory

two different parallel algorithms on top of task-based runtime systems. We evaluate the performance of their implementations. Experiments present encouraging results on a cluster of heterogeneous node.

The fourth chapter deals with the integration of our work in JOREK, a production controlled plasma fusion simulation code from CEA Cadarache. We describe here a generic finite element oriented distributed matrix assembly and solver management API. The goal of this API is to optimize and simplify the construction of a distributed matrix which, given as an input to PaStiX, can improve the memory scaling of the application. Experiments exhibit that using this API we could reduce the memory consumption by moving to a distributed matrix input and improve the performance of the factorized matrix assembly by reducing the volume of communication.

Finally, we summarize the results obtained in the context of this thesis and propose new directions opened by these works.

Chapter 1

Linear algebra on modern architectures

Contents

1.1 Parallel architectures 6
1.1.1 Multi-core machines 6
1.1.2 Accelerators 8
1.1.2.1 Graphical Processing Units 8
1.1.2.2 Intel Xeon Phi 10
1.2 Addressing parallel machines 10
1.2.1 Parallel programming 11
1.2.2 Addressing accelerators 13
1.2.3 Task-based runtime systems 14
1.2.3.1 Generic runtime systems 15
1.2.3.2 Runtime systems specifically designed for linear algebra 15
1.3 Sparse linear algebra methods 16
1.3.1 Prerequisite: Basic Linear Algebra Subroutines 17
1.3.2 Solving a sparse linear system 18
1.3.3 Description of the direct methods 19
1.3.3.1 Reordering of the unknowns 20
1.3.3.2 Symbolic factorization 22
1.3.3.3 Factorization methods 22
1.3.3.4 Triangular system solve 28
1.3.4 Linear algebra on homogeneous clusters 29
1.3.5 Linear algebra on heterogeneous clusters 31
1.3.5.1 Dense linear algebra evolution 31
1.3.5.2 Sparse linear solver on heterogeneous machines 32
1.4 Discussion 34

Numerical simulations require more and more computational power to reach higher accuracy in a shorter time and desktop computers quickly expose their limits. To bypass this limit, parallel machines have been developed and keep evolving. To efficiently exploit these machines, high performance algorithms have to follow the evolution of the machines architectures. In particular, the linear system solve is one of the most intensive computational tasks of many numerical simulations and algorithms' developers have to follow closely the transformation of parallel architectures.

In this chapter, we first present the current evolution of parallel machine architectures. We describe the now common multi-core machines and the different many-core architectures proposed by the manufacturers. Then, we introduce the different techniques and languages available to address these machines efficiently. And finally, we describe how sparse linear algebra developers have recently updated their software to target those powerful and complex machines.

Once the high performance context has been set up, we look into the main subject of this thesis: "sparse linear algebra solvers." First, we describe the multiple solutions available to solve sparse linear systems, exposing the advantages and the drawbacks of each solution. After that, we specifically focus on direct factorization methods, describe the different steps they are based on, and exhibit the characteristics of each existing algorithm. Several sparse linear solvers implement these algorithms, we present them and specify the differences and advantages of each implemented methods. These solvers already handle clusters of multi-core nodes, they are now tackling the heterogeneous architectures. This chapter finally describes the techniques used to target parallel clusters and how sparse linear solvers are evolving to take into account emerging heterogeneous clusters.

1.1 Parallel architectures

Machines kept getting more complex to get more efficiency while using and dissipating less energy. In this section, we describe the different architectures that emerged to increase the effectiveness of our computers, specifically in the field of high performance computing. Two solutions are available to increase computational power. The simplest one is the multiplication of classical cores that will execute tasks in parallel. Then, more specialized computing units, called accelerators, were proposed besides the classical cores. This section details these heterogeneous computing environment.

1.1.1 Multi-core machines

During last decade, multi-core machines have been widely generalized. Indeed, increasing transistor's frequency has not been possible anymore due to limits of thermal dissipation. Frequency has even been decreasing on some architectures to cut down the energy

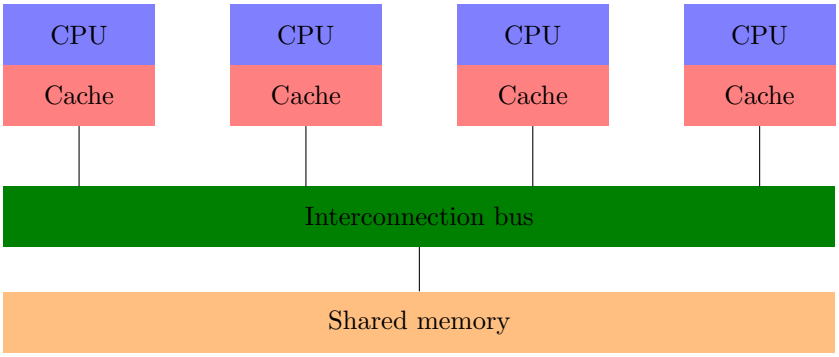


Figure 1.1 – A four CPUs SMP machine.

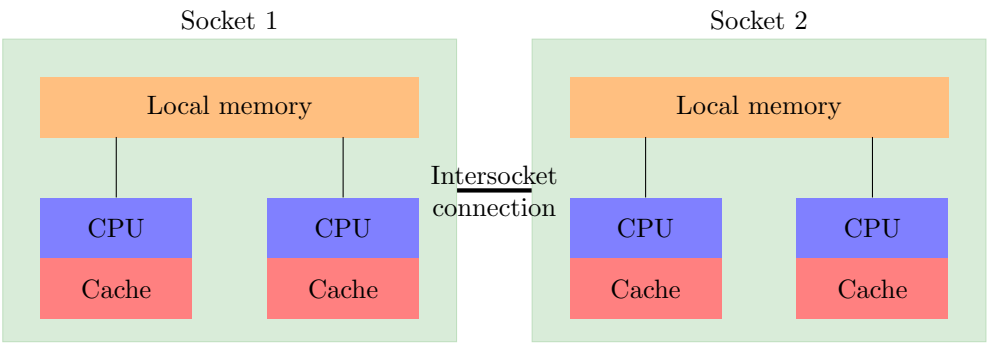


Figure 1.2 – A four CPUs NUMA machine.

consumption and avoid the dissipation issues. Nowadays, the mean frequency of a processor is about 2.8GHz whereas it reached more than 3.5 GHz ten years ago. In order to keep increasing the computational power and follow *Moore's law* [Moo+65] which predict that the number of transistors in our machines double every 18 months, machines manufacturers have multiplied the computing units in our computers. This trend spread to all the market, even our phones have multi-core processors nowadays. Indeed, as the engraving sharpness keeps improving, it is possible to produce chips with more and more computational cores. We can distinguish two classes of multi-core machines.

SMP (Symmetric Multi-Processor, see Figure 1.1) machines offer a symmetric access to the memory, each core is connected equally to the memory. These machines appeared first but are almost nonexistent today. The high number of interconnection buses required limited the number of cores on a chip caused the extinction of SMP architectures. On the contrary, with their simpler design, Hierarchical NUMA (Non-Uniform Memory Access, see Figure 1.2) architectures are more scalable and flood the market. Their memory is distributed among the different processors, connected in a tree-shape structure. Adding cores in this tree is easier than in an SMP machine. With NUMA machines, the complexity is delayed to the developers.

Simultaneously to this evolution of shared memory machines, the cores' complexity

increased to accelerate computations. Today, CPU cores are integrating multiple cores and many transistors. They are capable of computing complex operations. Out-of-order execution also improved the efficiency of the computation at runtime. Modern CPUs integrate vectorial computation units, they are capable of executing the same operation simultaneously on a data vector. This vectorized operations are called SIMD instructions (Single Instruction Multiple Data). These CPUs also integrate several levels of cache memory for an optimal data management.

To get more and more computational power, machines are also connected together and work can be distributed among the different nodes. Such a group of connected computers form a cluster. Usually, a high-speed and low-latency network connects the different nodes and reduces the communication time between two nodes. The different computers in a cluster are also usually identical to avoid an additional complexity in their exploitation.

1.1.2 Accelerators

Using accelerators is not a new concept in the field of scientific computing and application-specific accelerating boards have been used for decades. Yet, these dedicated boards, specially developed for specific applications, had prohibitive cost, and their distribution kept being limited. First accelerators, called ASIC (Application Specific Integrated Circuits) appear around 1980. They were specifically designed to execute a unique task with power-efficiency. In 1985 Xilinx proposed the FPGAs (Field-Programmable Gate Arrays), a flexible alternative to ASICs. They are composed of programmable logic blocks wired together using a circuit description language. Thus, FPGAs are more flexible than ASICs that are not reconfigurable, but they are also less specific and, consequently, less efficient. We can also cite ClearSpead boards which implement widely used operators and were integrating in 2006 in Tsubame Grid Cluster which reached the ninth rank of top500 [Meu+14] ranking list. On this machine, these boards gave a 24% performance increase against a 1% additional power requirement. In any case, this kind of card did not generalize because of their high production cost. Today, the energy required to exploit a large parallel machine using classical cores became prohibitive. Large supercomputers consume much energy (e.g. 17 mega watts for the first machine of June 2014 top500) and having high Flop/s (Floating Point Operations per second) per watt processing units is a requirement. Supercomputers assemblers need both energy efficient and cheap accelerators. [KR12] gives a quick overview of the existing accelerators available, and next paragraphs present the two of them currently available in clusters.

1.1.2.1 Graphical Processing Units

The widely spread GPU cards revealed itself as a good candidate to accelerate computation. Indeed, the large video game community allows for the massive graphic boards production. Graphic cards vendors can sell billions of GPUs and economy of scale is possible. Moreover, these cards are very efficient SIMD vectorial processing units with a large memory bandwidth. They can compute vectorized operations at a very high rate.

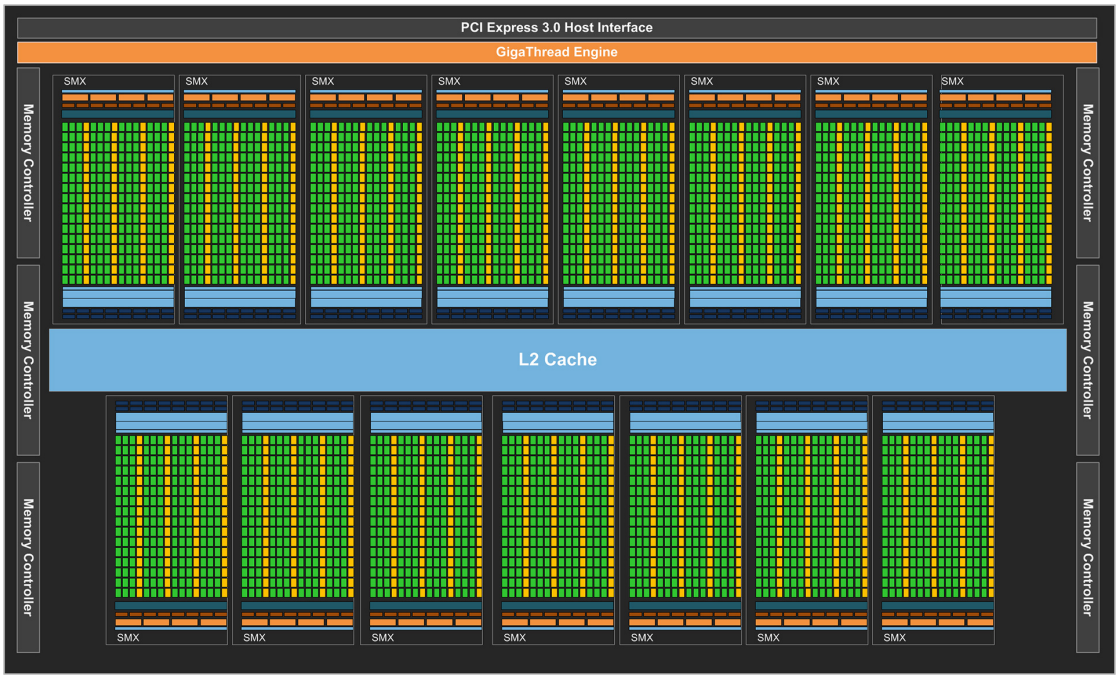


Figure 1.3 – Architecture of the NVIDIA Kepler GK110.

Finally, these cards have a low computational power over energy consumption ratio. Noticing these advantages, few highly skilled developers diverted these graphical cards to accelerate vectorized operations. Considering this new possibility, manufacturers developed specialized chipsets and APIs to target high performance computing; GPGPUs (General Purpose Graphical Processing Unit) emerged. In [Owe+08], Owens et al. describe the characteristics and possibilities of a GPU. Still, GPUs are not suitable to all types of computations and programmers started to select which part of the computation they offload to the GPU.

We call "GPU kernel" a portion of code that is going to be executed on these devices. Each instance of a kernel is run by a thread. Usually, but it can depend on the device, 32 of these threads form a block, also called a warp. An internal scheduler executes each thread within a block simultaneously, and a thread cannot communicate with threads outside the block. Each thread can have a fast access to the shared memory as long as each thread accesses a different bank (of size 16 or 32 depending on the device). A larger global memory is also available to all the blocks but with a higher latency. Thus, developers have to be very careful to manage correctly their memory accesses when writing a GPU kernel.

Figure 1.3 represents the NVIDIA Kepler GK110 with its 15 SMX (next-generation Streaming Processor) units and 1536KB shared L2 cache. Each SMX contains 192 single precision units, 64 double precision units, 32 SFU (Special Function Units) and 32 load/store units. Therefore, a single precision code will run three times faster than

the double version. Each SMX is also equipped with 64KB memory divided into shared memory and L1 cache (48KB/16KB or 16KB/48KB).

AMD also provides GPU dedicated to high performance computing. Today, their performance is comparable to the NVIDIA GPUs, but they have had a hard time gaining market share.

1.1.2.2 Intel Xeon Phi

More recently, Intel designed new accelerators to counter GPGPUs. The Intel Xeon Phi [JR13] also aims at reducing the energy cost of a Flop. This accelerator is less specific than a GPU. Indeed, the Xeon Phi comprises many (61 on last release) simplified classical processors. The cores are based on the well-tried P54C design (Original Pentium design). This design has been simplified to the maximum to reduce their energy cost. Figure 1.4 presents this device architecture. A very high bandwidth bidirectional ring of interconnection is linking these cores together. Each core comes with its private cache memory that is kept coherent using the global-distributed tag directory (TD). The PCIe Client Logic is managing the PCI bus (i.e. the link to the host machine) while the GDDR MC is in charge of controlling accesses to the GDDR5 (Graphic Double Data Rate, specialized memory originally used on graphic cards) memory available on the coprocessor. Each core is capable of executing four threads and has a vector processing unit executing 512 bits SIMD instructions. Developers can either off-load part of the computation to the Intel Xeon Phi, as it is done with a GPU, or execute the whole code on the coprocessor (as if it was a remote server). The later option requires the ability to compile the whole application with a dedicated compiler. To get performance from the Intel Xeon Phi, algorithms must be highly parallel and use efficiently both the 244 threads and the SIMD units.

Fastest machines in the top500 ranking are clusters of distributed NUMA machines equipped with many-core devices (i.e. either GPUs or Intel Xeon Phi coprocessors). The top machine from June 2014 ranking is Tianhe-2 a Chinese cluster. Each of the 16000 nodes is equipped with two classical Intel Xeon cores, and three Intel Xeon Phi coprocessors. Figure 1.5 presents a smaller configuration, one Mirage node from PlaFRIM experimental platform in Bordeaux, which has been used in the experiment presented in chapter 2. It contains two hexa-core processors: one connected to a GPU, and the other to a pair of GPUs. Several nodes of this kind are then interconnected using a high performance network, adding an additional level to the hierarchical layout. The current challenge for developers is to use efficiently all the computational power offered by these complex architectures.

1.2 Addressing parallel machines

As we have seen before, to reach higher and higher computational power, machines had to integrate more and more complex hardware. It became quite hard to reach the peak performance of such a machine. To illustrate this one can have a look at the top500

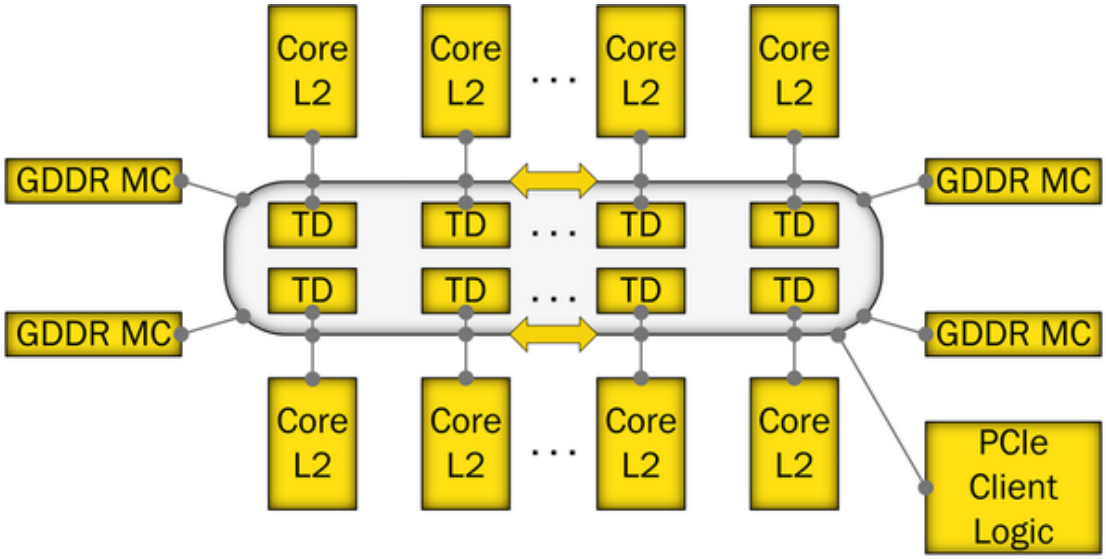


Figure 1.4 – Architecture of the Intel Xeon Phi.

ranking. Indeed, on this list we can notice the difference between the theoretical peak performance and the greatest performance obtained running the LINPACK test suite. Depending on the architecture, approaching the peak performance is harder. A solution to ease the development on such architectures is thus required. In this section, we present the different frameworks that we can use to exploit efficiently the complex architecture provided by machine manufacturers. First, we introduce the techniques available to get an efficient parallel code on a classical multi-processor machine. Then, we specifically detail solutions to implement high performance algorithms for modern accelerators.

1.2.1 Parallel programming

Inside a node, several solutions were developed to implement shared memory algorithms. The solutions are identical for SMP and NUMA nodes. While high level shared memory solutions can handle NUMA effects through an intelligent runtime system [Bro+09], in the general case, the developers have to adapt its algorithm. To target shared memory architectures, operating systems first proposed inter-processes memory sharing solutions. Today, they integrate lightweight processes (threads) which can be created from a master process, and share access to its memory. For example, POSIX operating systems provide P-Threads (POSIX Threads) that developers can use to handle threads accurately. With this solution, the developer has to explicitly start threads from the main process with a given task to execute. Other solutions provide simpler and higher level ways to express parallelism, hiding the underlying thread library to the developer. The most common is OpenMP [Cha01], a language extension, available for both Fortran and C, that handles shared memory parallelism using threads. OpenMP proposes to annotate the code with *pragma* keywords. The simplest way to use OpenMP is to specify which

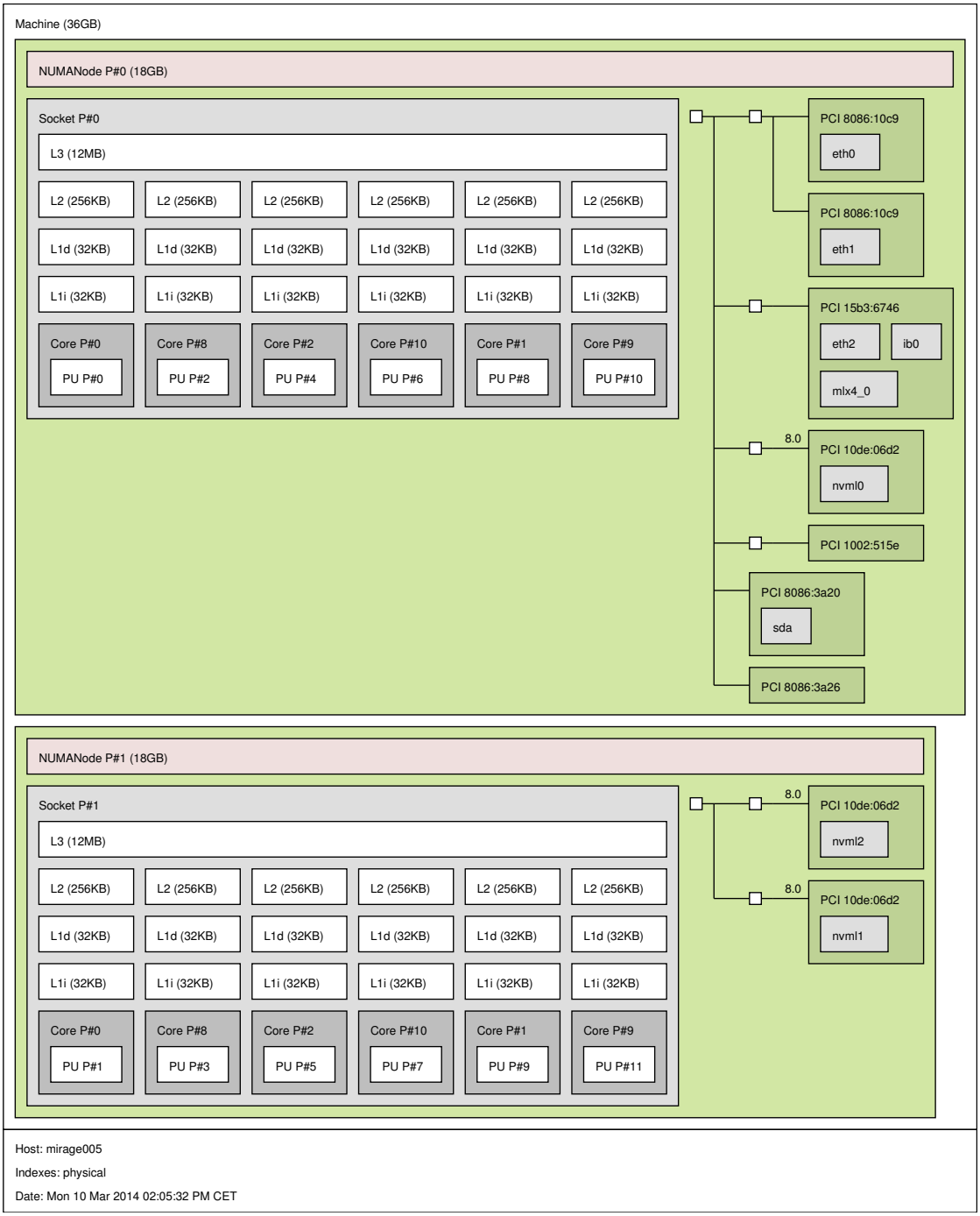


Figure 1.5 – A Mirage node described using Hardware Locality (hwloc) tool.

"for" loops can be executed in parallel so that the runtime can execute iterations simultaneously. OpenMP also proposes the possibility to split a program into independent tasks for a more advanced usage. Intel also proposes the Intel TBB (Thread Building Blocks) library [Rei07], in C++, that provides similar tools such as `parallel_for`, `parallel_while`, and `parallel_reduce` functions, or a task-based programming API. Cilk [FLR98] proposes to extend C language to extract "fork-join" parallelism from the algorithm. The user specifies which routines are *Cilk* functions and can be executed in parallel. These selected functions can then be *spawned* and executed by parallel threads. To ensure the ending of the *spawned* functions, user has to call a *Cilk* synchronization routine. All these solutions can reach high performance if the developer describes accurately enough the parallelism of its algorithm.

To address distributed machines, the message passing paradigm is mainly used. This paradigm proposes to exchange data between different instances of one program using messages that are explicitly sent by an instance and received by another. The developer has to describe precisely the data distribution, and the way data are exchanged. One specific standard, MPI [GLS99] (Message Passing Interface), emerged as the elected one to handle parallel clusters. Many implementations of this standard were developed. Only MPICH [Gro02] and OpenMPI [Gab+04] survived and are used by manufacturers to develop their own libraries such as Intel MPI, Bull MPI, etc. Languages extensions such as Co-Array Fortran [NR98], HPF [Ric96] (High Performance Fortran), UPC [Car+99] (Unified Parallel C), and more were also developed to address distributed machines. Each language extension provides a higher abstraction of the algorithm, but MPI, which let the programmer decide exactly how its application will behave, is the most used parallel library. All these solutions can also be applied to shared memory systems (which is simpler as all processes can reach all data). Although such a solution can use optimizations to exploit the direct memory access possibilities, the message passing paradigm implies data duplication that could be avoided in a shared memory context. Indeed, threads would share the same data. However, using memory sharing and threads requires protection locks that can slow down the computation. The developer has to minimize the size of lock protected area and find a tradeoff between memory sharing and distributed implementation corresponding to its application.

In a distributed code, communications can consume a lot of time. In order to get more efficiency, the developer has to overlap the communications with computations. In a more general case, one should avoid synchronization to get performance. Indeed, when the number of computational unit increases, synchronizing the whole machine can be prohibitive. Thus, the commonly used *fork-join* parallelization scheme where sequential section and parallel section are interleaved, implying synchronization in-between, and must be avoided.

1.2.2 Addressing accelerators

To off-load part of a computation to an accelerator, one would have to write a specialized kernel for the targeted architecture. The developer can either write a whole code that will be executed on the accelerator, or offload only part of the computation to the device. The

second option appeared to be more efficient in many cases as the accelerators, particularly the GPUs, but also the Intel Xeon Phi, are not suited for all kinds of operations. Thus, the developer has to send the data to the accelerator, execute the kernel and retrieve the data from the device. The memory transfer cost can be very expensive and, as in a distributed code, the developer has to overlap those communications with computations.

To help in the design of applications adapted to these complex architectures, frameworks have been developed. NVIDIA proposes CUDA [NVI11] and several organizations (Apple, AMD, Intel, and NVIDIA) promote OpenCL [SGS10] to develop the kernels, handle the memory transfers, and execute the kernels on the device. Intel encourages using OpenMP [Cra+12] to produce parallel code to target its Intel Xeon Phi. While CUDA is specific to NVIDIA GPUs, OpenCL aims at providing a portable solution to accelerators and multi-core management. In reality, if OpenCL provides portable code, most of the time the performance is not portable, and the kernels have to be redesigned to target a new architecture. The main challenges raised by these heterogeneous platforms are mostly related to task granularity and data management: although regular cores require fine granularity of data as well as computations, accelerators such as GPUs need coarse-grain tasks. This unavoidably introduces the need for identifying the parts of the algorithm that are more suitable to be processed by each kind of architecture. Using one of the presented frameworks, one can execute part of its code on an accelerator and keep the other part on the classical CPU(s). The drawback of this approach is that, except if a very complex code is written to handle it, the CPUs are idle while the GPU computes, and vice versa. A need for more flexibility is required to use the maximum power from heterogeneous nodes.

This, coupled to the solutions for shared and distributed architectures, is a large burden to the developers who want to benefit from all the capabilities of the clusters. An additional layer, which would insulate the algorithms and their developers from the rapid hardware changes, is required. This portability layer recently reappeared as under the denomination of task-based runtime. Those runtime systems are presented in next subsection.

1.2.3 Task-based runtime systems

Many initiatives have emerged in previous years to develop efficient runtime systems for modern heterogeneous platforms. Task-based runtime systems propose to describe the algorithms as tasks with data dependencies in-between. The underlying runtime systems manage the tasks dynamically and schedules them on all available resources. The algorithm can then be represented as a DAG (Directed Acyclic Graph) where vertices are representing the tasks and edges their dependencies. These runtime systems can be generic, as StarPU [Aug+11; Aug11] and PaRSEC [Bos+12], or more specialized as QUARK [Yar12]. Without going into details, the main differences among these different runtime systems reside in their representation of the graph of tasks, their management of data movements between computational resources, the extent they focus on task scheduling, and their capabilities to handle distributed heterogeneous platforms.

1.2.3.1 Generic runtime systems

Many different task-based runtime systems were recently developed to help developers produce efficient algorithm on heterogeneous architectures. Among them, Qilin [LHK09] provides an interface to submit kernels that operate on arrays that are automatically dispatched among the different processing units of a heterogeneous machine. Moreover, Qilin dynamically compiles parallel code for both CPUs (by relying on the Intel TBB technology), and for GPUs using CUDA. Another relevant framework is Charm++ [KK93] which is a C++-based parallel programming system 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 [KK11]. Runtime systems like KAAPI/XKAAPI [Her+10] or APC+ [HSÇ12] also offer support for hybrid platforms mixing CPUs and GPUs. Their data management is based on a DSM-like mechanism: each datum block is associated with a bitmap that permits the determination of whether there is already a local copy available to a specific processing unit or not. The StarSs project is actually an umbrella term that describes both the StarSs language extensions, and a collection of runtime systems [Pla+09] targeting different types of platforms [Ayg+09; Bad+09; Bel+09]. StarSs provides an annotation-based language that extends C or Fortran applications to offload pieces of computation on the architecture targeted by the underlying runtime system. With StarPU [Aug+11], the user uses a generic sequential task-based programming model to insert tasks which will be executed in parallel by the different workers (i.e. computing units). The scheduler that will execute these tasks can be either chosen among the StarPU predefined ones or provided by the user as a plugin. Finally, the PaRSEC [Bos+12] (formerly DAGuE) runtime system dynamically schedules tasks within a node using a rather simple strategy based on work-stealing and following an initial data distribution. It was first introduced for dense linear algebra, but was later extended to more generic applications (e.g. in the high order finite element library Aerosol [Mbe+13]). It takes advantage of the specific shape of the task graphs (in the sense that there are few types of tasks) to represent the task dependency graph in an algebraic fashion as expressed in [CJ99]. Following this idea, the OpenMP consortium, supported by Intel, is developing OpenMP 4 which integrates the concept of interdependent tasks extending the task concept introduced in OpenMP 3.

1.2.3.2 Runtime systems specifically designed for linear algebra

Some other runtime systems are specifically designed for dense linear algebra. For example, the TBLAS runtime system [SYD09] 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 provide a static mapping of the data on the different processing units, but it supports heterogeneous data block sizes (i.e., different granularity of computations). The QUARK runtime system [KD09] was specifically designed for scheduling linear algebra kernels on multi-core architectures and is used in the PLASMA project [Agu+09]. It is characterized by a scheduling al-

gorithm based on work-stealing and by its higher scalability in comparison with other dedicated runtime systems. Finally, the SuperMatrix runtime system [Cha+08], used by FLAME [Igu+12] library, follows nearly the same idea as it represents the matrix hierarchically: the matrix is viewed as blocks that serve as units of data where operations over those blocks are treated as units of computation. The implementation transparently enqueues the required operations, internally tracks dependencies, and then executes the operations using out-of-order execution techniques.

1.3 Sparse linear algebra methods

The scientific community has developed multiple methods to efficiently solve sparse linear systems of the form $Ax = b$, where A is a sparse matrix of size n. A is said to be a sparse matrix if it contains only few non zero terms. This low fill-in is owed to either nonexistent or neglected interactions in the mathematical model used by the simulation. Indeed, coefficients of the matrix usually represent the interactions between two points resulting from the discretization of the studied problem. Frequently, interactions between two remote points can be neglected and zero terms appear in the matrix. x and b are two n sized vectors. x is the unknown of the system to be solved and b is called the right-hand-side. x can then be obtained with the formula $b = A^{-1}x$.

To illustrate this, we can consider the air flow around a plane. One can consider that the characteristics of the air flow at the end of the left-wing would have only little influence on the flow on the right-wing. The corresponding term in the matrix would then be null. If we keep following the air flow simulation example, x would be the air flow speed at time $t + 1$, computed from b, the air flow speed at time t.

However, the computation of the matrix A^{-1} is generally avoided. Indeed, we have no a priori knowledge of the structure of A^{-1}, and it might require being stored in an expensive dense matrix. This computation would require too much memory and computational time. The scientific community developed many other methods to obtain the solution x. These methods are adapted to the characteristics and the constraints of the simulation code. Indeed, the need for accuracy, the structural and numerical stability of the matrix at each time step, and its numerical complexity depend on the studied problem and the method used for the simulation.

Two main classes can be distinguished to solve sparse linear systems: direct methods and iterative methods. These sparse linear algebra methods rely on the BLAS (Basic Linear Algebra Subroutine) libraries that we will present in next section. Then, we will exhibit the advantages and drawbacks of the existing methods. After that, we will focus on direct methods that are the main topics of this thesis. We will explain the different steps involved in a direct sparse linear solver. Finally, we will compare the multiple implementations available and show how they started addressing heterogeneous machines.

1.3.1 Prerequisite: Basic Linear Algebra Subroutines

Independently of the method, if the computational intensity is not too low, operations have to be gathered into dense blocks to be efficient. For example, in MATLAB (a widely used numerical computing environment), a BLAS-based method is used to solve $Ax = b$ when the ratio between the number of operations and the number of entries in the matrix is smaller than 40. Otherwise, a scalar method is used. Indeed, dense matrix-matrix operations can be computed very efficiently on computing unit whereas memory accesses are limiting the efficiency of sparse operations. These operations have to be adapted to the structure of the computing unit to use its different level of cache memory efficiently. The BLAS [Law+79] API became the standard to handle these operations. The API defines three classes of operation:

- BLAS1: for scalar, vector and vector-by-vector operations ($O(n)$ operations for $O(n)$ data);
- BLAS2: for matrix-by-vector operations ($O(n^2)$ operations for $O(n^2)$ data);
- BLAS3: for matrix-by-matrix operations ($O(n^3)$ operations for $O(n^2)$ data).

As they involve more computations, the BLAS3 operations are the ones that benefit most from a precise tuning of the algorithm. Among the multiple implementation of the BLAS API, we can cite several implementations.

refblas [Don+88; Don+90] is the reference implementation from netlib. This implementation must be avoided to obtain performance.

ATLAS [WD97; WPD00] (Automatically Tuned Linear Algebra Software) is an open source implementation that relies on automatic tuning, and thus available for all CPU architectures.

MKL [Int] (Math Kernel Library) is the Intel implementation mainly dedicated to Intel processors.

ACML [AMD] (AMD Core Math Library) is the AMD implementation of the BLAS API.

ESSL [KLV98] (Engineering and Scientific Software Library) is IBM's implementation.

OpenBLAS [GG08] (Formerly GotoBLAS) originally developed by Kazushige Goto relies on TLB (Translation Lookaside Buffer) misses optimization instead of cache misses.

These implementations are finely tuned to fit perfectly the characteristics of the machine they are executed on and get the maximum performance from the chipset [Edd10]. To address SMP and NUMA machines, multi-threaded versions of these libraries were developed such as for ACML, MKL or OpenBLAS [Dia+08]. Distributed versions of

these libraries are also available. For example PBLAS [Cho+96] provides a distributed implementation of the BLAS2 and BLAS3 routines. LAPACK [And+99], and ScaLAPACK [Bla+96], on distributed memory clusters, use the classical BLAS routines and provide higher level routines such as dense linear decomposition, eigenvalues problems, etc.

1.3.2 Solving a sparse linear system

Direct methods [Dav06] consist in the decomposition of the matrix A in a product, LU, of two triangular matrices. The first one, L, is lower triangular where the other one, U, is upper triangular. Once it has been transformed into $LUx = b$, the system can then be simply decomposed in two triangular systems easy to solve. If the matrix has specific properties, the problem can then be further simplified, often in term of computational cost. For example, in the symmetric positive definite (SPD) case, the Cholesky factorization gives $A = LL^T$. If A is only symmetric, it is still possible to achieve a Cholesky-Crout decomposition: $A = LDL^T$, where D is a diagonal matrix. These last two decomposition methods reduce by half the computational and memory costs. Using direct methods, one can get a very accurate solution of the $Ax = b$ system. Moreover, once the decomposition is computed, it can be used to solve several systems where only the right-hand-side would differ. The drawback of these methods is that they require a huge amount of memory, which can be a bottleneck preventing the increase of the solved system size. Indeed, during the decomposition, new non-zero terms, called fill-in, appear in the matrix, increasing the memory requirement. This phenomenon is more detailed in subsection 1.3.3. A second bottleneck is the complexity of the decomposition algorithm, which is very large: $\mathrm{O}(n^3)$ operations for a dense factorization, $\mathrm{O}(n^{\frac{3}{2}})$ for a 2D sparse decomposition and $\mathrm{O}(n^2)$ for a 3D sparse factorization.

Iterative methods are a cheaper alternative to the direct methods, both in terms of memory and complexity. They solve the equation $Ax = b$ from an initial guess x_0 that is iteratively improved at each iteration until the inquired precision is reached. In [Saa96], Youssef Saad presents a large diversity of iterative methods (e.g. GMRES, Conjugate Gradient, etc.). These methods are cheap both in terms of memory and computational time. They are generally based on the matrix-vector product that can be computed in $\mathrm{O}(nnz)$, where nnz is the number of non-zero terms in the matrix. To reach the precision of a direct method, a large number of iterations (or a very accurate initial guess) would be required. However, in many simulations a fine accuracy is not required, and iterative methods can be a good candidate. Depending on the complexity of the studied problem, an iterative method can be unable to converge to the solution. Indeed, the behavior of iterative methods is specific to the problem they are trying to solve. The convergence of these algorithms also relies on a good choice of the initial guess x_0.

For solving 3D problems, where the matrix A is quite dense, iterative methods are generally more interesting because of their low computational cost. On the contrary, 2D problems lead to rather sparse matrices that will not fill-in too much during the decomposition. Direct methods are then well suited for these problems.

Transitional solutions were proposed to exploit the best of the two methods. The

first possibility is to approximate the decomposition of the matrix A. For instance, it is possible to suppress some values considered negligible or to ignore the *fill-in* terms that appears during the decomposition. Those methods are called incomplete LU decompositions. The selection of the terms to remove is either based in value threshold (ILU(t) methods [Saa94]), or on their level of interconnection (ILU(k) methods [Man80]). This level is defined as follows: level zero comprises initial terms and level k ($k \geq 1$) terms are fill-in terms generated by the updates involving level $k-1$ terms. With these methods, we get an approximation of A's decomposition that is used as a preconditioner P for an iterative method. Then, for a left preconditioning, one solves $P^{-1}Ax = P^{-1}b$ which leads to a better convergence. One can also use a residual factorization as a preconditioner. This is the method used by the JOREK application studied in this thesis and presented in chapter 4.

To reach larger problems while keeping a high accuracy, one can use domain decomposition methods. These algorithms split the problem into sub-domains, solve independently the problem with a direct method on the interior of each sub-domain while reporting the contribution to the Schur complement (i.e. the local interface), and, finally, solves iteratively a global system on the interfaces. HIPS [GH08], MaPHyS [GH09], PDSLin [Yam12], and ShyLU [RBH12] implement these domain decomposition methods in different ways. PDSLin and ShyLU use a twofold approach to compute the preconditioner. First, an approximation to the global Schur complement is computed. Then, this approximation is factorized to form the preconditioner for the Schur complement system that does not need to be formed explicitly. HIPS builds its preconditioner using an Incomplete LU factorization while MaPHyS computes an additive Schwartz preconditioner. All these methods try to benefit both from the speed and good scalability of the iterative methods, and from the accuracy and robustness of direct method.

Finally, multi-grid methods [Hac85] speed-up the performance of an iterative method by correcting, from time to time, the solution. The linear system is projected onto a coarse mesh, a direct method solves the coarse problem, and then the correction is reported to the finer original mesh. The multi-grid methods can be separated into two sub-classes: geometric [Hül+06], where the multilevel hierarchy is tightly linked to the mesh geometry and the partial differential equations; and algebraic [Bra86], where the solver uses only the matrix and can be used as a black box.

Thus, various methods have been developed to solve $Ax = b$ equation. The choice of the method depends on the characteristics and the needs of the simulation. Direct methods are the most robust, can be unavoidable in certain cases, and is an essential component to many other solutions. In our case, we will focus on high accuracy of direct methods which is a requirement for many simulation codes such as JOREK. We will now describe more precisely these methods and the techniques that improve their efficiency.

1.3.3 Description of the direct methods

Direct methods are the central subject of this thesis. These methods are the most expensive both in term of memory and computational power. However, as we have

explained, they are widely used because they can reach a very high accuracy. They can also be used as a preconditioner for hybrid methods, mixing direct and iterative methods to obtain a tradeoff among speed, accuracy, and memory consumption. The possibility to save and reuse the decomposition can also reduce the computational cost by computing the factorization only once to solve multiple linear systems sharing the same matrix A. We will see later how we can control the memory consumption of direct solvers. We will also describe the algorithms, and the parallelization techniques used to obtain efficient computation. Four main steps compose the direct methods:

1. The reordering of the unknowns can reduce the *fill-in* that appears during the decomposition.
2. The symbolic factorization that predicts the structure of the factorized matrix. Analysis of this symbolic factorization predicts the amount of memory, and the number of operations required to perform the decomposition.
3. The numerical factorization that computes the decomposition of A into $L \times U$. It is the most expensive step.
4. The forward/backward substitutions that compute the solution x of the system using the factorized matrix and the right hand side b.

1.3.3.1 Reordering of the unknowns

For the sake of simplicity, we consider here the LL^T decomposition, when the matrix is SPD. The algorithms are easily adaptable to obtain an LDL^T decomposition by modifying the computation on the diagonal. In a general case, to obtain an LU factorization, one can consider A', the matrix obtained by completing A with zeros such that the non-zero pattern of A' is symmetric (i.e. $pattern(A') = pattern(A + A^T)$). Once this matrix is constructed, the LU factorization can be obtained by performing the symmetric operations on the upper part of the matrix.

During the decomposition of the matrix A into LL^T, *fill-in* [GL81] terms will appear. Indeed, after a Cholesky decomposition, the matrix L hold more terms than the matrix A. A crucial tool to study this fill-in is the *graph model* [GL81] associated to the Gaussian elimination. This fill-in is directly linked to the ordering of the unknowns. Therefore, we are looking for an ordering of the unknowns, that is to say an *ordering of the initial matrix associated graph vertices* [ADD96; GL81; PR97; PRA99] that minimizes the fill-in, and simultaneously, the number of operations required to perform the decomposition. Figure 1.6 illustrates the fill-in related problems. In the case presented here, fill-in terms, in red, can be suppressed using a good reordering (Figure 1.6(b)). This minimization (or eventually, suppression in the given example) of the fill-in reduces both the storage size of the L matrix, and the number of operations to perform. Indeed, only non-zeros values of LL^T are stored and require computations.

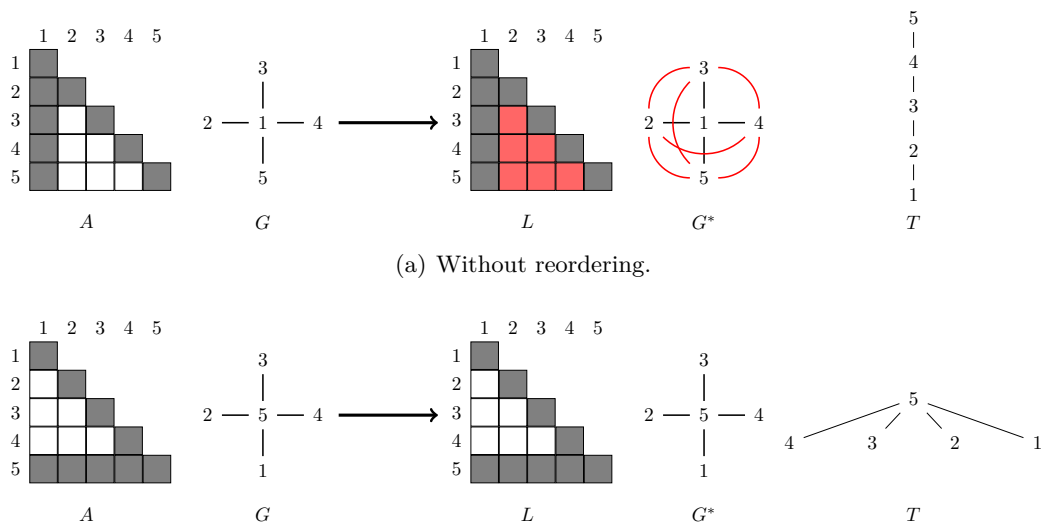
The undirected graph G associated with a symmetric (or structurally symmetric) n by n matrix A is a graph with n vertices where there is an edge (i, j) between the vertices i and j if, and only if, a_{ij} is not null.

$$\exists (i,j) \in G \Leftrightarrow a_{ij} \neq 0$$

The elimination graph is the graph G^* associated to the lower triangular matrix L resulting from the factorization. Naturally, G^* has the same number of vertices as G but (many) more edges. Indeed, it contains the fill-in edges (in red on Figure 1.6(a)). The fill-in of the matrix can be computed using the following theorem:

Theorem 1 (Characterization Theorem [RTL76])

$$(i,j) \in G^* \Leftrightarrow \begin{cases} (i,j) \in G \\ or \\ \exists \ a \ path \ (j, k_1, \ldots, k_l, i) \ such \ that \ \forall p \in [\![1, l]\!], k_p < min(i,j) \end{cases}$$



(a) Without reordering.

(b) With reordering.

Figure 1.6 – Advantage of reordering the unknowns for sparse matrix decomposition. Blank squares represent zeros, greys are initial non-zeros and red squares and edges the fill-in.

The *Elimination tree* T [Liu90] associated to the factorized matrix is a tree (in the classical meaning of the graph theory, in the case of reducible matrix it is a forest) with n vertices. There is an edge between the vertices i and j if, and only if, the row of the first non null off-diagonal term in column j in the factorized matrix is i.

The elimination tree is crucial for sparse direct solver parallelization because it describes the dependencies among the computations: if two vertices are on different branches of the elimination tree, then the corresponding unknowns can be eliminated independently in parallel.

In a parallel context, the reordering of the vertices of G should then both *minimize the fill-in*, and *maximize the independence of the computations during the factorization*, that is to say leading to *large and low height* elimination trees. Following these criteria, the efficient reordering are based on *minimum degree* [ADD96; TW67] and *nested dissection* [GL81]. One can notice that the classical Cuthill-McKee [CM69] reordering must be discarded because it creates more fill-in, and produces very high elimination trees with little independence among the computations. Figure 1.6 shows that a good reordering can both decrease fill-in and exhibit more parallelism.

1.3.3.2 Symbolic factorization

The following step is the *blocked symbolic factorization* [CR89]. It computes the blocked sparse structure of the matrix L from A's one. The goal of the blocked structure is to optimize the *numerical factorization* step by creating large blocks on which level 3 BLAS operations can be performed. This block structure forms a partition P of the unknowns where each part, called supernode, can be considered separately. In the following, supernodes will also be called column-blocks or panel. As we can see in figure 1.7, terms are created only to obtain this efficient block structure (e.g. $L_{9,1}$ is only present because of the blocking, there is no edge in G between 1 and 9) and might induce more fill-in. A compromise has to be found between memory used and BLAS efficiency. The prediction of the structure will free us from the effective fill-in terms management during computation. Indeed, structural changes during factorization would heavily reduce the efficiency of the *numerical factorization* step that uses a memory bound algorithm.

The complexity [CR89] in time and in space of this algorithm increases with the total number of off-diagonal blocks in the structure built for L (see figure 1.7). The number of off-diagonal blocks, and consequently the complexity in time, depends on the reordering quality regarding A's sparsity conservation, and on the P partition. This complexity is always far below numerical factorization's one, which justifies its usage. The elimination tree is then a *blocked elimination tree* (tree T in figure 1.7) and its structure describes the dependencies among the solver's blocked computations.

The blocked elimination tree is a covering tree of the graph of the exchanges between column-blocks (see figure 1.7) during blocked factorization algorithm (see section 2.1.1 page 39). This graph is the quotient graph G^*/P symbolically computed in the form $(G/P)^*$.

In a parallel context, this step is necessary to compute the distribution of the data onto the nodes during preprocessing.

1.3.3.3 Factorization methods

With the block structure of A, obtained during previous step using the elimination tree, we can generalize the sequential scalar algorithm in a more efficient algorithm working directly on the blocks. Using blocks (or column-blocks) of the matrix we can benefit from the heavily optimized dense linear algebra routines.

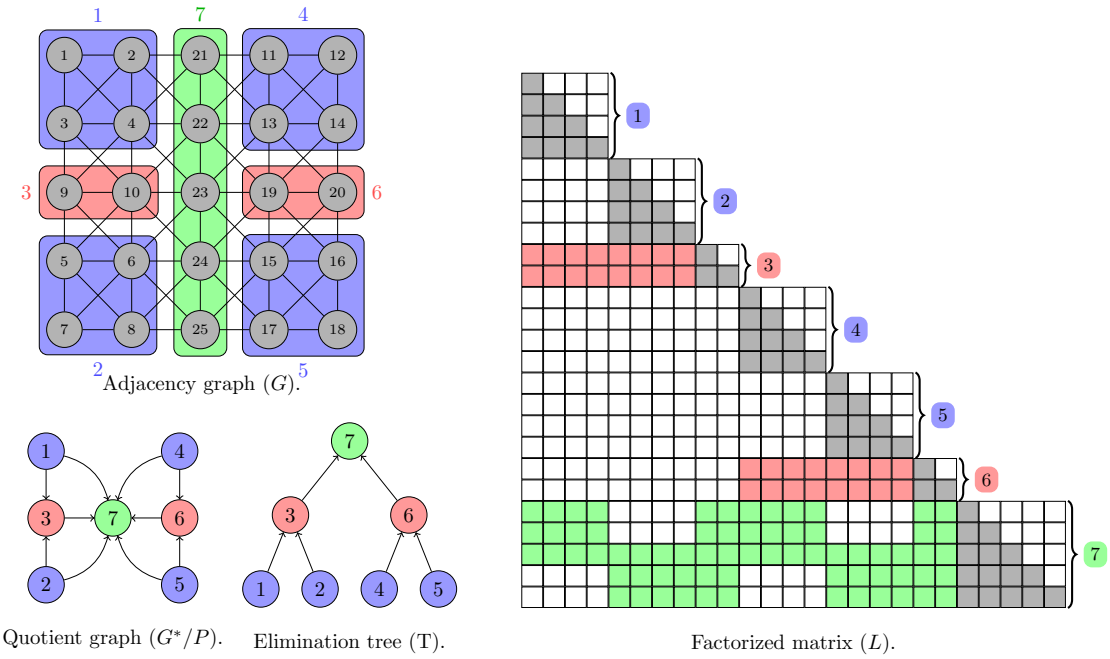


Figure 1.7 – An example of column-blocked structured factorized matrix L, with the graph G associated to the matrix A, the quotient graph G^*/P and the blocked elimination tree associated T. Each of the seven column-blocks comprises a dense diagonal block (in blue) and several dense off-diagonal blocks (in red and green).

For the sake of memory consumption reduction, the matrix A is actually stored in the structure that will store the decomposition during the algorithm execution. The decomposition is then performed *in place*.

Figure 1.8 presents the notations used in the algorithms presented in this thesis. For each column-block k, $1 \leq k \leq N$,

- $A_{k,k}$ is the dense diagonal block,
- b_k is the number of off-diagonal blocks in column-block k,
- $A_{(j),k}$ is the jth off-diagonal dense block with $1 \leq j \leq b_k$, (j) being a multi-index describing the row interval (on the Figure $j = 1$);
- $A_{(1-b_k),k}$ represent all the off-diagonal dense blocks of column block k, from block 1 to b_k.

Furthermore, $A_{(i),(j)}$ is the rectangular dense block corresponding to the rows of the multi-index (i) and to the columns of the multi-index (j). $A_{(j),(j)}$ corresponds to the same notion, but for a sub-block of the diagonal block $A_{l,l}$ facing the off-diagonal block $A_{(j),k}$.

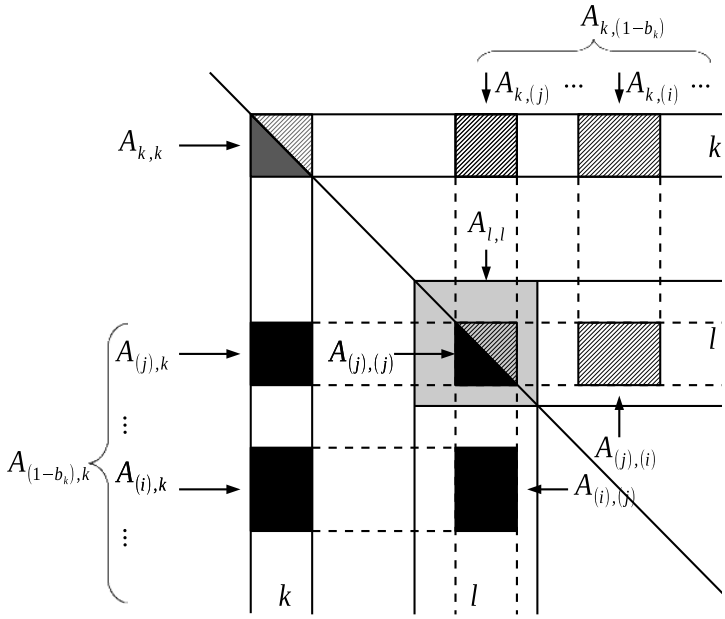


Figure 1.8 – Description of the notations used for the factorization algorithm.

Two methods have been developed and can both be parallelized to perform the numerical factorization. The first one is called the multifrontal method, while the other is called the supernodal method. These two techniques differ only in the way they apply contributions from one block to another. Those differences are also found in the parallel versions of these algorithms.

Multifrontal methods: With a multifrontal [Ame+01; ADL98; GKK97a; GKK97b] approach, each column-block is associated with a *frontal* matrix, also called Schur complement, that holds all the contributions from the corresponding sub-tree of the elimination tree to its father (see figure 1.9). Algorithm 1 presents the sequential multifrontal *LU* algorithm. For each block column, the column block associated matrix is allocated and assembled using initial values and the sons contributions. The diagonal-block is factorized, the off-diagonal system is solved and the frontal matrix is updated. This frontal matrix is then passed to the column-block's father in the elimination tree. The latter adds up all the frontal matrices he receives with its own Schur complement before sending the result to its father and so on. The presented algorithm is easily adaptable to the symmetric case by substituting references to $U_{i,j}$ for $L_{i,j}^T$ (i.e. $L_{j,i}$), removing the second solving operation, and performing only the lower part of the frontal matrix update. With these methods, we try to express most of the parallelism, not only by the frontal matrices dense block cutting and computations, but also by reordering methods that maximize the elimination tree width. A description of the uni-dimensional or bi-

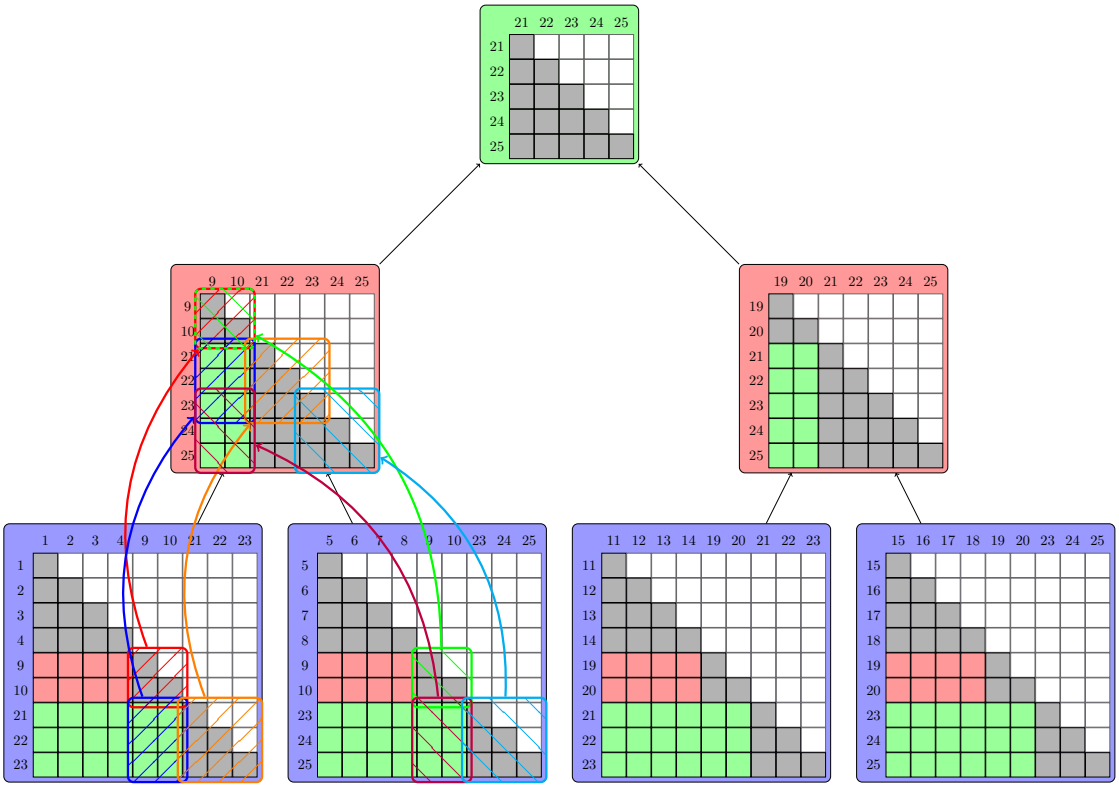


Figure 1.9 – Multifrontal algorithm applied on the matrix from Figure 1.7. Level 2 frontal matrix is updated using the Schur complement of level 3 frontal matrices (represented with boxes and bent arrows for the most left bottom frontal matrices). And level 1 is updated by its two sons from level 2.

dimensional parallelization of this approach can be found in [Ame+01]. This method is particularly well suited for a parallel and out-of-core implementation (i.e. with unused data stored on disk to save memory until it will be reused). Indeed, the factorization can progress level by level in the tree, without any requirement on data from the sons or the father nodes. All these data can be saved to disk before and after they are eliminated. Moreover, nodes at the same level of the tree can be eliminated independently providing parallelism. The dense operation on the Schur complement update also allows a compute intensive BLAS3 operation that is efficient. The drawback of this method is the memory required for the storage of the frontal matrices which grows while going up into the elimination tree and can become large.

Algorithm 1 Multi-frontal factorization: $A = LU$.

1: **For** $k = 1$ to N **Do**
▷ /* Assemble the matrix using frontal matrices from sons. */
2: Assemble the frontal matrix associated to $A_{k,k}$
▷ /* Factorize diagonal block */
3: Factorize $A_{k,k}$ in $L_{k,k}.U_{k,k}$
▷ /* Solve off-diagonal systems */
4: Solve $L_{(1-b_k),k}.U_{k,k} = A_{(1-b_k),k}$
5: Solve $L_{k,k}.U_{k,(1-b_k)} = A_{k,(1-b_k)}$
▷ /* Update the Schur complement. */
6: $A_{(1-b_k),(1-b_k)} = A_{(1-b_k),(1-b_k)} - L_{(1-b_k),k}.U_{k,(1-b_k)}$
7: **End For**

Algorithm 2 Right looking blocked sequential factorization: $A = LU$.

1: **For** $k = 1$ to N **Do**
▷ /* Factorize the column block */
2: Factorize $A_{k,k}$ in $L_{k,k}.U_{k,k}$
3: Solve $L_{(1-b_k),k}.U_{k,k} = A_{(1-b_k),k}$
4: Solve $L_{k,k}.U_{k,(1-b_k)} = A_{k,(1-b_k)}$
▷ /* Trailling supernodes updates */
5: **For** $j = 1$ to b_k **Do**
6: **For** $i = 1$ to b_k **Do**
7: $A_{(i),(j)} = A_{(i),(j)} - L_{(i),k}.U_{k,(j)}$
8: **End For**
9: **End For**
10: **End For**

Supernodal methods: Supernodal methods do not store contributions but apply them to the targeted panel (or column block) as soon as they are produced. The global storage of the data is then reduced, but all column-blocks (also named supernodes or

Algorithm 3 Left looking blocked sequential factorization: $A = LU$.

1: **For** $k = 1$ to N **Do**
▷ /* Apply updates from sons in the elimination tree */
2: **For** each $L_{k,(j)}$ facing $A_{k,k}$ **Do**
3: **For** each $L_{(i),(j)}$ below $L_{k,(j)}$ **Do**
4: $A_{(i),k} = A_{i,(k)} - L_{k,(j)}.U_{(j),k}$
5: **End For**
6: **End For**
▷ /* Factorize the column block */
7: Factorize $A_{k,k}$ in $L_{k,k}.U_{k,k}$
8: **For** $i = 1$ to b_k **Do**
9: Solve $L_{(i),k}.U_{k,k} = A_{(i),k}$
10: Solve $L_{k,k}.U_{k,(i)} = A_{k,(i)}$
11: **End For**
12: **End For**

panels) from the upper part of the tree have to be allocated to apply contributions on the destination column block. Within supernodal methods, one can distinguish two sub-classes that differ in the way the contributions are taken into account. The *right-looking* approach, presented in algorithm 2, and the *left-looking*, presented in 3. The right-looking algorithm factorizes one column-block and then applies, once for all, the contributions from the computed column-block onto the column-blocks *on the right*. The left-looking approach switches the contribution adding and factorization phases. Contributions *from the left* of current column-block are accumulated and then the panel is factorized. It will later be read again to add its contributions. One can consider that the right looking privileges the read data, while left looking focus on the written data. As for multifrontal methods, these two algorithms can be adapted to the symmetric case by substituting $U_{i,j}$ with $L^T_{i,j} = L_{j,i}$ and removing operations on the upper matrix. Contrary to multifrontal method, the trailing submatrix update cannot be gathered in one large matrix-matrix multiplication as the data are only contiguous in a column-block. Thus, the operation is less compute intensive than with multifrontal methods.The advantage of this method is that it only allocates the structure of the factorized matrix, which can save a lot of memory. A tradeoff between performance and memory consumption can be found by grouping updates by column-block and scattering the result onto the targeted panel. These operations can then be executed in parallel as they are updating different panels.

Once these two sub-methods have been explicated, one can differentiate two different parallel implementations of the supernodal method: *fan-in* or *fan-out* [Ash93; NP93; AEL90]. The fan-in method, consist in compressing the exchanged data. All contributions computed on one processor and addressing the same column-block are added in a temporary *phantom* column block, called fan-in buffer. As for frontal matrices in multifrontal methods this buffer covers all its contributions. The fan-out approach consists in

sending the column block once it has been computed so that the destination processor can directly apply the update on the destination supernode. This last method is only interesting with a *right-looking* approach which will consume messages as soon as they are received. This two techniques will be detailed in chapter 3.

fan-out/right-looking methods are generally used for dense matrices factorizations where contributions must be sent the sooner as precedence constraints among computation are strong. In a sparse matrix parallel decomposition context, the *fan-in* approach happens to be more efficient because it can reduce communications volume significantly. Indeed, as one can see in figure 1.8, in the sparse matrix case, contributions are often smaller than the targeted block (or column-block) and the cost of these "small" communications is harder to hide. On the contrary, storing these contributions locally can lead to a memory overhead which can become very high, or even prohibitive when factorizing huge problems [HNP91]. In this case, using a mix of shared memory, with no fan-in buffer requirement but direct updates using shared memory, and distributed memory algorithm can lead to a good tradeoff.

1.3.3.4 Triangular system solve

Once the matrices L and U (or L and D) are computed, the solution can be obtained by solving successively the following systems:

$$L.y = b \tag{1.1}$$

$$\text{(only with } LDL^T)D.z = y \tag{1.2}$$

$$U.x = z \text{ or } L^T.x = z \tag{1.3}$$

The first step, called *forward substitution* phase, solves the equation 1.1. The second one, called *diagonal* phase, computes the quotient of the previously computed vector over the diagonal (equation (1.2)), it's only computed for Cholesky-Crout decomposition (LDL^T). Finally, the *backward substitution* phase, solves the equation 1.3. In the case of a single right-hand-side (i.e. b is a vector), the triangular solve step is cheap in number of operations compared to the numerical factorization. Thus, data are distributed on the node regarding the factorization [GKJ98; Jos+99]. In the column-block data distribution context, b's elements are distributed using the same distribution as the columns of A. In the case of multiple right-hand-sides, b is a rectangular $n \times m$ matrix, where m is the number of right-hand-sides. If m is large enough, the computational intensity when solving the triangular systems becomes comparable to the one of the numerical factorization, and redistributing A and b on the computational nodes could be worth trying. While this step is not often detailed in the literature it can be critical in simulations. Indeed, several triangular systems can be solved for one factorization. For example, it occurs when the factorized matrix is used as a preconditioner for an iterative method. Multiple triangular systems solving steps also occurs when the solver uses a static pivoting [LD98], where nil values found on the diagonal are replaced by an ϵ, instead of real pivoting, where columns and/or rows permutations maximize values on

the diagonal and increase the stability. Such pivoting results in an approximate factorization that can be used as a preconditioner in an iterative solver to reach an accurate solution.

We have described here the different part of a direct sparse linear solver. Different implementations of such a solver were developed to solve linear systems in simulation software. The next subsection describes the existing implementations and their particularities.

1.3.4 Linear algebra on homogeneous clusters

During last 15 years, direct sparse linear solvers have been improved significantly [Ame+01; Gup01; HRR02; LD03]. It is now possible to solve efficiently, and in a reasonable time 3D problems with millions of unknowns. To face the multiplication of SMP and NUMA based super-computers, linear solvers had to propose implementations more suited to multi-core architectures. Most of them first adapted their sequential version using the Message Passing Interface (MPI) model to exploit clusters. Others developed multi-threaded versions adapted to multi-core architectures. Finally, some of these solvers proposed algorithms using a combination of these solutions. We present here the main existing sparse direct solvers which are: CHOLMOD, MUMPS, PARDISO, PaStiX, SuperLU, TAUCS, UMFPACK, and WSMP. This subsection details the different methods used by those libraries to exploit clusters and solve large problems. An exhaustive list of sparse linear algebra solvers can be found in [Don03].

MUMPS solver is developed by Patrick Amestoy team in Toulouse and Jean-Yves L'Excellent in Lyon. It uses a multifrontal method with a dynamic computation scheduling [Ame+01; ADL98]. This solver provides many numerical preprocessing techniques to improve the stability of the numerical factorization. To handle large memory requirement, an efficient out-of-core implementation can also be used. With this version one can limit the memory requirement to the minimum by storing most of the data on the hard drive and solve larger problems on clusters with low memory available. The last distributed version is a MPI version developed in Fortran. Recently, a study [LS13] was carried out to target multi-core architectures. It proposes to use multi-threaded BLAS and add some OpenMP loop instructions to parallelize main sequential loops. With this solution, a good efficiency has been obtained on the eight computational cores during the experimentation. Most of the parallelism is here exploited by the multi-threaded BLAS library.

Timothy A. Davis, formerly from the University of Florida and now at Texas A&M University, proposes several solvers. UMFPACK [Dav04] is an unsymmetric-pattern multifrontal solver. It provides scaling techniques to improve the stability of the factorization. CHOLMOD [Che+08] uses a left-looking supernodal sparse Cholesky method. It can also solve unsymmetric problems by symmetrizing the pattern of the matrix, and performing the symmetric operations. Both solvers can also handle rectangular matrices and are integrated into MATLAB [The05]. These solvers are implemented using shared memory parallelism.

Silvian Toledo et al. developed several solvers in TAUCS [TCR03]. They implemented shared memory version of the multifrontal method and a left looking supernodal sparse linear solver. They also propose incomplete factorization methods and out-of-core solution. The authors parallelized their solvers using Cilk language extension.

PARDISO [SG04] is developed by Olaf Schenk and Klaus Gärtner team to target specifically shared memory machines. The PARDISO solvers now proposes a bit-identical solution for shared memory and distributed architectures. This solver relies on a supernodal method with a mixing left and right-looking scheduling method. It is developed in Fortran, and the shared memory algorithm uses OpenMP directives. As for MUMPS, non-symmetrical permutation techniques based on coefficient values are offered to obtain a better matrix conditioning. This solver is included in the Intel MKL library

Jim Demmel, Sherry Li et al. have developed several versions of their solver SuperLU to address sequential, shared memory or distributed memory machines [Dem+99; GDL07]. The underlying algorithm is similar to PARDISO's one: it uses a supernodal method with a hybrid left and right-looking communication scheme. SuperLU_Dist is the distributed memory version using MPI, and SuperLU_MT [Li08] is the shared memory one. This version had been developed before the P-Thread library and OpenMP emergence. It used inter-process sharing memory techniques. Several solutions have been developed to address the different architectures and operating systems. The package now also proposes OpenMP and POSIX threads versions that have been developed to handle new architectures and have a more generic code. A hybrid version, based on SuperLU_Dist is proposed in [YL12]. The authors present a scheduling strategy to improve the scaling of SuperLU_Dist on a large number of cores and integrated OpenMP to save memory while exploiting more efficiently large multi-core architectures.

In [HS13], authors present several multi-core sparse direct factorization algorithms from HSL [Duf06] (Harwell Subroutine Library). This library provides a collection of algorithms, from which direct sparse solvers, to solve sparse linear systems or eigenvalues problems. In particular, `HSL_MA87` [HRS10] algorithm provides a DAG based Cholesky shared memory factorization comparable to the shared memory implementation of PaStiX direct solver. Their main differences and a performance comparison are detailed in [Hog10].

Finally, PaStiX [FR08; Ram00] solver, by Pierre Ramet et al., and WSMP [Gup07], by Anshul Gupta, George Karypis and Vipin Kumar, provide hybrid MPI and POSIX Threads implementations. They are both developed in C and rely respectively on a supernodal right-looking method and a multifrontal method. Both implementations provide four running modes: sequential, shared memory, distributed memory, or hybrid shared and distributed memory. PaStiX solver is a single library than can be built in the different modes using compilation flags. On the contrary, WSMP has been developed in two distinct versions: a distributed memory and a shared memory one. These libraries can be used separately or coupled to obtain the hybrid version. Using MPI for the distributed memory communications and POSIX threads in shared memory, these

solvers can significantly reduce the memory overhead resulting from the communication buffers and, in WSMP case, from the number of contributions blocks in the multifrontal method that is an obstacle to the usage of direct methods for large 3D problems. These two solvers are meant to be the best candidates to exploit as efficiently as possible clusters of multi-core nodes.

1.3.5 Linear algebra on heterogeneous clusters

We presented in last subsection how sparse linear algebra got adapted to the emergence of multi-core nodes. Linear algebra libraries propose either a distributed memory, a shared memory, or mix of distributed and shared memory implementations. As we have seen earlier, new machines now commonly integrate accelerators such as GPUs or Xeon Phi. Sparse linear algebra has to evolve to benefit as much as they can from these new energy-efficient devices. This subsection presents how the dense linear algebra community initiates its evolution toward heterogeneous architectures. Then, it details the existing sparse direct solvers evolution toward those architectures. Indeed, the sparse linear algebra algorithms are comparable to dense linear algebra. The sparsity however introduces more complexity and irregularity. Due to this complexity, preliminary works are generally first performed on dense linear algebra, and then extended, when possible, to sparse linear algebra.

1.3.5.1 Dense linear algebra evolution

The dense linear algebra community spent a great deal of effort to tackle the challenges raised by the sharp increase of the number of computational resources. Because of their heavy computational cost, most of their algorithms are relatively simple to handle. Avoiding common pitfalls such as the "fork-join" parallelism and expertly selecting the blocking factor (i.e. the size of the blocks the matrix is split in) provides an almost straightforward way to increase the parallelism, and thus achieves better performance. Moreover, thanks to their natural load-balance, most of the algorithms can be approached hierarchically, first at the node level, and then at the computational resource level. In a shared memory context, one of the seminal papers [But+06] replaced the commonly used LAPACK column major layout with one based on tiles/blocks. Using basic operations on these tiles exposes the algorithms as a Directed Acyclic Graph (DAG). The nodes of this graph represent the tasks to be executed, and the directed edges represent the data dependencies among them. In shared memory, this approach quickly generates a large number of ready tasks, while, in distributed memory, the dependencies allow the removal of hard synchronizations. This idea leads to the design of new algorithms for various algebraic operations [But+09], now at the base of well-known software packages like PLASMA [Agu+09], that aims at replacing LAPACK in a shared memory context. MAGMA [NTD11] provides LAPACK linear algebra routines equivalents for either NVIDIA or AMD GPUs, and Intel Xeon Phi relying on vendors BLAS implementations. Indeed, each of the device vendors quickly provided its implementation of the BLAS routines. NVIDIA developed cuBLAS [NVI08]

for its GPU, and AMD and Intel respectively adapted ACML and MKL to use their accelerators. MAGMA exploits those libraries to extend higher level algorithms from LAPACK to those devices. With distributed memory systems, DPLASMA [Bos+11] was developed on top of PaRSEC to be efficient on distributed heterogeneous systems and eventually replace ScaLAPACK. A concurrent to those solutions developed by the University of Tennessee is the FLAME [Igu+12] library developed by the University of Texas at Austin. This library is also adapted to distributed heterogeneous platforms.

This idea is recurrent in almost all novel approaches surrounding the many-core revolution, spreading outside the boundaries of dense linear algebra. Looking at the sparse linear algebra, the efforts were directed toward improving the behavior of the existing solvers by taking into account both task and data affinity, and relying on a two-level hybrid parallelization approach, mixing multi-threading and message passing. We will now describe the evolution of sparse linear algebra solvers during last years.

1.3.5.2 Sparse linear solver on heterogeneous machines

At the moment this thesis started, only few preliminary studies had been performed toward adapting sparse linear solvers to heterogeneous architectures. As a first step to heterogeneous system we can consider the evolution toward large NUMA nodes utilization. The chosen approach follows the one for dense linear algebra, fine-grained parallelism, thread-based parallelization, and advanced data management to deal with complex memory hierarchies. Most of the time, this is done using task-based runtime systems that can be seen as a first step toward heterogeneous clusters. We can cite `qr_mumps` [But13] for sparse QR fine grain multi-threaded factorization using a DAG of tasks. QR factorization is more stable but involves more operations than a Cholesky factorization (in dense: $\frac{4n^3}{3}$ with QR factorization versus $\frac{n^3}{3}$ with Cholesky). This additional computational intensity makes it a good candidate to target GPUs. In [YDR13], authors propose a mono-GPU implementation of the QR factorization and obtain a speed-up up to 11 compared to the 12 cores execution.

In [Vud+10], authors pointed out the limits of the GPU computations. In particular, they implemented a cuBLAS based direct sparse linear solver and compared it to the MKL one. The speed-up obtained, without taking into account the data transfers, compared to a single core did not exceeded three. They concluded that using a shared memory implementation would give better performances than using a full GPU based sparse solver.

Another option, implemented by the HSL team in the SPRAL [HOS14] library, proposed to execute the whole factorization and upward/backward substitutions algorithms on a GPU device. SPRAL implements a multifrontal method with pivoting and obtains decent speedup compared to a pair of quad-core processors. The main drawback of this approach is that the library is limited by the GPU device internal memory, but one could imagine implementing a domain decomposition method on top of this solver to reach higher scale matrices.

Pardiso team also investigates the possibility of using GPUs for sparse direct factorization. In [CSB07; SCB08], authors first compare the performance of single precision

matrix-matrix multiplication on CPU and GPU. If for large sizes GPU outperforms the CPU, for small matrices, as we encounter in sparse solvers, it is rarely the case. We can also notice that, because of cache effects, if the performance of the $A \times B$ product are symmetric on a GPU it is not the case on CPU and having a tall and skinny matrix A with a small B, as it occurs with sparse solvers, is really bad for performance. In a second time, the authors propose to replace the BLAS calls with cuBLAS ones if the number of Flops involved is beyond a given limit. Doing so, in simple precision, they could obtain a speed-up up to 4 against the simple precision CPU version (6.5 against double precision).

In CHOLMOD [RSD14], investigations have been carried toward the use of cuBLAS kernel for the dense linear operations. Operations on the lower part of the elimination tree are sent to the GPU using concurrent batched kernels while for the largest supernodes, at the top of the elimination tree, operations are processed either on CPU or GPU depending on a threshold. The chosen threshold is particular for each kernel as their efficiency is different. Doing so they could achieve a speed-up of up to 4.1 against the CPU version in double precision.

Authors of [KP10] propose to use a GPU to perform `GEMM` uptates in their LDL^T solver, both in double precision and mixed precision and obtained a speed-up up to 4 in double and 2.9 in mixed precision (i.e. cuBLAS calls in simple precision and CPU code in double precision) on a benchmark using one test case with a varying number of unknowns.

In [Luc+11], authors also replaced the BLAS calls on a multifrontal method and outperform the multi-core BLAS using cuBLAS in simple precision.

In [YWP11], in the multifrontal solver UMFPACK, the replacement of some of the BLAS operations with cuBLAS equivalent operation following certain threshold can give a speed-up of 3 in double precision against the CPU version.

On multifrontal methods, a study has been performed to estimate the benefits that one can obtain using accelerators in WSMP [Geo+11]. The multi-frontal methods, which generate operations on larger datum blocks are well suited to target accelerators. In this study, part of the BLAS operations involved in direct sparse factorization – `POTRF` (factorization), `TRSM` (solve) and `SYRK` ($C = \alpha AA^T + \beta C$) – is off-loaded to the GPU following empirical criteria based on the number of operations involved. In most of the presented cases, off-loaded `POTRF` is not relevant, it is only profitable with large 3D problems that lead to larger blocks with more computation to perform. On the contrary, it is worth off-loading the two other operations. In mixed precision, with two CPU threads and two GPU threads the speed-up reached is four times what can be obtained from four CPU threads in double precision.

In [SVL14], authors added OpenMP and CUDA support to SuperLu_Dist. The most intensive part of the computation, the Schur complement (columns on the right) update, is off-loaded to the GPU. To increase the computational intensity of the kernel, the whole update is computed at once. The update is then pipelined, the first columns are computed on CPU while data are transferred for the other parts. After each block computation, the results are transferred to the CPU which scatters the data on the

destination blocks using OpenMP and MPI. Other BLAS operations are performed on CPU using multi-threaded implementation. With a larger temporary memory requirement, authors achieve up to three times faster computations compared to the best full MPI implementation.

In [Bie+10], authors present a different vision of direct solver, not as a black box, but tightly coupled with a finite element method. They consider their finite element mesh as the recursive refinement of an original mesh. They obtain a tree where an element's sons are the elements obtained after refinement. They apply a multi-frontal method on this tree, where the elementary matrix is only assembled when it will be factorized, after receiving the Schur complement from its sons' factorizations. They can extract parallelism from the geometry and handle hp-adaptative (h for mesh refining using cell splitting, p for order increasing) finite elements method easily without recomputing the whole factorization when a part of the mesh is refined. In [Kyu14] they describe how they parallelized this solver using direct acyclic graphs and OpenMP. This way, they obtained a shared-memory solver and could compete both with MUMPS and PARDISO when elements' order increases.

Thus, many studies concerning sparse linear solvers using accelerators were performed using multifrontal methods, which a priori is more suited. Indeed, large front updates are produced in multifrontal method, allowing efficient `GEMM` executions. Even in the SuperLU_Dist supernodal method, the implementation they designed mimics the front factorization by gathering all operations resulting from all updates from one panel. In these studies GPUs are handled manually by the application code, this one will need to be adapted for each new architecture. The main idea is to treat some parts of the task dependency graph entirely on the GPU. Therefore, the main originality of these efforts resides in the methods and algorithms used to decide whether a task can be processed on a GPU. Usually, this was achieved through a threshold based criterion on the size of the computational tasks.

In our approach, we want to study GPUs usage with the supernodal methods. Our approach is close to the SuperLU_Dist one, but we do not want to increase the memory requirement which is already very important with direct methods. These methods are less memory consuming and can then address larger problems. Moreover, we want to estimate the potential use of generic task-based runtime systems that gives a higher abstraction of the machine architecture. Thus, moving to new computational units is less painful to the application developer.

1.4 Discussion

In this chapter, we have presented the current evolution of the hardware used in high performance computing field. New barriers have shown up preventing the continual increase of frequency and the multiplication of the processors. The frequency could not increase anymore without overheating the system. To bypass this limit, manufacturers proposed to multiply the number of cores: first, with a flat architecture, connecting multiple cores to a shared memory with symmetric memory accesses; then, with more

hierarchical architectures called NUMA, simpler to build but more complex for the developer. The multiplication of sophisticated cores led to prohibitive energy consumption. To work around this new barrier, manufacturers provided new accelerators, based on GPUs or integrating many simplified and low energy-consuming cores. Developing efficient kernels for these new devices is a tough task for the developers. New techniques and tools can be used to produce high performance code adapted to these complex machines (either for distributed, shared memory or accelerator programming). However, with these new heterogeneous and very evolving machines, a need for abstraction of the machine architecture arose. As we have shown, task-based runtime systems provide a solution to implement algorithms using an abstraction of the architecture, more flexible and portable.

From fast iterative solvers to robust direct solvers, the linear algebra community developed a large panel of methods to solve sparse linear systems depending on the characteristics of the system. The simulation developer must then choose the best algorithm. In this thesis, we focused on sparse direct solvers. Thus, we presented the different steps involved in a sparse direct factorization. We have seen that direct sparse linear solvers have already been adapted to the distributed clusters of SMP and, afterwards, NUMA multi-core nodes. The distributed parallelism is generally exploited using MPI, while there are various implementations to handle the shared memory parallelism. During last three years, the sparse linear algebra community studied the possibility to integrate GPU accelerators into sparse linear direct solvers. Most of them targeted multifrontal methods which use more compute intensive kernels well suited to GPUs. In these attempts, developers manually handled GPUs using the CUDA toolkit. In this work, we want to study the path taken by dense linear algebra developers, and use task-based runtime systems to abstract the machine architecture from the algorithm. Once the algorithm is transposed into a DAG of tasks, one can use accelerators by proposing new kernels to the runtime system.

Chapter 2

Sparse factorization on shared-memory heterogeneous machines

Contents

2.1 Framework **39**
2.1.1 PaStiX original algorithm description 39
2.1.1.1 Scheduling and data mapping 39
2.1.1.2 Numerical factorization 40
2.1.2 Elected runtime systems 41
2.1.2.1 PaRSEC 45
2.1.2.2 StarPU 46
2.2 Implementation on top of generic runtime systems **49**
2.2.1 PaRSEC implementation 50
2.2.2 StarPU implementation 53
2.2.3 Multi-core Architecture experimentation 55
2.3 Heterogeneous systems **57**
2.3.1 Implementation of a specific `GEMM` kernel 60
2.3.2 Data mapping over multiple GPUs 66
2.3.3 Heterogeneous experiments 69
2.3.4 Memory study 69
2.4 Optimizations **71**
2.4.1 Task granularity adapted to the runtime 71
2.4.2 Block splitting algorithm 73
2.5 Discussion **79**

This thesis aims at proposing a sparse direct solver adapted to emerging clusters of heterogeneous nodes. In previous chapter, we have presented the architectures and solutions to use them. We also presented the multiple solutions available to solver sparse linear systems. In particular, we presented the sparse direct methods framework and compared the main direct sparse solvers libraries available.

Among previously presented solvers, we chose to rely on the PaStiX solver to perform the studies described in this thesis. This solver was initiated in Bordeaux, to be used in electromagnetic field by the CEA Cesta. This library has been written in C, initially using MPI for the parallel version. Then, to reduce memory overhead induced by MPI communication buffers, a hybrid MPI/POSIX Threads version has been developed. Besides being locally developed, several criteria guided our choice of this solver. We want to describe our algorithm as a task graph in order to execute it with a task-based runtime. In PaStiX, operations are already separated into tasks that are distributed by a specific scheduler on the computational cores. Moreover, by its advanced adaptation to multi-core clusters, PaStiX is a good challenge for our performance studies. Last, but not least, with its open source Cecill-C license, PaStiX is a good framework to integrate our developments.

For a better flexibility and upgradability in our developments we elected the generic task-based system to integrate accelerators into the PaStiX library. In our exploratory approach moving toward a generic scheduler for PaStiX, we considered two different runtime systems: StarPU and PaRSEC. Both runtime systems have been proven mature enough in the context of dense linear algebra, while providing two orthogonal approaches to task-based systems.

In this chapter, we describe the implementation of a sparse factorization algorithm on top of tasks based runtime systems to target heterogeneous shared memory machines; distributed memory systems will be addressed later in Chapter 3. We first detail the library used for our implementation. We describe the static scheduling system and the numerical factorization algorithm used in PaStiX solver. This implementation, prior to this thesis, will be referred to as the *original* PaStiX implementation in all the following of the manuscript. We also present the two task-based runtime systems we have chosen to use in our experiments.

After that, we describe the implementation of PaStiX algorithm on top of the two elected runtimes. Then, multi-core results validate our task-based runtime implementation.

We also present our specialized kernel which performs our updates on trailing supernodes and study its performances separately from PaStiX. Then, we use this new kernel with task-based runtime systems to benefit from all the computing resources available on an heterogeneous node in our sparse linear algebra library.

2.1 Framework

In this section, we go further into PaStiX algorithms. We explain how we can exhibit parallelism, at multiple levels, from the operations involved in the factorization. We also describe the data distribution and the computations scheduling. Then we detail the parallel blocked supernodal algorithm used in PaStiX. Static scheduling, and factorization algorithm are the only parts of the code that are involved in this thesis, a latter study may address the forward and backward substitutions. Finally, we present and compare the two generic task-based runtime systems selected for this thesis.

2.1.1 PaStiX original algorithm description

The first issue encountered when developing an application for a distributed memory machine is to decide on the data distribution. The PaStiX solver, contrary to some other solvers, chooses to statically distribute its data once, during preprocessing. Data are not redistributed dynamically afterwards, during factorization, as it might be done in MUMPS for example. To achieve good performance, the data distribution and numerical factorization steps must be strongly coupled.

2.1.1.1 Scheduling and data mapping

The numerical factorization algorithm efficiency relies on a good partitioning and data mapping over the computational nodes. This step has been developed in Pascal Hénon's thesis [Hén01]. It statically computes a *balanced regulation* for the solver following load balancing between processors and precedence constraints between blocks of the matrix. Dependency rules between computations are given by the blocked elimination tree structure described in 1.3.3.2 page 22: the decomposition of a column block cannot be performed before all its descendants in the elimination tree have brought their contributions to it. Conversely, a column block's factorization will produce a contribution to all its ancestors. The wider the elimination tree is, the more the sparsity of the matrix induces parallelism. The computational cost regulation and the data mapping over processors rely on a precise modelization of BLAS routines and communications required by the algorithm. Those cost models have been developed in Pierre Ramet's thesis [Ram00] and are obtained experimentally on a large range of different sized input data on the target architecture. The behavior of the studied operations is then represented by a polynomial formula using a linear regression. A message passing benchmark tool has been designed to simulate communication through the computation of the network's latency and bandwidth. The MPI/Thread implementation of the solver required distinguishing between intra-node communications and inter-node ones. Indeed, a multi-threaded application incurs no cost for a shared memory data transfer.

The goal of this data distribution step is to exploit, using cost models, the different existing level of parallelism in computation:

1. the coarse-grained parallelism, induced by independent computations between sub-trees of a same node of the elimination tree. This parallelism is induced by the sparsity of the matrix. As we have seen in 1.3.3.1 the reordering of the unknowns leads to the creation of an elimination tree. Distinct branches of the tree can be eliminated independently.

2. the medium-grained parallelism. To obtain more parallelism, one can split large nodes and distribute their computations over several processors (see 2.4.2). This parallelism is induced by the dense block factorization. It is also called node level parallelism.

3. the fine-grained parallelism, or micro-parallelism. It is obtained, when performing block operations, using the internal processor parallelism (optimal usage of the pipeline effect of super-scalar processors). This last level is absolutely necessary to obtain a good performance. It relies mainly, with a given *good block size*, on the usage of BLAS 3 routines [GKK97a; GKK97b; NP93; Rot96; RG94; RS94].

This scheduling step uses the elimination tree to distribute, in a balanced way, the independent column-blocks from the bottom of the tree. The sub-blocks of higher levels in the tree are most of the time bigger and denser. A larger number of operations are performed on these blocks. Thus, it is necessary to split them into smaller ones, using an optimal size, to extract more parallelism from them. After that, they are distributed, following a block cyclic pattern, to exploit the dense computation parallelism. Using this process we can obtain a good static computation regulation, by simulating the factorization using cost models [FR95; GN89; Rom94; RG94].

This technique can be used following a block-column unidimensional distribution scheme (1D), which allows to exploit the parallelism linked with the independence of the computations between column-blocks, or following a bi-dimensional scheme (2D) where we also exploit the independence between computations on elementary blocks of this distribution, as it is done with dense matrices. A 2D distribution gives a better scalability of the parallel solver [RG94; Sch93] but the timing overhead during preprocessing is very large. Indeed, the complexity of the simulation used to compute the static distribution increases. Moreover, a 2D distribution creates a communication overhead compared to a 1D distribution.

2.1.1.2 Numerical factorization

The previous step provides a static scheduling of computations that must be followed *literally* if we do not want to break the computed regulation. To store this scheduling order we use an ordered bi-dimensional array, $Task(t, i)$, associating the factorization of a column-block i to the thread t. Moreover, to obtain a good reuse coherency in BLAS calls, PaStiX solvers bind each computational thread on a computational core using HwLoc library [Bro+10]. Once bound to a computational core, during numerical factorization, each thread follows the array of N_t tasks that are assigned to it, as shown

in Algorithm 4 that uses notations from Figure 2.1. For each task, the thread executes the following steps:

- it receives and adds the column-block associated data;
- it waits for local contributions to be applied;
- it factorizes the diagonal block;
- it solves an off-diagonal blocks triangular system;
- it computes the outgoing contributions and sends them.

The first step of each task computation is data reception (lines 3 to 7). In this computational loop, the thread waits for each remote contribution required by the factorization of the column-block k. To improve communication reactivity, each thread waits for any communication it might receive, not specifically the ones needed for column-block k. Additions can then be performed before the computation of the receiving task. Once all remote contributions have been received, the thread waits for all local contributions that are computed by other threads belonging to the same MPI process (lines 8 to 10).

The remainder of the factorization algorithm, for the task computation, is similar to the sequential algorithm: diagonal block factorization and the solution of the off-diagonal blocks triangular system (line 14). The contribution addition is decomposed into two steps, the computation (lines 15 to 31) and the send (lines 32 to 36). We use here a "fan-in/right-looking" method, hence, remote contributions are added in a provisional block-column before being sent (lines 27 and 28). In the hybrid MPI/Thread version, the storage of these block-columns is avoided as we can directly add contributions on the recipient block-columns with shared memory (lines 19 and 20). In this case, we use a "fan-out" approach. The two approaches are automatically mixed following the MPI/Thread distribution.

Now that we have seen more in details the PaStiX main algorithm we can present the two task-based runtime systems we want to couple PaStiX with. Using these libraries, the algorithm can be separated from the hardware kernel implementation, and ease the introduction of new computational devices.

2.1.2 Elected runtime systems

The PaRSEC [Bos+12] distributed runtime system, developed by the University of Tennessee in Knoxville, is a generic data-flow engine supporting a task-based implementation and targeting hybrid systems. Domain specific languages are available to expose a user-friendly interface to developers, and allow them to describe their algorithm using high-level concepts. This programming paradigm relies on an abridged representation of the tasks and their dependencies. This representation structure is agnostic to algorithmic subtleties, where all intrinsic knowledge about the complexity of the underlying

Notations: For each column-block k, $1 \leq k \leq N$,

- $Proc_k$ and $Thread_k$ are respectively the process and the thread elected by the static scheduler to factorize the column-block k;
- L_k is the column-block k, symmetrically, U_k is the row-block from U;
- $L_{(i),k}$ is the i-th block in column-block k, symmetrically, $U_{k,(i)}$ is the i-th row-block k from U;
- b_k is the number of off-diagonal blocks in column-block k;
- C_k^L (resp. C_k^U) is the contributions column-block computed on process $Proc_k$ for column-block k to be applied on L (resp. U);
- NCD_k is the number of remote contributions the column-block k will receive;
- NC_k is the total number of contributions the column-block k will receive: remotely and locally;
- NT_t is the number of tasks assigned to thread t;
- $Task(t, l)$ is the lth column-block thread t has to factorize;
- $getLock()$ (resp. $releaseLock()$) are used to enter (resp. leave) a protected shared memory area. It prevents two threads from modifying the same data concurrently.

For the sake of simplification, we consider that $A_{(i),(j)} \sim A_{i,j}$. In other words, we consider that $L_{i,j}$ is in column block j and $U_{i,j}$ is in row block i.

Figure 2.1 – Notations used in algorithms

Algorithm 4 Parallel column-block factorization on thread t of process p.

```
1:  For l = 1 to NT_t Do
2:      k = Task(t, l)
    ▷ Wait for a contribution to a local column-block r and add the received contribution
    C_r^L, C_r^U in the correct place
3:      While NCD_k > 0 Do
4:          (r, C_r^L, C_r^U) ← WaitContributions();                    ▷ Receiving
5:          GetLock(L_r);
6:          L_r = L_r − C_r^L
7:          U_r = U_r − C_r^U
8:          RealeaseLock(L_r);
9:          NCD_r--; NC_r--;
10:     End While
11:     While NC_k > 0 Do
12:         Wait for a signal for k
13:     End While
14:     Task l computation: A_{k,k} factorization and L_k and U_k solve    ▷ Computation
15:     For j = 1 to b_k Do
16:         For i = j to b_k Do
17:             If A_{i,j} is local Then
    ▷ Prevent other shared memory access to the blocks owning A_{i,j} and A_{j,i}
18:                 GetLock(L_{i,j});
19:                 L_{i,j} = L_{i,j} − L_{i,k}.U_{k,j};
20:                 U_{j,i} = U_{j,i} − L_{j,k}.U_{k,i};
21:                 RealeaseLock(L_{i,j});
22:                 NC_j--;
23:                 If NC_j = 0 Then
24:                     Send signal to column-block j
25:                 End If
26:             Else
27:                 CL_{i,j} = CL_{i,j} − L_{i,k}.U_{k,j}
28:                 CU_{j,i} = CU_{j,i} − L_{j,k}.U_{k,j}
29:             End If
30:         End For
31:     End For
32:     For i = 1 to b_k Do
33:         If A_i is non local and C_i has received its contributions Then
34:             Send C_i contribution to process Proc_i for Thread_i    ▷ Sending
35:         End If
36:     End For
37: End For
```

algorithm is extricated, and the only constraints remaining are annotated dependencies between the tasks [CJY04]. This symbolic representation, augmented with a specific data distribution, is mapped on a particular execution environment. The runtime supports the usage of different types of accelerators, GPUs and Intel Xeon Phi, besides distributed multi-core processors. Data are transferred between computational resources based on coherence protocols and computational needs, with emphasis on minimizing the unnecessary transfers. The resulting tasks are dynamically scheduled on the available resources following a data reuse policy mixed with different criteria for adaptive scheduling. The entire runtime targets very fine grain tasks (order of magnitude under ten microseconds), with a flexible scheduling and adaptive policies to mitigate the effect of system noise and take advantage of the algorithmic-inherent parallelism to minimize the execution span.

The experiments presented in this thesis take advantage of a specialized domain specific language of PaRSEC, designed for affine loops-based programming [Bos+11]. This specialized interface allows for a drastic reduction in the memory used by the runtime, as tasks do not exist until they are ready to be executed, and the concise representation of the task-graph allows for an easy and stateless exploration of the graph. In exchange for the memory saving, generating a task requires some extra computations, and lies in the critical path of the algorithm. However, these task's generations are executed concurrently by each thread reducing this extra-cost. The runtime can explore the graph dynamically based on the ongoing state of the execution without need to compute and store the whole task graph.

StarPU [Aug+11] is a runtime system aiming to allow programmers to exploit the computing power of clusters of hybrid systems composed of CPUs and various accelerators (GPUs, Intel Xeon Phi, etc.) while relieving them from the need to specially adapt their programs to the target machine and processing units. The StarPU runtime supports a *task-based programming model*, where applications submit computational tasks, with dependency analysis computed at the submission. These tasks are composed with the data from the algorithm and the different implementations of the kernel available (e.g. CPU or GPU implementations). Then StarPU schedules these tasks, and the associated data transfers on available resources. The data that a task manipulates are automatically transferred among the accelerators and the main memory in an optimized way: minimized data transfers, data prefetch, communications overlapped with computations, etc. Programmers are relieved of the scheduling issues and technical details associated with these transfers. StarPU takes particular care of scheduling tasks efficiently by establishing performance models of the tasks through on-line measurements, and then using well-known scheduling algorithms from the literature. In addition, it allows scheduling experts, such as compilers or computational library developers, to implement custom scheduling policies in a portable fashion.

The differences between the two runtime systems can be classified into two groups: conceptual and practical differences. At the conceptual level the main differences between PaRSEC and StarPU are the tasks submission process, the centralized scheduling, and the data movement strategy. PaRSEC uses its own parameterized language to

describe the DAG in comparison with the simple sequential submission loops typically used with StarPU. Therefore, StarPU relies on a centralized strategy that analyzes, at runtime, the dependencies between tasks and schedules these tasks on the available resources. On the contrary, through compile-time information, each computational unit of PaRSEC immediately releases the dependencies of the completed task solely using the local knowledge of the DAG. At last, while PaRSEC uses an opportunistic approach, the StarPU scheduling strategy exploits cost models of the computation and data movements to schedule tasks to the right resource (CPU or GPU) in order to minimize execution makespan. However, it does not have a data-reuse policy on CPU-shared memory systems, resulting in lower efficiency when no GPUs are used, compared to the data-reuse heuristic of PaRSEC. At the practical level, PaRSEC supports multiple streams to manage the CUDA devices, allowing partial overlap between computing tasks, and maximizing the occupancy of the GPU. On the other hand, StarPU allows to transfer data directly between GPUs without going through central memory, potentially increasing the bandwidth of data transfers when a datum is needed by multiple GPUs.

2.1.2.1 PaRSEC

PaRSEC uses the task graph differently avoiding manipulating the whole graph at anytime. This different usage implies a different way of describing the graph for the user. The data distribution and dependencies are specified using the Job Data Flow (JDF) format. Listing 1 presents the description of the dense `POTRF` task (i.e. diagonal block factorization) in a JDF format. The JDF file starts with a C preamble including all headers required to describe the task (line 1 to 7). Then, a set of global variables is defined to be used in the task descriptions (line 7 to 13). Next starts the description of the `POTRF` task. It takes one parameter k (line 18) that varies in the interval $[\![0, SIZE-1]\!]$ (line 21). Line 24 indicates that the task will be executed were the datum $A(k,k)$ is located. The task will require a `RW` (i.e. read and write) access to a datum `T` that comes either from the memory if it is the first block, or from the last `SYRK` (update from left blocks $L_{i,j} = L_{i,j} - L_{i,k}.L_{j,k}$) update (line 27). At the end of the `POTRF` task the `TRSM` tasks (Solve $L_{i,j}.L_{i,i} = A_{i,j}, j > i$) on the blocks beneath will receive the datum T (line 28) and it will also be and the data can be written back to the memory as it will not be modified anymore (line 29). Finally, the lines 31 to 33 of the JDF file indicate the C code that will correspond to the execution of the task. It can use the task parameters (here k), the data elements (here T) and the global variables defined earlier (here A, NB, $SIZE$, $uplo$, and $INFO$).

Listing 1 Part of the JDF description of the dense Cholesky factorization.

```
1 extern "C" %{
2 #include <plasma.h>
3 #include <core_blas.h>
4
5 #include "parsec.h"
```

```
6  #include "data_distribution.h"
7  %}
8
9  A              [type = "struct tiled_matrix_desc_t*"]
10 NB             [type = int]
11 SIZE           [type = int]
12 uplo           [type = PLASMA_enum]
13 INFO           [type = "int*"]
14
15 /*************************************************
16  *                        POTRF                  *
17  *************************************************/
18 POTRF(k)
19
20 // Execution space
21 k = 0..SIZE-1
22
23 // Parallel partitioning
24 : A(k, k)
25
26 // Parameters
27 RW T <- (k == 0) ? A(k, k) : T SYRK(k-1, k)
28      -> T TRSM(k, k+1..SIZE-1)
29      -> A(k, k)
30
31 BODY
32         CORE_zpotrf(uplo, NB, T, NB, INFO );
33 END
```

When the JDF file is ready, one can use the JDF translator `daguepp`. This JDF translator will define data elements (here T) as a function of the data flow in the C code together with some tools to handle the data elements (i.e. move it to global memory, pass it to other tasks...). Then, the user describes the data distribution in a C code and instantiates the DAG generator that constructs the first tasks (i.e. the tasks with no parent in the DAG that are ready at the beginning of the algorithm) of the DAG, and enqueues these tasks into PARSEC context. PARSEC will never construct the whole graph of the factorization but discovers it while the tasks are executed.

2.1.2.2 StarPU

Implementing an algorithm using STARPU is fairly simple. Listing 2 presents the dense Cholesky factorization written using STARPU. First, the user describes the *codelets*. A codelet is a structure describing a computational kernel with all the available implementations targeting different architectures. The codelet describes the number of data

accessed and the access mode (e.g. `STARPU_R`, `STARPU_W`, `STARPU_RW` standing respectively for read, write and read/write accesses). Here, we have three codelets, one per each of the BLAS operations involved in the factorization. The first one, `cl_spotrf` use one datum in read/write mode. The codelet is associated to a datum (or a set of data) with the `starpu_insert_task()` function. The tasks are inserted into StarPU runtime system that will execute them, when the dependencies are satisfied.

StarPU builds the DAG following the sequential order of submission, and following data dependencies. Indeed, the user only follows the sequential algorithm he wants to execute and replace the calls to the kernels with task submissions. The submission call is asynchronous; the user will continue submitting all its tasks without waiting for their execution. Once a task is ready (i.e. all the tasks it depends on have been executed), it is scheduled for execution on a given resource and, at the end of its execution, it releases the dependencies on its children in the task graph. StarPU is then able to decide which tasks can be executed concurrently on the multiple computing units following the data accesses and the access modes. It also decides which computing unit will execute the tasks depending on the expected kernel performance and the computing units availability.

Listing 2 StarPU dense Cholesky example.

```
1
2  void spotrf_cpu(void *buffer[], void *cl_arg)
3  {
4    float   *A     = (float *)STARPU_MATRIX_GET_PTR(descr[0]);
5    unsigned nx    = STARPU_MATRIX_GET_NY(descr[0]);
6    unsigned ld    = STARPU_MATRIX_GET_LD(descr[0]);
7    int      info;
8    spotrf('L',nx, A, ld, info);
9  }
10 void spotrf_cuda(void *buffer[], void *cl_arg);
11 void strsm_cpu(void *buffer[], void *cl_arg);
12 void strsm_cuda(void *buffer[], void *cl_arg);
13 void sgemm_cpu(void *buffer[], void *cl_arg);
14 void sgemm_cuda(void *buffer[], void *cl_arg);
15
16 static struct starpu_codelet cl_potrf =
17 {
18   .cpu_funcs  = {spotrf_cpu, NULL},
19   .cuda_funcs = {spotrf_cuda, NULL},
20   .nbuffers = 1,
21   .modes = {STARPU_RW}
22 };
23
24 static struct starpu_codelet cl_strsm =
25 {
```

```
26    .cpu_funcs  = {strsm_cpu, NULL},
27    .cuda_funcs = {strsm_cuda, NULL},
28    .nbuffers = 2,
29    .modes = {STARPU_R, STARPU_RW}
30  };
31
32  static struct starpu_codelet cl_sgemm =
33  {
34    .cpu_funcs  = {sgemm_cpu, NULL},
35    .cuda_funcs = {sgemm_cuda, NULL},
36    .nbuffers = 3,
37    .modes = {STARPU_R, STARPU_R, STARPU_RW}
38  };
39
40  int main(int argc, char **argv)
41  {
42    /* Initialisation of the matrix data handles and
        parameters*/
43    starpu_init();
44    ...
45    /* create all the DAG nodes */
46    for (k = 0; k < nblocks; k++) {
47      /* insert the factorization of A_{k,k} */
48      ret = starpu_insert_task(&cl_sportf,
49                               STARPU_RW, MATRIX_DATA[k][k],
50                               0);
51      for (j = k+1; j<nblocks; j++) {
52        /* A_{k,j} = A_{k,j}, A^-1_{k,k} */
53        ret = starpu_insert_task(&cl_strsm,
54                                 STARPU_R,  MATRIX_DATA[k][k],
55                                 STARPU_RW, MATRIX_DATA[k][j],
56                                 0);
57        for (i = k+1; i<nblocks; i++) {
58          if (i <= j) {
59            starpu_data_handle_t sdataki =
                starpu_data_get_sub_data(dataA, 2, k, i);
60            starpu_data_handle_t sdataij =
                starpu_data_get_sub_data(dataA, 2, i, j);
61
62            ret = starpu_insert_task(&cl_sgemm,
63                                     STARPU_R,  MATRIX_DATA[k][i],
64                                     STARPU_R,  MATRIX_DATA[k][j],
65                                     STARPU_RW, MATRIX_DATA[i][j],
```

```
66                                  0);
67          }
68        }
69      }
70    }
71    starpu_task_wait_for_all();
72    starpu_shutdown();
73    return 0;
74  }
```

2.2 Implementation on top of generic runtime systems

This section presents our implementation of a sparse direct linear solver on top of a generic task-based runtime system. As we have seen before, for this study, we use the PaStiX library as a starting point and couple it with two generic runtime systems: PaRSEC and StarPU. First, we describe the multi-core implementation, and the next section deals with integration of heterogeneous architectures. The section 2.4 describes the optimizations that were integrated to obtain an efficient implementation on top of tasks based runtime systems. These optimizations are enabled in all results shown in this section and the next one, and benefit both to the original scheduler and to the runtime system libraries.

We present how we adapted our direct sparse solver algorithm to generic runtime systems in order to target accelerators. The two scheduling systems we want to use to help accessing accelerators may lead to some additional cost compared to the lightweight original solvers scheduler and, to reduce the overhead, we had to adapt the task granularity to the runtime systems. To ensure the introduction of the runtime systems gives decent performance, we compare its execution with the original finely tuned PaStiX static scheduler.

Figure 2.2(b) presents the three tasks that compose the blocked sparse supernodal decomposition. This decomposition uses the same tasks as the dense factorization (Fig. 2.2(a)), but the sparsity induces irregularity in the size of the data it is applied to. The blocks created in a sparse context are also generally smaller than with dense linear algebra leading to less compute intensive tasks. The three tasks involved in a decomposition are:

1. the factorization of the diagonal block: $A_{ii} = L_{ii}L_{ii}^T$ (`POTRF`);
2. triangular system solving using the previously factorized block and the off-diagonal blocks of the panel: $L_{ji}L_{ii} = A_{ji}, j > i$ (`TRSM`);
3. the update to the trailing sub-matrix: $L_{jk} = L_{kj} - L_{ki}L_{ji}^T, j > i, k > j$ (`GEMM`).

Whereas the tasks dependencies graph from a dense Cholesky factorization [But+09] is extremely regular (Fig. 2.2(c)), the DAG describing the sparse supernodal method

contains rather small tasks with variable granularity and less uniform ranges of execution space. This lack of uniformity makes the DAG resulting from a sparse supernodal factorization complex (Fig 2.2(d)), increasing the difficulty to efficiently schedule the resulting tasks on homogeneous and heterogeneous computing resources.

The current scheduling scheme of PaStiX exploits a 1D-block distribution, where a task assembles a set of operations together, including the tasks factorizing one panel (`POTRF` and `TRSM`) and all updates generated by this factorization (`GEMM`s). However, increasing the granularity of a task in such a way limits the potential parallelism and has a growing potential of bounding the efficiency of the algorithm when using many-core architectures. To improve the efficiency of the sparse factorization on a multi-core implementation, we introduced a way of controlling the granularity of the BLAS operations. This functionality dynamically splits update tasks, so that the critical path of the algorithm can be reduced. In this thesis, for both the PaRSEC and StarPU runtime systems, we split PaStiX tasks into two subsets of tasks:

`panel`(C) or *panel factorization:* diagonal block factorization and off-diagonal blocks updates, performed on panel C;

`gemm`(B) or *`GEMM` updates:* the updates from off-diagonal blocks of the panel C_1 to one other panel c_2 of the trailing sub-matrix, where C_1 owns B and B is in the row block of the diagonal block of C_2 (we say that C_2 is the column block facing B).

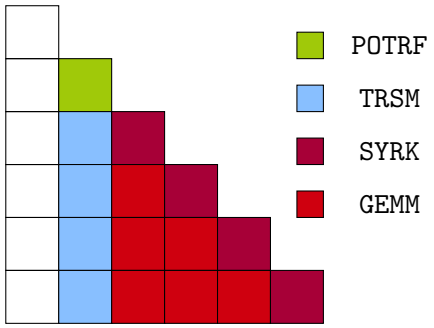
Hence, the number of tasks is bound by the number of blocks in the symbolic structure of the factorized matrix.

Moreover, when taking into account heterogeneous architectures in the experiments, a finer control of the granularity of the computational tasks is needed. Some references for benchmarking dense linear algebra kernels are described in [VD08] and show that efficiency could be obtained on GPU devices only on relatively large blocks (a limited number of such blocks can be found on a supernodal factorization only on top of the elimination tree). Similarly, the amalgamation algorithm [HRR08], reused from the implementation of an incomplete factorization, is a crucial step to obtain larger supernodes and efficiency on GPU devices. The default parameter for amalgamation has been slightly increased to allow up to 12% more fill-in to build larger blocks, increasing the power efficiency, while maintaining a decent memory consumption.

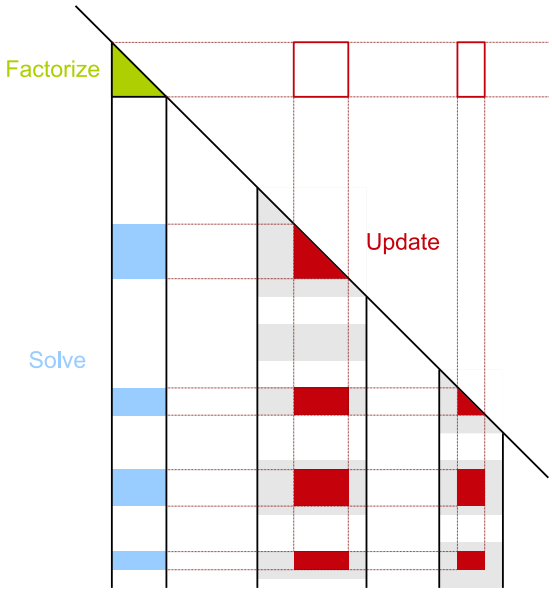
2.2.1 PaRSEC implementation

Listings 3 and 4 show our JDF representation of the sparse Cholesky factorization using the previously described `panel factorization` and `GEMM updates` tasks.

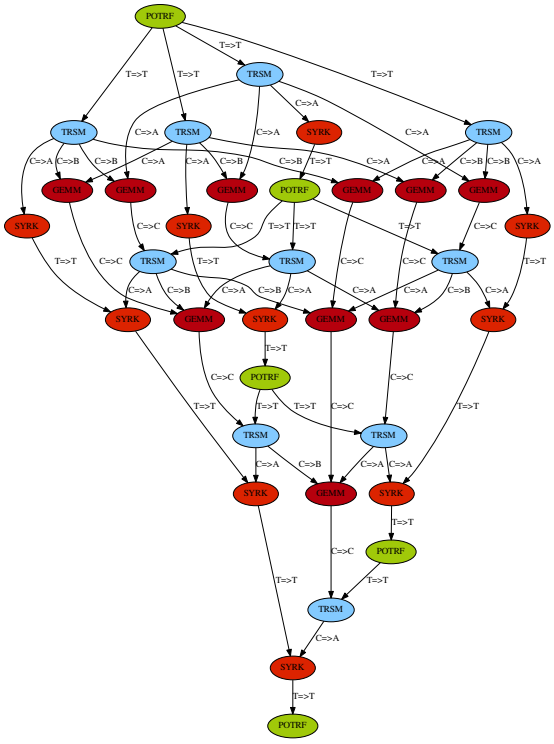
On line 3 of `panel(j)`'s JDF, `cblknbr` is the number of block columns in the Cholesky factor. Once the j-th panel is factorized, the trailing sub-matrix can be updated using the j-th panel. This data dependency of the sub-matrix update on the panel factorization is specified on line 16, where `firstblock` is the block index of the j-th diagonal block, and `lastblock` is the block index of the last block in the j-th block column. The output dependency on line 17 indicates that the j-th panel is written to



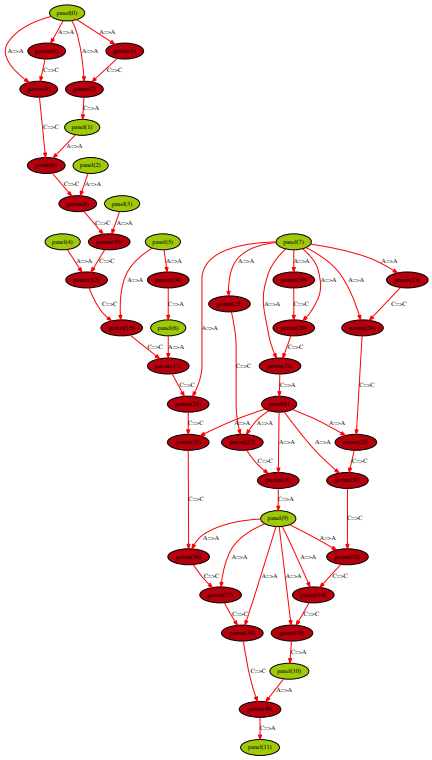
(a) Dense tile task decomposition.



(b) Decomposition of the task applied while processing one panel.



(c) Dense DAG representation.



(d) Sparse DAG representation of a sparse LDL^T factorization.

Figure 2.2 – Comparison of dense and sparse task decomposition.

Listing 3 Panel factorization JDF description.

```
1  panel(j)
2  /* Execution Space */
3  j = 0 .. cblknbr-1
4
5  /* Local variables */
6  leaf       = inline_c %{ return is_leaf(j); %}
7  lastbrow   = inline_c %{ return last_block_row(j); %}
8  firstblock = inline_c %{ return firstblock(j); %}
9  lastblock  = inline_c %{ return lastblock(j); %}
10
11 /* Task Locality (Owner Compute) */
12 :A(j)
13
14 /* Data dependencies */
15 RW A <- ( leaf ) ? A(j) : C gemm( lastbrow )
16      -> A gemm(firstblock+1 .. lastblock)
17      -> A(j)
18
19 /* Kernel function */
20 BODY
21   panel(j);
22 END
```

memory at the completion of the panel factorization. The input dependency of the j-th panel factorization is specified on line 15, where `leaf` is *true* if the j-th panel is a leaf in the elimination-tree and `lastbrow` is the index of the last block updating the j-th panel. Hence, if the j-th panel is a leaf, there are no updates to wait for, and the panel is directly read from memory. Otherwise, the input is the result of the last update task performed on the panel.

Similarly, `gemm(k)` (see listing 4) updates the `fcblk`-th block column using the k-th block, where `fcblk` is the index of column block facing the k-th block (i.e. which diagonal block is in the same block row), and `blocknbr` on line 3 is the number of blocks in the Cholesky factorized matrix. The input dependencies of `gemm(k)` are specified on lines 17 and 18, where the `cblk`-th panel `A` is being used to update the `fcblk`-th column `C`. Specifically, on these lines:

- `diag` is *true* if the k-th block is a diagonal block, and it is *false* otherwise;
- `first` is *true* if the k-th block is the first block to contribute to the $fcblk$-th panel;

PaRSEC does not provide reduction operators yet. Thus, it forces us to impose an order to apply the updates. In this sequence, `prev` and `next` are respectively the indices of the previous and the next block updating the `fcblk`-th panel. Solutions with control dependencies could bypass this restriction to restore an out of order scheduling of the updates, but those solutions were not compatible with the use on a heterogeneous architecture. Consequently, the data dependencies of the `gemm(k)` task are resolved once the `cblk`-th panel is factorized, and all previous updates are performed. The `fcblk`-th column is updated using the `prev`-th block. The diagonal blocks are not used to update the trailing sub-matrix, but it is included in the code to have a continuous space of execution for the task. This is required by the abridger representation of PaRSEC. In that case, we use `A(fcblk)` for both inputs as it is always available locally, and anyway the task will do nothing. Finally, lines 19 through 21 specify the output dependencies of the `gemm(k)` task. If this is the last update to the `fcblk`-th panel, then `next` is null and the panel is forwarded to the factorization task. Otherwise, the panel is released to the next update in the precomputed order.

This algorithm can be seen as a left-looking variant for the inputs, and right looking for the outputs.

2.2.2 StarPU implementation

The pseudo-code presented in algorithm 5 shows the StarPU tasks submission loop for the Cholesky decomposition. The submission of the tasks follows the sequential algorithm. StarPU uses the task insertion order to discover the dependencies between the tasks using the same data.

It is also possible to submit only ready tasks, as it would be the case using PaRSEC and use nested task submission inside tasks. This reduces the size of the DAG the runtime system must maintain, but gives it less information about the DAG at any given time. In that case, the panel factorization kernel of a supernode S_1 submits all the `GEMM`

Listing 4 Trailing sub-matrix update JDFdescription.

```
1  gemm(k)
2  /* Execution space */
3  k = 0 .. blocknbr
4
5  /* Local variables */
6  fcblk = inline_c %{ return facing_cblknum(k); %}
7  cblk  = inline_c %{ return cblknum(k); %}
8  diag  = inline_c %{ return is_diag(k); %}
9  prev  = inline_c %{ return prev_in_row(k, fcblk); %}
10 next  = inline_c %{ return next_in_row(k, fcblk); %}
11 first = inline_c %{ return prev_in_row(k, fcblk) == 0; %}
12
13 /* Task Locality (Owner Compute) */
14 :A(fcblk)
15
16 /* Data dependencies */
17 R  A <- diag ?  A(fcblk) : A panel(cblk)
18 RW C <- first ? A(fcblk) : C gemm(prev)
19      -> (!diag && next == null) ? A panel(fcblk)
20      -> (!diag && next != null) ? C gemm(next)
21
22
23 /* Kernel function */
24 BODY
25 if (!diag)
26    if( EXECUTION_ON_GPU) {
27      sparse_gemm_update_gpu(k);
28    } else {
29      sparse_gemm_update_cpu(k);
30    }
31 END
```

Algorithm 5 StarPU tasks insertion algorithm.

1: **For** all Supernode S_1 **Do**
2: $submit_panel(S_1)$ ▷ update of the panel
3: **For** off-diagonal block B_i of S_1 **Do**
4: $S_2 \leftarrow supernode_in_front_of(B_i)$
5: $submit_gemm(S_1, S_2)$ ▷ sparse GEMM $B_{k,k\geq i} \times B_i^T$ subtracted from S_2
6: **End For**
7: **End For**

updates using S_1. The `GEMM` task's kernel submits the factorization of the output panel if there is no other update to perform on it. This requires the introduction of an update counter per column block, as it is already done in PaStiX original implementation. An additional conditional wait is also required. Indeed, the main thread has to wait for all the tasks to be submitted before calling `starpu_wait_for_all()`. This is done using a pair of `pthread_cond_wait()/pthread_cond_signal()` where the main thread waits for the condition to be set by the last panel factorization. A quick study of the two different implementations showed no significant performance differences.

Thus, with PaRSEC and StarPU we can express the same task parallelism in a different way. At the moment of the experimentation, the two schedulers did not provide the same capabilities. For example, PaRSEC respects data locality while tasks can be commutable using StarPU.

2.2.3 Multi-core Architecture experimentation

In this thesis, we present the extensions to the solver to support heterogeneous many-core architectures. These extensions were validated through experiments conducted on Mirage nodes from the PlaFRIM cluster at Inria Bordeaux - Sud-Ouest. A Mirage node is equipped with two hexa-core Westmere Xeon X5650 (2.67 GHz), 32 GB of memory and 3 Tesla M2070 GPUs. The theoretical peak of one node reach 128.16 GFlop/s on this machine. PaStiX was built without MPI support using GCC`4.6.3`, CUDA `4.2`, Intel MKL `10.2.7.041`, and Scotch `5.1.12b`. Experiments were performed on a set of nine matrices, most of them parts of the University of Florida sparse matrix collection [DH11] (`matr5`, and `PmlDF` are respectively coming from University of Minnesota, and University of Cambridge). Their main characteristics are given in Table 2.1. These matrices represent different research fields and exhibit a wide range of properties (size, arithmetic, symmetry, definite problem, etc.). The TFlop column reports the number of floating point operations (Flop) required to factorize those matrices. Those numbers are used to compute the performance results shown in this section. The Number of tasks is computed for a shared memory run with twelve threads, minimal and maximal block sizes respectively set to 60 and 120, and minimal amalgamation level during graph preprocessing in Scotch set to 20. The total number of task is displayed and the number of column block factorization is between brackets. The number of sparse `GEMM` tasks can then be deducted by subtracting these two numbers.

As mentioned earlier, the PaStiX solver has already been optimized for distributed clusters of NUMA nodes [Fav09]. We use the current state-of-the-art PaStiX scheduler as a basis, and compare the results obtained using the StarPU and PaRSEC task-based runtime systems from there.

In this experiment, we do not expect to outperform the original PaStiX dedicated scheduler. However, we want to estimate the possible loss of performance introduced by using a generic task-based runtime system.

Figure 2.3 reports the results from a strong scaling experiment, where the number of computing resources varies from one to twelve cores, and where each group represents a particular matrix. Empty bars correspond to the PaStiX original scheduler, shaded

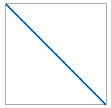
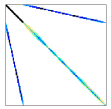
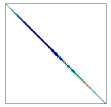
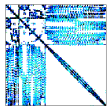
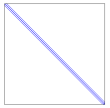
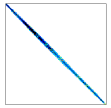
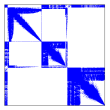
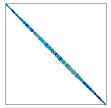
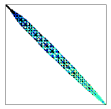
Matrix	Prec	Method	Size	nnz_A	nnz_L	Field	Profile	TFlop	Tasks number (panel))
`af_shell10`	D	LU	1.5e+6	27e+6	610e+6	Sheet metal forming		0.12	83 989 (13 970)
`dielFilterV2clx`	Z	LU	0.6e+6	12e+6	536e+6	High-order vector finite element method in electromagnetic		3.6	264 383 (11 289)
`Flan_1565`	D	LL^T	1.6e+6	59e+6	1712e+6	3D model of a steel flange, hexahedral finite elements		5.3	311 494 (14 986)
`audikw_1`	D	LL^T	0.9e+6	39e+6	1325e+6	structural problem		6.5	338 493 (10 082)
`matr5`	D	LU	0.5e+6	24e+6	1133e+6	Magneto-hydrodynamic		6.6	189 661 (9292)
`Geo_1438`	D	LL^T	1.4e+6	32e+6	2768e+6	geomechanical model of earth crust with underground deformation		23	608 356 (15 100)
`PmlDF`	Z	LDL^T	1.0e+6	8e+6	1105e+6	Accoustic		28	1 047 556 (30 588)
`Hook_1498`	D	LU	1.5e+6	31e+6	4168e+6	3D model of a steel hook with tetrahedral finite elements		35	561 323 (21 519)
`Serena`	D	LDL^T	1.4e+6	32e+6	3365e+6	gas resevoir simulation for CO2 sequestration		47	683 337 (15 515)

Table 2.1 – Description of the matrices (Z: double complex, D: double real).

bars correspond to StarPU, and filled bars correspond to PaRSEC. The performance results are in Flop/s, the higher the better. Overall, this experiment shows that on a shared memory architecture the performance obtained with all of the above-mentioned approaches is comparable, the differences remaining minimal on the target architecture. We could reach respectively 57.6%, 63.7%, and 68.9% of the peak performance with StarPU, PaStiX original scheduler, and PaRSEC in double precision.

Figure 2.4 shows results on `audi_kw1` test case with a larger number of cores. The experiment was performed on Romulus machine at ICL laboratory at UTK (University of Tennessee, Knoxville). The machine comprises four 12-core AMD Opteron 6180 SE at 2.5GHz with 256 GB RAM. One can notice that the scaling of the task-based runtime systems implementation compared to the original scheduler one is slightly reduced when the number of cores becomes greater than 12. This is due to the increasing overhead in the runtime scheduler when the number of computing units increases.

One can see that, in most cases, the PaRSEC implementation is more efficient than StarPU, especially when the number of cores increases. StarPU shows an overhead on multi-core experiments attributed to its lack of cache reuse policy compared to PaRSEC and the PaStiX internal scheduler. A careful observation highlights the fact that both runtime systems obtain lower performance compared to PaStiX for LDL^T factorization on both `PmlDF` and `Serena` matrices. This comes from a different implementation of the kernels. Due to its single task per node scheme, PaStiX stores the $D_{k,k}L_{k,j}^T$ matrix in a temporary buffer which allows the update kernels to call a simple `GEMM` operation. On the contrary, both StarPU and PaRSEC implementations are using a less efficient kernel that performs the full $A_{i,j} = A_{i,j} - L_{i,k}D_{k,k}L_{k,j}^T$ operation at each update. Indeed, because of the extended set of tasks, the life span of the temporary buffer could cause large memory overhead.

In conclusion, using these generic runtime systems shows similar performance and scalability to the PaStiX internal solution on the majority of the test cases, while providing a suitable level of performance and a desirable portability, allowing for a smooth transition toward more complex heterogeneous architectures.

2.3 Heterogeneous systems

While obtaining an efficient implementation is one of the goals of the previous experiments, it is not the major one. The ultimate goal is to develop a portable software environment allowing for a smooth transition to accelerators, a software platform where the code is factorized as much as possible, and where the human cost of adapting the sparse solver to current and future hierarchical complex heterogeneous architectures remains consistently low. Building upon the efficient supernodal implementation on top of DAG based runtimes, we can more easily exploit heterogeneous architectures. The `GEMM` updates are the most compute-intensive part of the matrix factorization, and it is important that these tasks are offloaded to the GPU. We decide not to offload the tasks that factorize and update the panel to the GPU due to the limited computational intensity, in direct relationship with the small width of the panels. It is common in

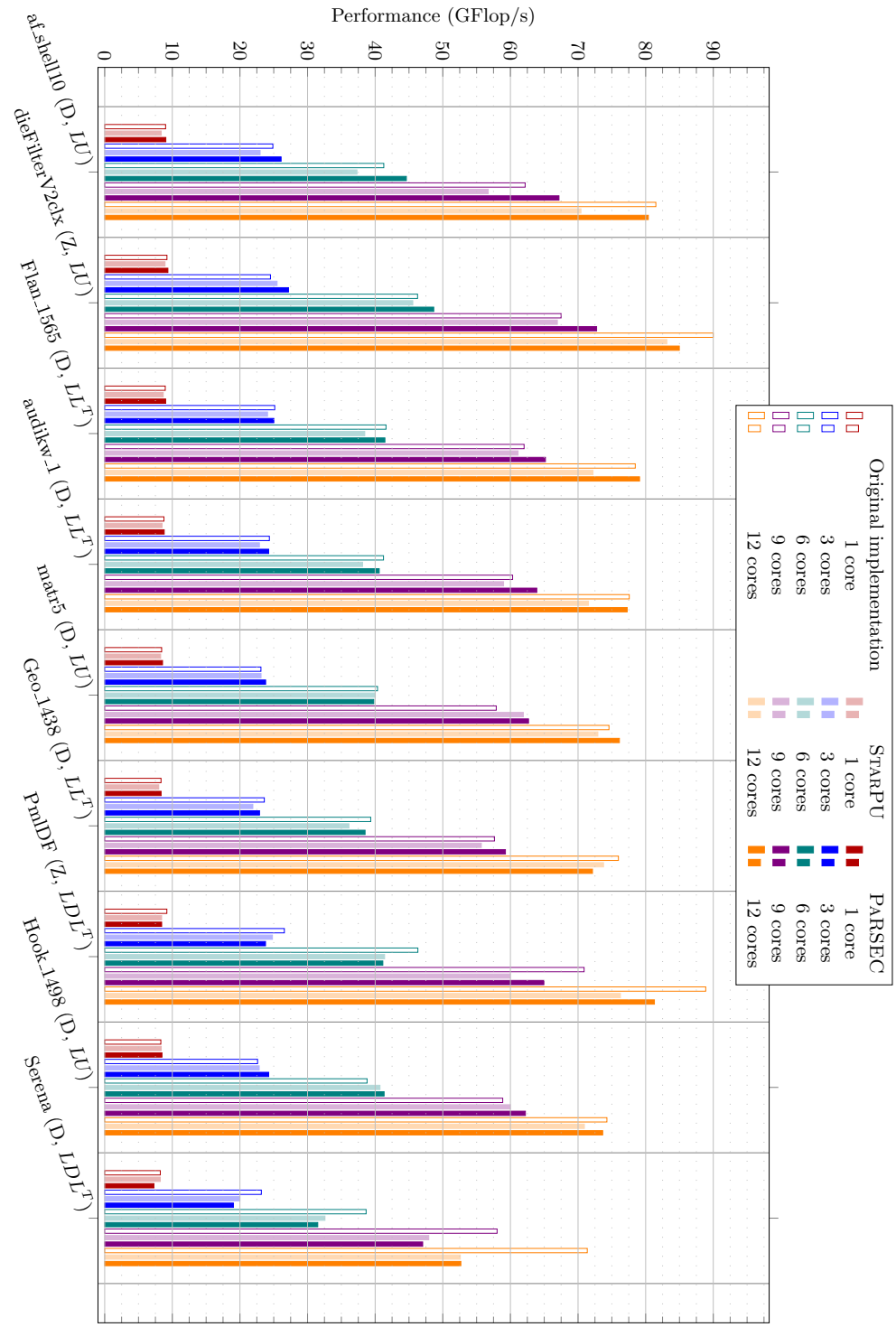


Figure 2.3 – CPU scaling study: GFlop/s performance of the factorization step on a set of nine matrices with the three schedulers.

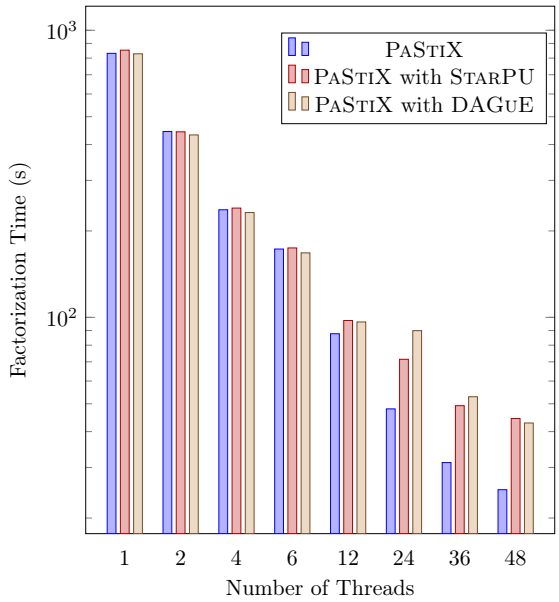


Figure 2.4 – CPU scaling study: Time to factorize on `audikw_1` test case on Romulus machine.

dense linear algebra to use the accelerators for the update part of a factorization while the CPUs factorize the diagonal blocks; so from this perspective our approach is conventional. However, such an approach, combined with look-ahead techniques, gives really good performance for a low programming effort on the accelerators [YTD12]. The same solution is applied in this study, since the panels are split during the analysis step to fit the classic look-ahead parameters.

When working on CPU the `GEMM` product corresponding to an update from a panel to another one is performed once, in a temporary buffer, and scatter added into the target column block. Indeed, performing a larger `GEMM` update increases the performance of the update. On a GPU we also want to perform the largest matrix product possible. However, we do not want to use a temporary buffer on the limited memory of the accelerator nor to multiply the small kernel addition calls as CUDA start-up would be paid for each one of them. Thus, we introduce a new kernel to perform the update at once on the GPU. This start-up cost could be reduced by using batch kernels submission on GPU but this is only available for identical task execution where only the data change while the sparse linear algebra decomposition produces very irregular tasks. The next subsection presents the new "sparse" GPU kernel that performs the trailing supernodes updates. Then, it compares the performance of this kernel with the original dense one to estimate the performance drop resulting from the sparsity of the update. After that we describe how we indicate the scheduler which update will be performed on GPU. Finally, we present a scaling study of the two runtime systems implementations with accelerators on a single node.

2.3.1 Implementation of a specific GEMM kernel

It is a known fact that the update is the most compute intensive task during a factorization. Therefore, generally speaking, it is paramount to obtain good efficiency on the update operation in order to ensure a reasonable level of performance for the entire factorization. Because of the embarrassingly parallel architecture of the GPUs and to the extra cost of moving the data back and forth between the main memory and the GPUs, it is of greatest importance to maintain this property on the GPU.

As presented in Figure 2.2(b), the update task used in the PaStiX solver groups together all outer products that are applied to a same panel. On the CPU side, this GEMM operation is split in two steps because of the gaps in the destination panel: the outer product is computed in a contiguous temporary buffer, and upon completion, the result is scattered on the destination panel. This solution has been chosen to exploit the performance of vendor provided BLAS libraries in exchange for constant memory overhead per working thread.

For the GPU implementation, the requirements for an efficient kernel are different. First, a GPU has significantly less memory compared with what is available to a traditional processor, usually between 3 to 6 GB that can be used by multiple kernels simultaneously. This forces us to carefully restrict the amount of extra memory needed during the update, making the temporary buffer used in the CPU version unsuitable. Second, the uneven nature of sparse irregular matrices might limit the number of active computing units per task. Hence, only a partial number of the available warps on the GPU might be active, leading to a deficient occupancy. Thus, we need the capability to submit multiple concurrent updates in order to provide the GPU driver with the opportunity to overlap warps between different tasks to increase the occupancy, and thus, the overall efficiency.

Many CUDA implementations of the dense GEMM kernel are available to the scientific community. The most widespread implementation is provided by NVIDIA itself in the cuBLAS library [NVI08]. This implementation is extremely efficient since CUDA 4.2, allows for calls on multiple streams, but is not open source. Volkov developed an implementation for the first generation of CUDA enabled devices [VD08] in real single precision. In [Tan+11], authors propose an assembly code of the DGEMM kernel that provides a 20% improvement on cuBLAS 3.2 implementation. The MAGMA library proposed a first implementation of the DGEMM kernel [NTD10] for the NVIDIA Fermi GPUs. Later, an auto-tuned framework, BEAST (Bench-testing Environment for Automated Software Tuning) from ICL, was presented in [KTD12] and included into the MAGMA library. This implementation, similar to the ATLAS library for CPUs, is a highly configurable skeleton with a set of scripts to tune the parameters for each precision.

As our update operation is applied on a sparse representation of the panel and matrices, we cannot exploit an efficient vendor-provided GEMM kernel in a single call per panel. We need to develop our own, starting from a dense version, and altering the algorithm to fit our needs. Because of the source code availability, the coverage of the four floating point precisions, and its tuning capabilities, we decided to use the

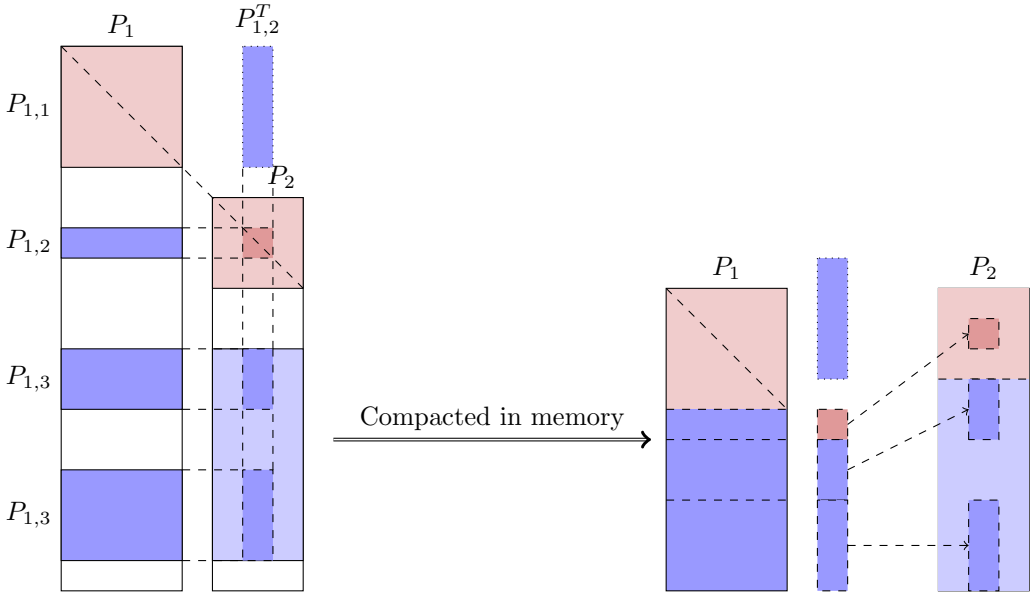


Figure 2.5 – Description of the panel update task. On the right we see the panel with the non allocated zeros represented in white. On the left the panels are represented as they are stored in memory. Each column block is a dense memory area in column-major order. Red color represent the diagonal blocks and blue off-diagonal blocks.

BEAST-based version for our *sparse* implementation. As explained in [KTD12] the matrix-matrix operation is performed in two steps in this kernel. Each block of threads computes the outer-product $tmp = AB$ into the GPU shared memory, then the addition $C = \beta C + \alpha tmp$ is computed. To be able to compute the result of the update from one panel to another directly into C, we altered the kernel to provide the structure of each panel to the kernel. This allows the kernel to compute the correct position inside C during the *sum* step. This introduces a loss in the memory coalescence and deteriorates the update parts. However, it prevents the requirement of an extra buffer on the GPU for each offloaded kernel. Figure 2.5 describes how a panel receives an update from another panel. On the picture, P_1 updates P_2 by deducting from it the result of the product $P_{1,j}.P_{1,2}^T, j \geq 2$ in the corresponding memory area of P_2. The right part of the Figure presents how the update is performed on a CPU. The product is first computed in a temporary buffer with a `GEMM` operation, then it is scatter-subtracted from P_2 using multiple BLAS matrix-matrix additions. Figure 2.6 describe the execution of our modifications to the original BEAST kernel. In that case, everything is performed in one kernel call. First, each tile of the product is computed in the shared memory then the result is deducted, using a computed offset, in the target panel. Our contribution here is to use the offset to directly update the panel in one call and without additional temporary buffer.

The BEAST kernel has been tuned in MAGMA library with texture usage enabled,

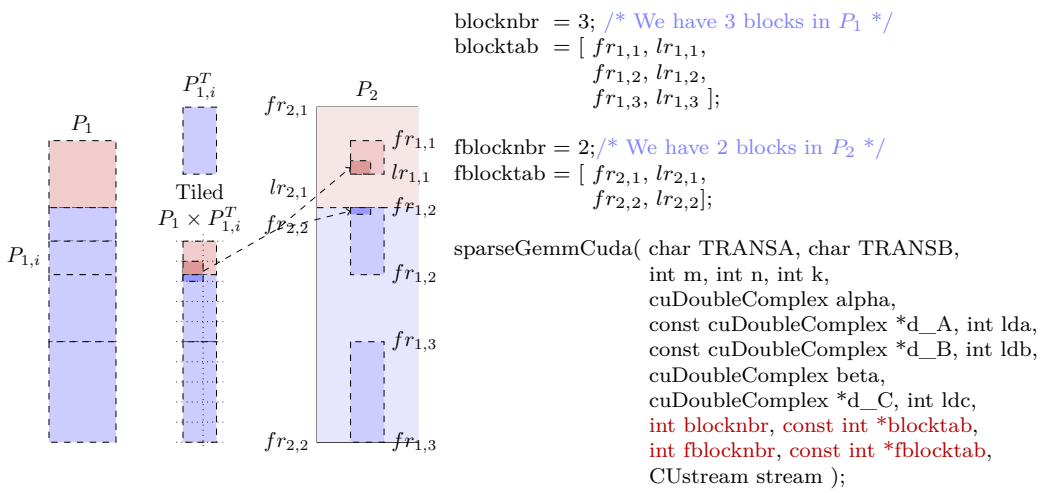


Figure 2.6 – Description of the GPU sparse `GEMM` update operation. fr (resp. lr) stands for *first row* (resp. *last row*). Four parameters – in red – are added to describe the sparse structures.

which gives the best performance. Our problem is that the functions `cudaBindTexture` and `cudaUnbindTexture` are not compatible with concurrent kernel calls on different streams. Therefore, the textures have been disabled in the kernel, reducing the performance of the kernel by about 5% on large square matrices, and by a larger factor on irregular sizes.

Figure 2.7 and Figure 2.8 show the study made on the GPU `GEMM` kernel and the effect of the modifications done on the BEAST kernel. These experiments are done on a single GPU of the *Mirage* cluster (NVIDIA M2070). The experiment consists of computing a representative matrix-matrix multiplication of what is typically encountered during sparse factorization in double precision. Each point is the average performance of 100 calls to the kernel that computes: $C = C - AB^T$, with A, B, and C, matrices respectively of dimension M-by-K, N-by-K, and M-by-N. B is taken as the first block of K rows of A as it is the case in Cholesky factorization. M, N and K are set according to the average values encountered as presented in subsection 2.4.2. N being the height of the first block (and the width of the updated panel), it has been set to 32. K is the width of the first panel and has been set to 96. M varies from 192 to 9600. In Figure 2.7, we first compare the lost of performance of our sparse kernel against a single dense kernel. For the sparse kernel, blocks in A are also randomly generated, with an average eight of 32, with the constraint that the rows interval of a block of A is included in the rows interval of one block of C, and no overlap is made between two blocks of A. The plain lines are the performance of the CUBLAS library with one stream (*purple*), two streams (*green*), and three streams (*blue*). The black line represents the

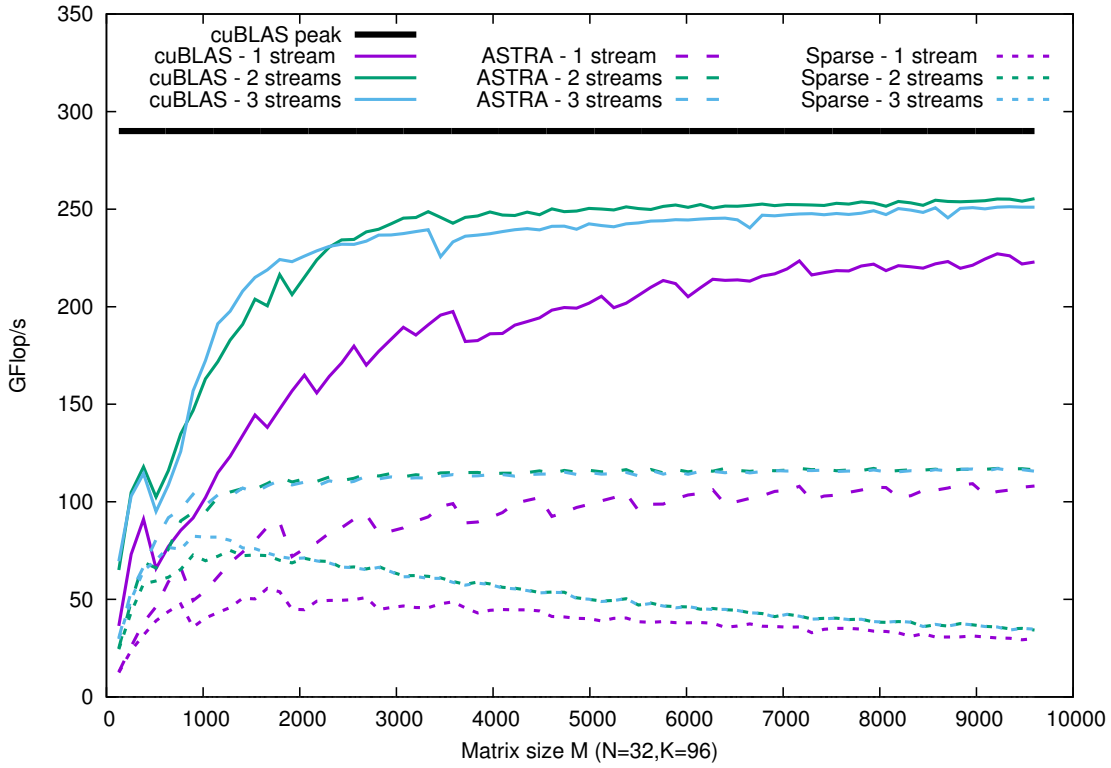


Figure 2.7 – Multi-stream performance comparison, on a M2070 GPU, on the `DGEMM` kernel for three implementations: cuBLAS (one dense `GEMM`) library, BEAST (one dense `GEMM`) framework, dense operations, without taking into account the sparsity of the matrix, and our sparse adaptation of the BEAST framework with additional offset to the C updates.

peak performance obtained by the cuBLAS library on square matrices. This peak is never reached with the particular configuration case studied here because of the tall and skinny shape of A and C that decreases the ratio of computations per data. The dashed lines are the performance of the BEAST library in the same configurations. We observe that this implementation already loses 140GFlop/s, around 60%, against the cuBLAS library in that configuration. Removal of texture and design for square matrices even if tuned for rectangular one are responsible for this performance loss. Finally, the dotted lines illustrate the performance of the modified BEAST kernel to include the gaps into the C matrix. It highlights the loss of performance due the memory fragmentation and the additional tests. Introducing the sparsity, 50 GFlop/s more are lost. We observe a direct relationship between the height of the panel and the performance of the kernel: the taller the panel, the lower the performance of the kernel. The memory loaded to do the outer product is still the same as for the BEAST curves, but memory loaded for the C matrix grows twice as fast without increasing the number of Flop to perform. The ratio Flop per memory access is dropping and explains the decreasing performance. However, when the factorization progresses and moves up the elimination trees, nodes get larger and the real number of blocks encountered is smaller than the one used in this experiment to illustrate worst cases.

Another alternative solution to the implementation of a new kernel would have been to call a dense kernel for each block of contribution. In Figure 2.7, cuBLAS and BEAST were given a single large block to update, whereas Sparse was given an update to a matrix of the same size, but which has been split into a randomly generated set of smaller non-contiguous blocks. In Figure 2.8, the Sparse kernel performs the same operation (on a set separate blocks) as in Figure 2.7, but cuBLAS and BEAST are now being used to compute on the same set of split blocks. This is a fairer comparison of the three kernels, since this computation mimics what occurs in a supernodal update show in Figure 2.5. Now that all three kernels are performing the same task on the same data structure, we see that the sparse kernel implementation is ten to twenty times more efficient than the solution multiplying the calls to `GEMM` routine.

Without regard to the kernel choice, it is important to notice how the multiple streams can have a large effect on the average performance of the kernel. For this comparison, the 100 calls made in the experiments are distributed in a round-robin manner over the available streams. One stream always gives the worst performance. Adding a second stream increases the performance of all implementations and especially for small cases when matrices are too small to feed all resources of the GPU. The third one is an improvement for matrices with M smaller than 1000, and is similar to two streams when greater than 1000.

Our modified kernel is the one we provide to both runtime systems to offload computations on GPUs. An extension of the kernel to handle the LDL^T factorization has also been developed. It takes an extra parameter to the diagonal matrix D and computes: $C = C - LDL^T$ with a 5% additional cost.

Whereas PaRSEC requires the `GEMM` tasks to be chained (2.9(a)) more parallelism

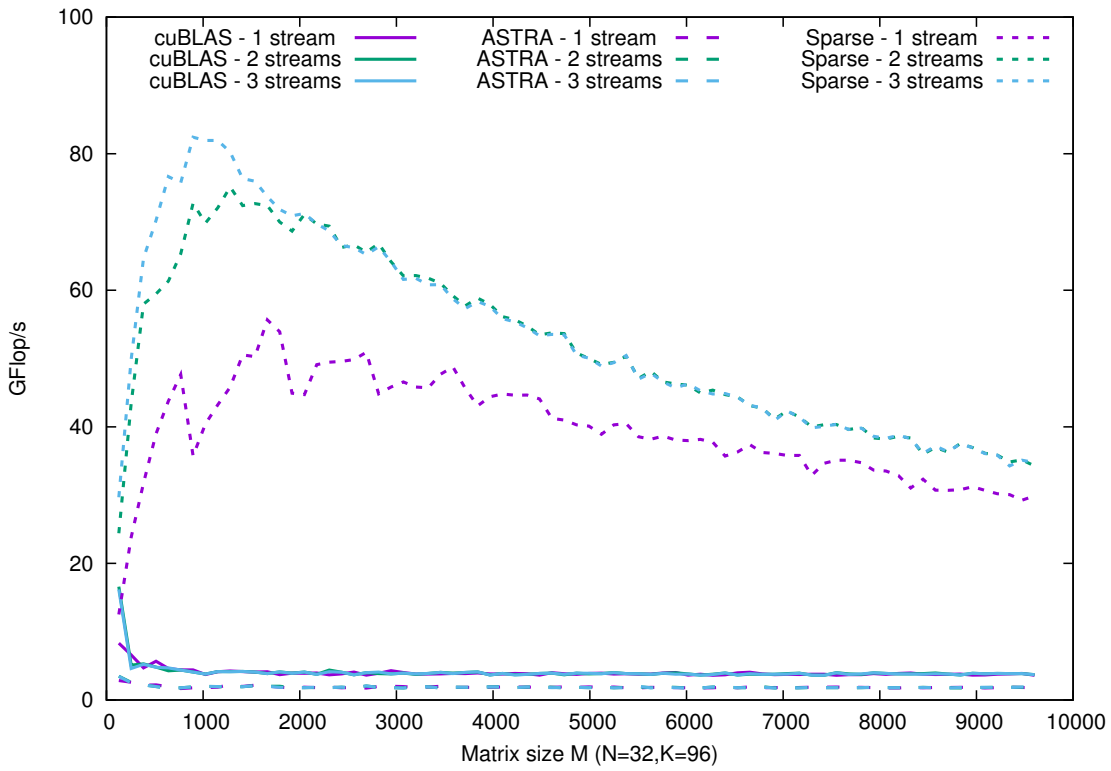
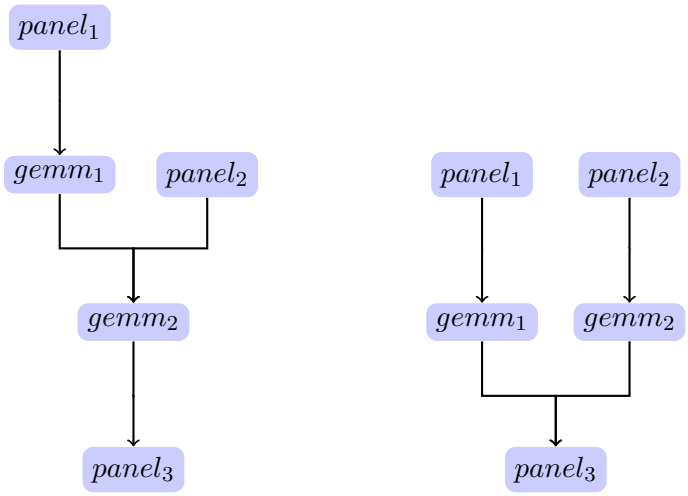


Figure 2.8 – Multi-stream performance comparison, on a M2070 GPU, on the `DGEMM` kernel for three implementations: cuBLAS (mutiple dense `GEMM`s) library, BEAST (multiple dense `GEMM`s) framework, with the sparsity taken into account, and our sparse adaptation of the BEAST framework with additional offset to the C updates.



(a) without commutable GEMMs (b) with commutable GEMMs

Figure 2.9 – Effects of commutable tasks on the graph.

can be exposed (2.9(b)) using the STARPU_COMMUTE option provided by StarPU to allow two tasks targeting the same data to be executed in an undefined order.

The GEMM updates could also be considered as a reduction operation, but this possibility is not used because it would imply more operations. Indeed, a reduction would imply adding together all coefficients of each copies of the whole panel whereas the GEMM panel updates affect only a small part of the panels. Moreover, a reduction requires a copy of the destination supernode on each computing device, which could represent a large memory overhead. These commutable tasks allow only an out of order execution of the updates while guaranteeing the mutual exclusion of the computations, and the consistency of the data on all devices. PaRSEC in its actual release does not allow those commutable operations and keep the order defined by the dependencies. As the order of the reduction is imposed by the data dependencies, the potential concurrency is reduced but the accuracy of the numerical results is maintained among successive runs.

2.3.2 Data mapping over multiple GPUs

The two considered runtimes handle the tasks distribution over the heterogeneous resources differently. PaRSEC was originally designed for dense algorithms such as dense Cholesky or QR factorizations. In these very regular algorithms, GEMM updates can be considered as reductions on each tile of the matrix. To avoid data movement, PaRSEC, in its simple heuristic, maps all the GEMM operations involved in one reduction to a single computing unit (one GPU or all CPUs). To decide where to map each reduction, PaRSEC relies on a dynamic decision based on a counter of load for each unit when the first operation of a reduction is triggered. PaRSEC considers the number of Flops involved in the whole reduction and the known performance of the considered accelerator. If the

computation of the number of Flops received by a tile is easy in dense algorithm, it is more costly to compute the number of Flops received by a panel at runtime in sparse algorithms. Moreover, the performance on the GPUs of the irregular small sparse `GEMM`s operations that occur in sparse algorithm is not known. For this reason, we decide to compute a static mapping of the `GEMM` updates at analysis step. The static mapping could lead to deadlocks as PaRSEC 1.0 has no LRU (Least Recently Used: algorithm used by the runtime to discard in priority least recently used data from cache) to remove unused written data from the GPU before the last update has been performed, and requires to keep the written data on GPU until all contributions are received. Thus, there could be no memory left for read only data required to perform the updates and, in that case, the computation would be locked. Therefore, a memory constraint is required to limit the load of a GPU.

On the other side, the StarPU runtime can compute historically based prediction models and dynamically decide when to offload tasks to GPUs. Then, it can use an HEFT (Heterogeneous Earliest Finish Time) algorithm to schedule the tasks on the available computing units. However, the performance sampling build by StarPU on our sparse kernel is highly disrupted by noise and StarPU can not take good decisions. Moreover, StarPU takes the scheduling decision early in the task submission process and is not able to reconsider its decisions bad or good. Thus, we decided to use the same static mapping for StarPU as the one we had to use with PaRSEC.

The static mapping algorithm is a bin-packing algorithm were all column blocks are sorted following a given criterion where the load of a GPU is limited by the amount of memory it can store. Besides avoiding filling the whole GPU, this limit will reduce the data transfer by keeping data on the GPUs. Several criteria can be used:

***Surface of the target panel*:** This criteria corresponds to the memory occupied by the panel receiving the update. This will avoid filling the whole GPU with the small panel on the bottom of the elimination tree. The larger is the panel the more chances it has to receive updates;

***Number of updates received by the panel*:** This criteria corresponds to the number of `GEMM` applied to the panel;

***Number of Flops received by the panel*:** Here we consider the number of Flops involved in all the updates a column block will receive. Indeed, not only the number of update is important, but larger are those updates, the more Flops will be performed. This criterion is close to the one used in dense linear algebra algorithms except that we have no good prediction of the performance of the kernel in sparse linear algebra. Thus we cannot use the Flop/s as it would be used in dense, but only the Flops;

Position in the critical path: Here we want to accelerate the tasks that are more critical to the execution. Thus, we use the priority computed by PaStiX that corresponds to the position in the critical path. The higher is the priority, sooner the result is required, and accelerators can help providing them quickly.

A greedy bin packing algorithm dequeues the first panel according to the selected criteria and associates it with the less loaded GPU. When a panel is associated to a GPU, all the updates on this panel will be performed on the GPU. A verification is made to guarantee we do not exceed the memory capacity of the GPU to avoid excess use of the runtime LRU.

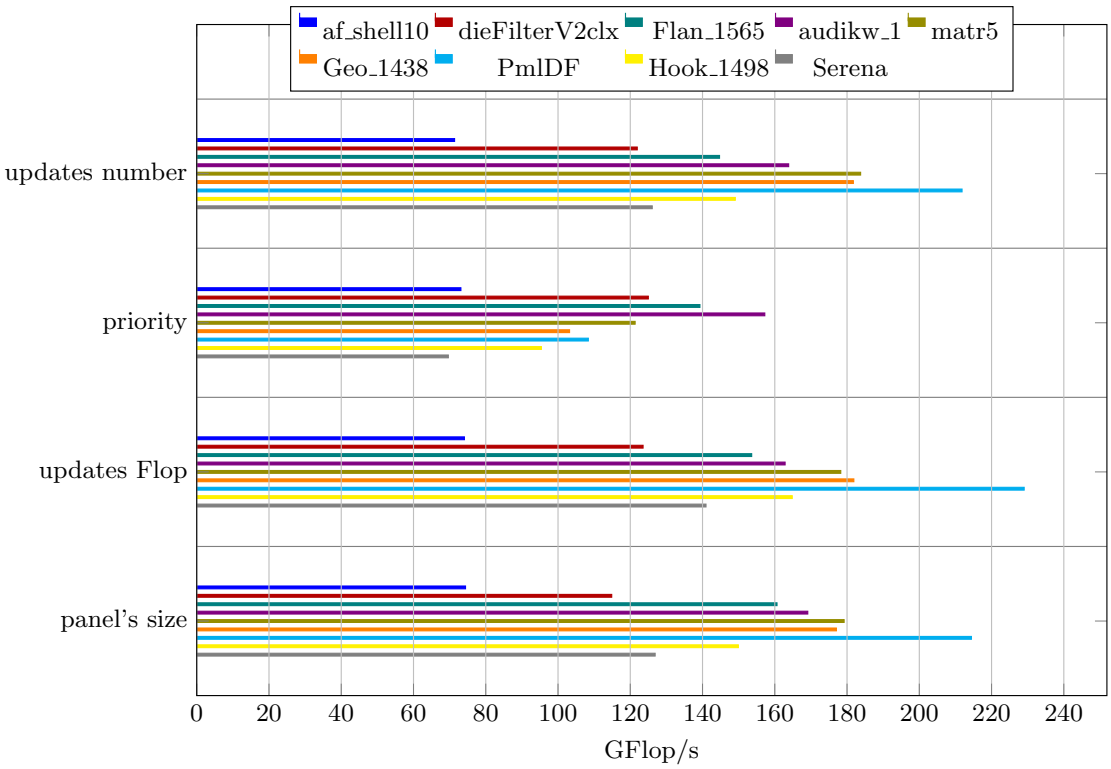


Figure 2.10 – Sort criteria study: Flop/s obtained with PARSEC, on 3 GPUs, with 3 streams.

Figure 2.10 compares Flop/s obtained with the different criteria on our set of matrices. Each color corresponds to a matrix, and the bars are grouped by criterion. The best performances were obtained using the Flops criterion. Indeed, on average, the priority criteria, which will offload small panels, from the leaves of the elimination tree, first produces less Flop/s than the others. Using the number of updates, the panel's size, and the number of Flops we obtain respectively 36%, 38%, and 42% more Flop/s. We could expect that except the priority criteria, all the other would have similar results. Indeed, the larger is the panel the more update it will received, and the number of Flops is also linked with the number of updates. In future experiments, we report only results with the Flops criterion.

2.3.3 Heterogeneous experiments

Figure 2.11 presents the performance obtained on our set of matrices on the Mirage platform by enabling the GPUs in addition to all available cores. The PaStiX run is shown as a reference. StarPU executions are represented with empty bars, PaRSEC runs with 1 stream are shaded, and PaRSEC runs with 3 streams are fully colored. This experiment shows that we can efficiently use the additional computational power provided by the GPUs using the generic runtime systems. In its current implementation, StarPU has either GPU or CPU worker threads. A GPU worker will execute only GPU tasks. Hence, when a GPU is used, a CPU worker is removed. With PaRSEC, no core is dedicated to a GPU, and CPUs might execute CPU tasks as well as driving GPU ones. The first computational threads that submit a GPU task takes the management of the GPU until no GPU work remains in the pipeline.

Both runtime systems manage to get similar performance and satisfying scalability over the three GPUs. For example, on `PmlDF`, using StarPU with 3 GPUs increases the performance by 2.86 against PaStiX (without GPU). On the same case, using PaRSEC we could obtain an acceleration of 2.35 with 3 GPUs and 2.65 when using streams. In only two cases, `matr5` and `PmlDF`, StarPU outperforms PaRSEC results with three streams. This experimentation also reveals that, as it was expected, the computation takes advantage of the multiple streams that are available through PaRSEC. Indeed, the tasks generated by a sparse factorization are rather small and will not use the entire GPU. The size of the task is common to any sparse linear algebra solver and using multiple streams could also be beneficial to previously cited studies about sparse factorization on heterogeneous architectures. This PaRSEC feature compensates for the prefetch strategy of StarPU (data used by the GPUs are loaded into GPUs at the beginning of the algorithm) that gave it the advantage when compared with the results with one stream. One can notice the poor performance obtained on the `af_shell10` test case: in this case, the amount of Flop (0.12 TFlop) produced is too small to efficiently benefit from the GPUs.

2.3.4 Memory study

Memory is a critical resource for direct solver, and using generic framework might lead to memory overhead. Figure 2.12 compares the memory peaks obtained with the three implementations of the solver. This information was obtained using the memory module of the EZTrace[7] [Tra+11] library which overloads the memory allocation routines to increment counters. The runs where obtained with 12 cores but the results would not be much different with another setup.

The memory allocated can be separated in three categories :

- the coefficients of the factorized matrix, which are allocated at the beginning of the computation and represent the largest part of the memory;
- Common data structures to the different implementations including:

[7] automatic execution trace generation tool (`http://eztrace.gforge.inria.fr/`)

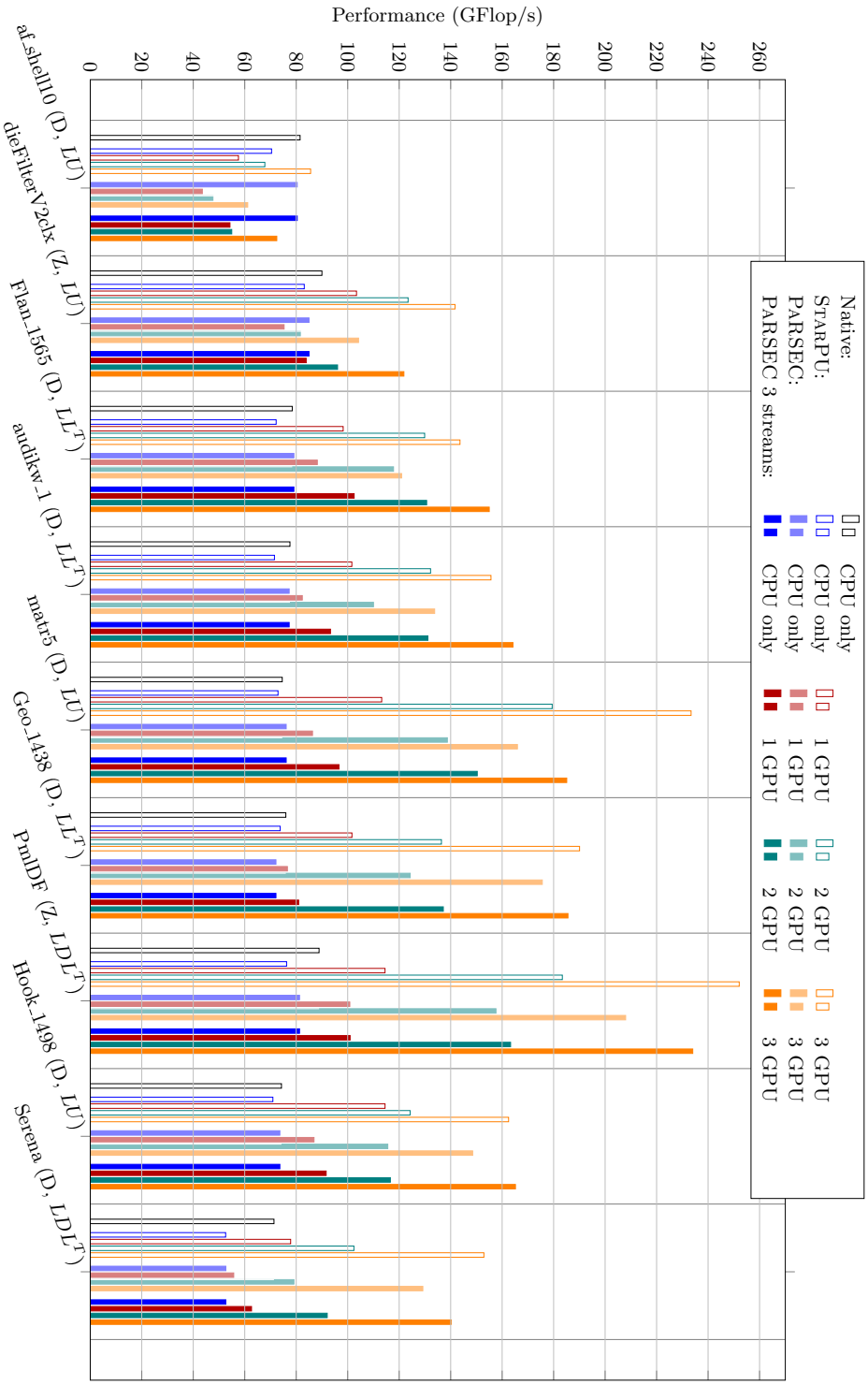


Figure 2.11 – GPU scaling study: GFlop/s performance of the factorization step with the three schedulers on a set of 10 matrices. Experiments exploit twelve cpu cores and from zero to three additional GPUs.

  - the structure of the factorized matrix describing the column blocks structure and built before the factorization is performed;
  - the user's CSC(Compress Sparse Column) matrix which is the input given to PaStiX that can be freed before factorization if user specifies it;
  - the internal block distributed CSC, which corresponds to the input matrix reordered and stored in a compressed block format. It is used to initialize the coefficients of the factorized matrix, to compute relative error, and in the matrix-vector products operation of the iterative refinement. It can be freed if the user specifies it and does not want to refine the solution inside PaStiX;

- the last part of memory that includes the selected scheduler overhead.

As shown in the plot, a large part of memory corresponds to the first four categories and is independent of the runtime used. The last part of the bars corresponds to the overhead of the scheduler.

The values on top of the bars are the overhead ratio compared with memory overhead obtained with PaStiX original scheduler. We can see that we obtained a small overhead with PaRSEC (half to three times the original scheduler overhead), whereas StarPU allocates about 7% more memory than PaStiX (three to four and a half time the original scheduler overhead). This runtime overhead is resulting from the DAG management, and all the internal structure required for ensure the data coherency. StarPU uses more memory than PaRSEC because it has to store the whole graph whereas PaRSEC only keeps the active part of the graph in memory. Compared to the whole memory allocated the task-based runtime overhead is affordable.

2.4 Optimizations

In this section, we present the different optimizations that led to the performance we obtained and showed in this chapter. First, we had to adapt the granularity of our blocks to the runtime systems to avoid generating too many small tasks and overload the runtime system. Then, we explain how we modified our block splitting algorithm which creates more parallel tasks but could decrease their computational intensity.

2.4.1 Task granularity adapted to the runtime

Another parameter to study in order to reduce the number of small tasks is the level of amalgamation used during graph partitioning, when minimum degree algorithm is used. During this phase, all column blocks smaller than a given size are amalgamated to their parent, creating bigger column blocks. The bigger column blocks we have, the smaller is the number of tasks to factorize the matrix but the more fill-in is created because of this amalgamation. Thus, the number of operations and the memory consumption increase with this amalgamation level while the efficiency of the blocked operations is increased. Then, it is a matter of compromise and cost models to decide if the amalgamation in beneficial or not.

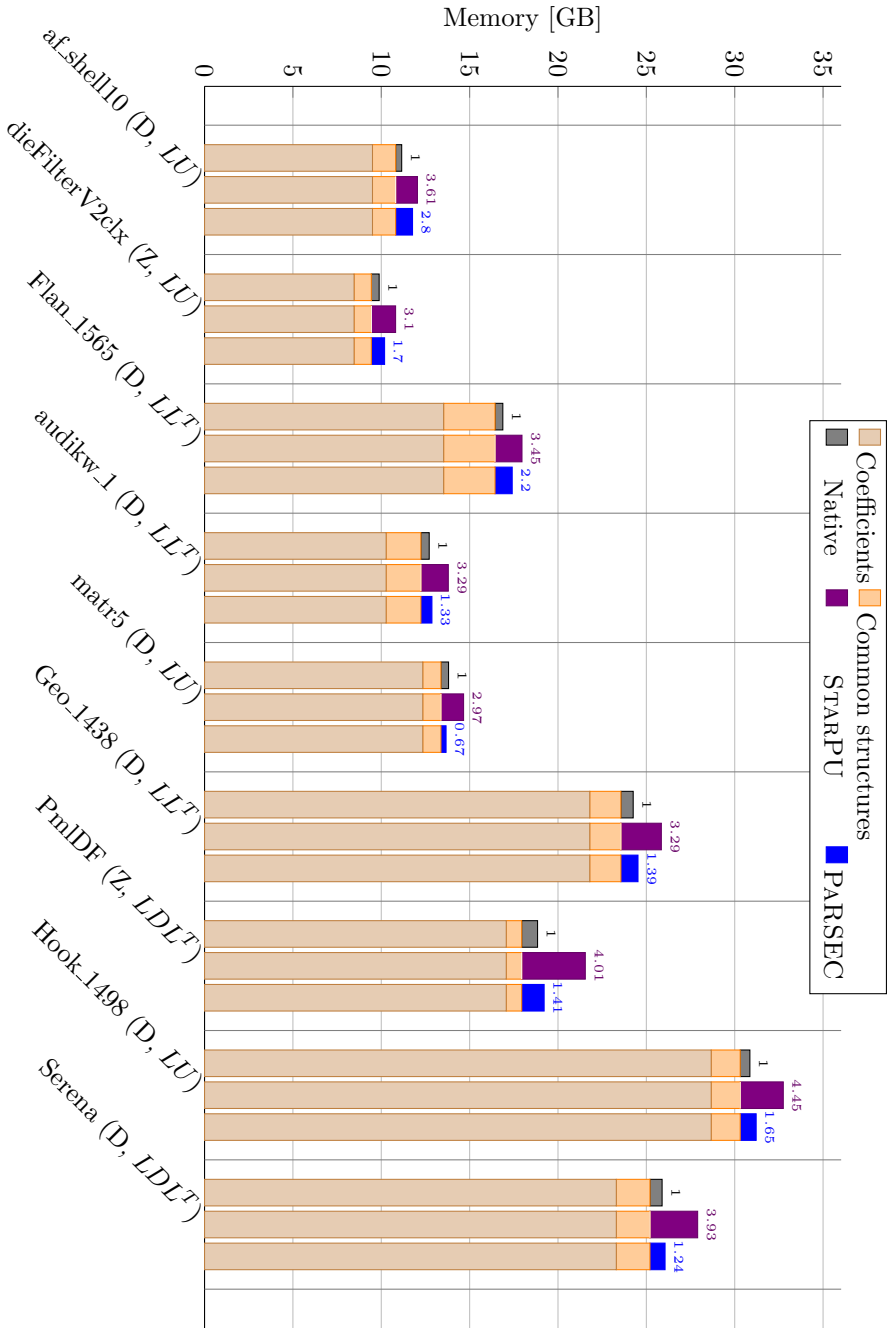


Figure 2.12 – Memory consumption comparison, overhead ratio is written on top of bar chart.

In order to reduce the runtime overhead the amalgamation threshold is increased to twenty which correspond to a good ratio for task-based runtime systems (see Figure 2.14).

2.4.2 Block splitting algorithm

During preprocessing, in order to create more parallelism, large panels are split into smaller ones that can be handled by different cores. Typically, the default behavior splits column blocks composed of more than 64 columns into smaller same-sized panels. This block column splitting also reduces the memory consumption, specifically with Cholesky algorithm. Indeed, when allocating a column block, the unused triangular upper part of the diagonal block is also allocated to simplify memory access as it is done in dense linear algebra libraries kernels. The larger are the block columns, the larger is the unused allocated area. Thus, splitting the column blocks reduces this memory overhead while increasing the concurrency.

In our algorithm, blocks have to fit the diagonal block in their row (i.e. the first row and last row of the block must be included in the first and last rows of the diagonal block). Thus, splitting a column block induces splitting of the blocks facing the diagonal block, and that can create tiny blocks (as shown in Figure 2.13(a)), which leads to non-efficient BLAS calls. On the given example, we have two blocks facing the diagonal block. It appears that splitting on the border of one of the facing block leads to bigger facing block splits than splitting using a constant split size. Also, fewer blocks are created and thus fewer tasks will be involved in the factorization algorithm, lightening the load on the scheduler.

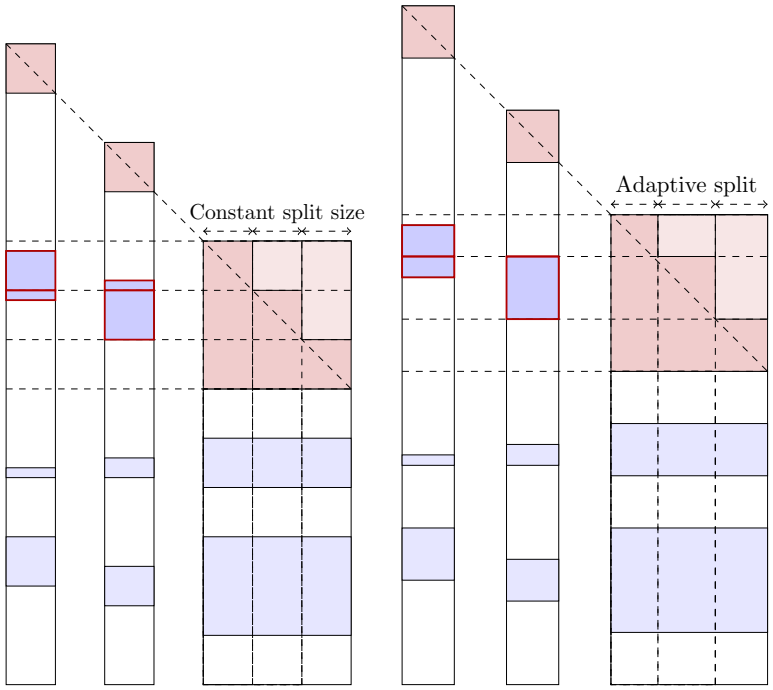
In order to control the creation of flat blocks when splitting column blocks, we updated the panel splitting algorithm (see algorithm 6 and Figure 2.13(b)). Using a variable criterion also leads to more efficient computation during factorization.
The variable size algorithm introduces the creation of an array (line 1), called $nbBlocksPerLine$. $nbBlocksPerLine[i]$ contains the number of blocks that would be split if a split occurs between line i and $i+1$.
Then, for each panel, an average block size is computed, and the actual split is made in a given interval around the mean value to minimize the number of blocks affected by the cut (line 9). This choice, of the splitting column/row, can follow several rules:

Constant split size: splits a block with $averageSize$ (or $maxSize$ if smaller than $averageSize$);

Maximum width: chooses the largest block minimizing the cut (Alg. 7);

Medium width: chooses a block minimizing the cut. If several cuts are possible the chosen one is either the closer larger than $averageSize$ or, if not found before, closer to and smaller than $averageSize$ (Alg. 8).

The second one will create larger blocks although the third one will keep panels' width closer to the computed average size. Once a panel has been split, the array



(a) Classical equal splitting (b) Adaptive splitting

Figure 2.13 – Comparison of panel splitting.

nbBlocksPerLine is updated as the number of column blocks has increased (line 15 to 19).

Table 2.2 shows a comparison of the different methods with several values of authorized intervals in the choice of the splitting column. For this experiment, we used the default minimal and maximal block splitting size parameters of respectively 60 and 120. All experiments were conducted using 16 threads on one node of 16 cores from the Curie cluster at the TGCC (Only the factorization time is machine dependant). They were performed using the `audikw_1` test case. One can expect similar results with the other matrices as the algorithm used is identical. One can notice that the larger the interval is, the smallest the number of column blocks (and also blocks) created is. The structure contains fewer small blocks and, thus, the performance of the factorization is increased. Even if the minimal and mean block height is not always maximal with *larger width split* method, the factorization time is improved. This is resulting from more efficient BLAS calls which lead to more Flop/s during factorization (see Figure 2.14). Using a 200% authorized variation one can get a 20% improvement in timing that corresponds to a 30% improvement in Flop/s. One can also notice that if the amalgamation level chosen for graph partitioning and supernode detection has nearly no effect on PaStiX scheduler, the runtime system is more sensitive to the size of the supernodes. Indeed, the larger is the amalgamation, the more efficient are the BLAS operations, but the lightweight original scheduler can compensate the less intensive operations with the fewer total number of operations involved when `cmin` amalgamation parameter used in Scotch is low.

Because of the increase of diagonal blocks size and thus, the storage of more upper (resp. lower) zero diagonal terms, the memory consumption also increases a little with the blocks' size. While increasing the liberty in the choice of the split, the obtained results get away from user's `IPARM_MAX_BLOCKSIZE` setup, changing this setup could also affect performance.

Figure 2.15 shows the distribution of the blocks. The blocks are gathered in 10 classes, each one representing 10% of the sizes and corresponding to a step in the curves. Curves are cumulative (i.e. the first step corresponds to the first 10% sizes, second to the first 20% and so on) . Most blocks are rather small in any case, but tend to decrease when we give more liberty to the splitting decision. Specifically, while the number of blocks larger than the maximum required size (120) increases, the number of blocks under the minimal size of (60) decreases. Many block sizes are around 60 rows because the default number of blocks during split is set to $blockwidth/60$.

Thus, using an intelligent block splitting algorithm can help improving the efficiency of the BLAS calls during factorization and, consequently, reducing factorization time. Using larger blocks can also be beneficial in terms of efficiency. The main drawback of this approach is a slight increase in memory consumption. A large percent, above 25%, of authorized variation is required to get a good improvement in the performances. This is even more relevant for task-based runtime systems implementations that are really penalized by low-Flop tasks.

To obtain an efficient execution of the factorization on top of tasks based runtime systems, one can set the `cmin` amalgamation parameter to 20 and an authorized variation

Type of split	Constant	Median block first					
Modulation percent	-	5	25	50	100	150	200
blocks number	523428	513277	500510	476937	396602	353541	321746
column blocks number	12009	12020	12033	11913	10579	10284	9956
Factorization time	52.5	52.9	51.6	50.2	45.9	43	42
Average block height	28.8	29.5	30.2	30.6	31.3	30.5	29.5
Minimal block height	1	3	3	3	3	3	3
Maximal block height	120	126	150	180	240	300	360
Average cblk width	78.6	78.5	78.4	79.2	89.2	91.8	94.8
Minimal cblk width	3	3	3	3	3	3	3
Maximal cblk width	120	126	150	180	240	300	360
Coeftab Size	9.5 GB	9.5 GB	9.5 GB	9.5 GB	9.5 GB	9.5 GB	9.6 GB

Type of split		Larger blocks first					
Modulation percent		5	25	50	100	150	200
blocks number		509470	494586	467416	380287	337001	307415
column blocks number		12006	12009	11869	10479	10161	9840
Factorization time		52.3	51.1	49.7	45.1	42.3	41.3
Average block height		29.5	30.2	30.7	31	29.9	28.5
Minimal block height		1	1	1	1	3	3
Maximal block height		126	150	180	240	300	360
Average cblk width		78.6	78.6	79.5	90.1	92.9	95.9
Minimal cblk width		3	3	3	3	3	3
Maximal cblk width		126	150	180	240	300	360
Coeftab Size		9.5 GB	9.5 GB	9.5 GB	9.5 GB	9.6 GB	9.6 GB

Table 2.2 – Comparison of the different splitting methods, on `audikw_1` test case, with one node of 16 cores using PaStiX original scheduler. Minimal block size is set to 60, Maximal to 120. Amalgamation level is set to 20.

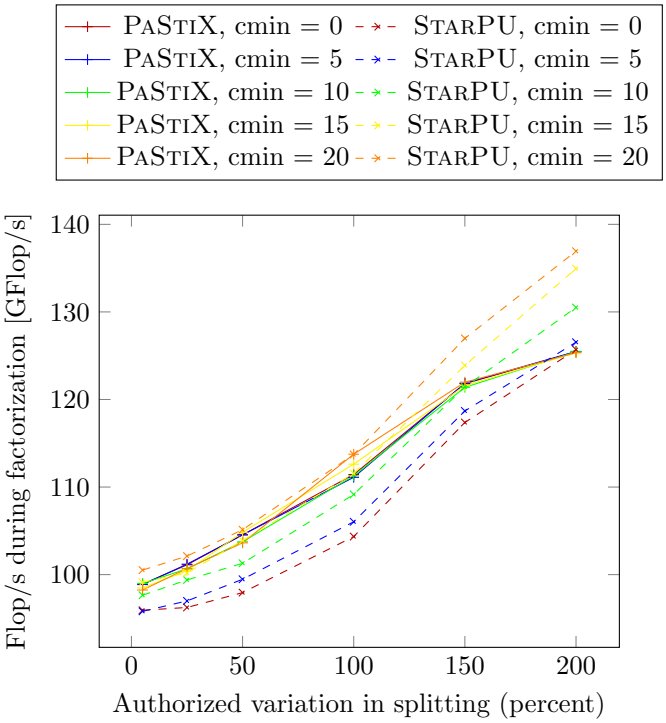


Figure 2.14 – Flop/s during factorization depending on authorized variation in column block splits on `audikw_1` matrix using a median first block splitting algorithm with a minimal block of 60 and a maximal value of 120.

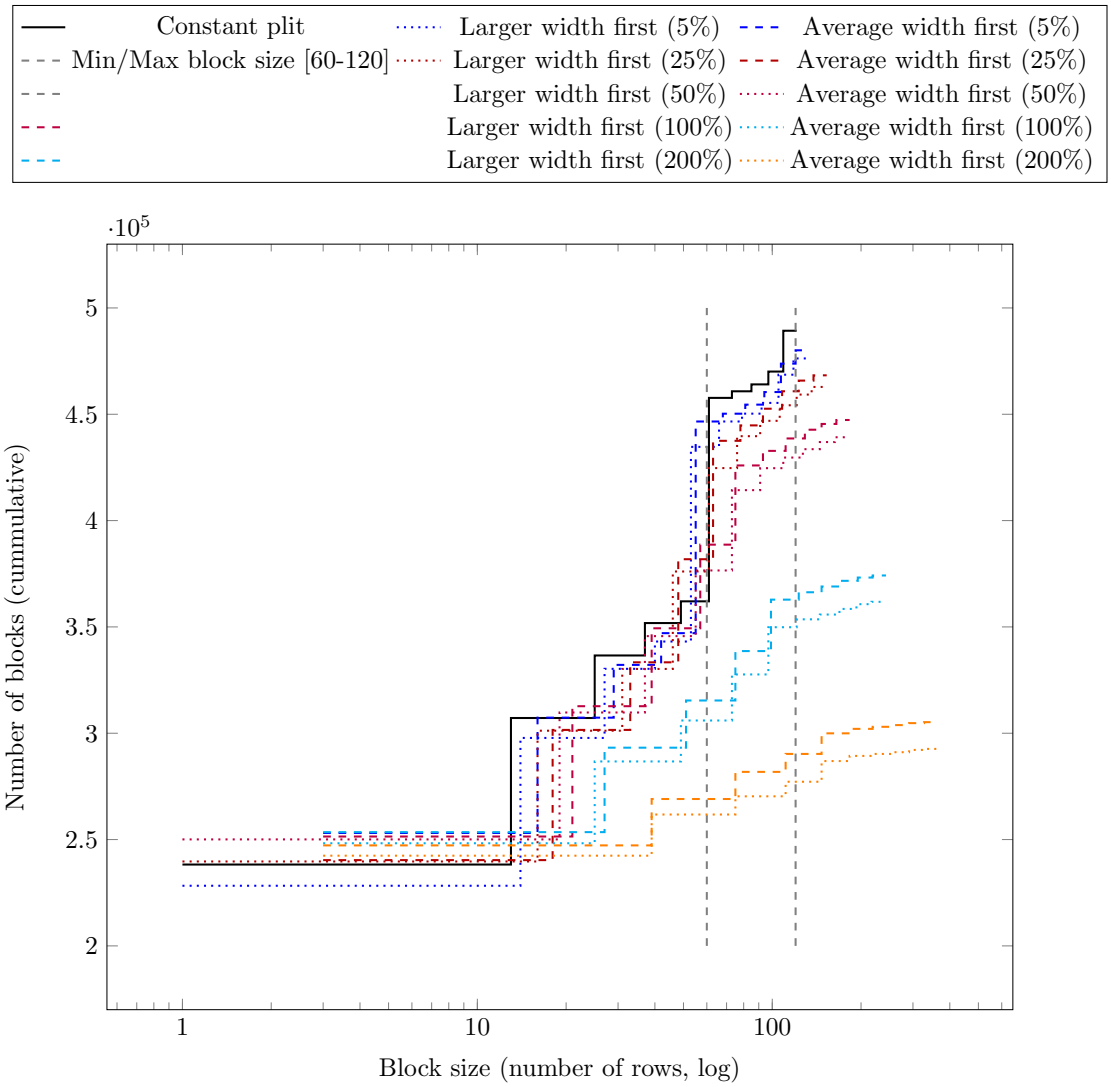


Figure 2.15 – Block's size study on `audikw_1` test case, with 8 cores. Plot of the number of blocks in the matrix beneath a given size.

Algorithm 6 Adapted splitting algorithm.

```
    ▷ Count the number of blocks facing each row
 1: nbBlocksPerLine ← computeNbBlocksPerLine()
 2: For each panel P Do
 3:     fcol ← firstColumn(P)
 4:     maxSize ← width(P)
 5:     nbCblk ← 0
    ▷ First compute an average splitting size
 6:     nseq ← computeNbSplit(candidateNbr(P), colNbr(P))
 7:     avgSize ← width(P)/nseq
    ▷ Split column block P while all the panel has not been split
 8:     While fcol < lastColumn(P) Do
 9:         lcol ← computeBestSplit(nbBlocksPerLine, avgSize, maxSize, ratio)
10:         addExtraCblk(fcol, lcol)
11:         maxSize ← maxSize − (lcol − fcol + 1)
12:         fcol = lcol + 1
13:         nbCblk ← nbCblk + 1
14:     End While
    ▷ Update number of blocks per line
15:     For each block b in P Do
16:         For each row r in b Do
17:             nbBlocksPerLine[r] ← nbBlocksPerLine[r] + nbCblk − 1
18:         End For
19:     End For
20: End For
```

of 200% in the column-block splitting algorithm. This will increase the performance of each task at the cost of a small memory overhead.

2.5 Discussion

In this chapter, we presented the framework used during this thesis. Our developments were integrated into PaStiX, a direct sparse linear solver library. To implement a heterogeneous aware algorithm, we used two different task-based runtime systems, StarPU and PaRSEC. We described how we adapted our algorithm to task-based runtime and compared performance with the original PaStiX implementation. Once we had demonstrated that the performances were comparable, we explained how we could efficiently address the GPUs using these task-based runtime implementations and presented a scaling study on heterogeneous nodes. Finally, we described the optimizations we had to perform to achieve the speed-ups. One of the required optimizations was to statically decide which updates will be performed on a GPU. We also had to rewrite the block splitting algorithm to minimize the number of tiny blocks in the block structure of the

Algorithm 7 Choice of the split: largest first.

```
1:  Function computeBestSplit_larger(nBlksPerLine, avgSize, maxSize, ratio)
2:      limit ← ceil(step * percent/100);
3:      If step ≥ max Then return max − 1
4:      End If
5:      lavg ← step − 1
6:      lmin ← MAX(lavg − limit, 1)
7:      lmax ← MIN(lavg + limit + 1, max)
8:      lcolnum ← lmin
9:      nbSplit ← nbBlocksPerLine[lcolnum]
10:     For i ∈ [[lmin + 1, lmax − 1]] Do
11:         If nbBlocksPerLine[i] ≤ nbSplit Then
12:             lcolnum ← i
13:             nbSplit ← nbBlocksPerLine[i]
14:         End If
15:     End For
16: End Function
```

factorized matrix. We still need to investigate which information could help the runtime system to schedule efficiently tasks on the GPU dynamically. Now that we can efficiently manage one heterogeneous node with those runtime systems, we can target clusters of heterogeneous nodes. The next chapter explains how we can address distributed heterogeneous nodes over generic task-based runtime systems.

Algorithm 8 Choice of the split: average first.

```
1:  Function COMPUTEBESTSPLIT_AVG(nBlksPerLine, avgSize, maxSize, ratio)
2:      limit ← ceil(step * percent/100);
3:      If step ≥ max Then return max − 1
4:      End If
5:      lavg ← step − 1
6:      lmin ← MAX(lavg − limit, 1)
7:      lmax ← MIN(lavg + limit + 1, max)
8:      lcolnum ← lavg
9:      nbSplit ← nbBlocksPerLine[lcolnum]
10:     For i ∈ [[lavg + 1, lmax − 1]] Do
11:         If nbBlocksPerLine[i] ≤ nbSplit Then
12:             lcolnum ← i
13:             nbSplit ← nbBlocksPerLine[i]
14:         End If
15:     End For
16:     For i ∈ [[lmin + 1, lavg − 1]] Do
17:         If nbBlocksPerLine[i] ≤ nbSplit Then
18:             lcolnum ← i
19:             nbSplit ← nbBlocksPerLine[i]
20:         End If
21:     End For
22: End Function
```

Chapter 3

Sparse factorization on distributed heterogeneous systems

Contents

3.1 Solving distributed sparse linear system 84

3.1.1 *Fan-out* implementation 84

3.1.2 *Fan-in* implementation 86

3.1.3 Data mapping 89

3.2 Experiments 92

3.2.1 Distributed implementation on homogeneous nodes 92

3.2.2 Distributed implementation on heterogeneous nodes 97

3.3 Discussion 99

We have seen in the last chapter that we can implement a sparse linear solver on top of a task-based runtime system with a rather good performance. Doing so, one can benefit from the separation of concern between the algorithm and the architecture to target heterogeneous shared memory environments. Now, we want to study the suitability of this approach to distributed memory architectures.

In this chapter, we present the distributed version of the matrix decomposition algorithm. The first section describes two versions of the distributed supernodal algorithm. The first-one is the *fan-out* method, really close to the sequential task algorithm and the second one, called *fan-in*, is the one implemented in the original version of PaStiX and that has been adapted to task based runtime systems. We implemented these two algorithms in PaStiX on top of StarPU. Then, we present the data distribution algorithm used in PaStiX (for both task-based runtime and original implementations of the

factorization algorithm) and the different versions of the original scheduler implemented in PaStiX prior to this thesis. After these descriptions, we compare the performances of the new StarPU distributed version of the factorization with the original scheduler implementations. These experiments are performed on two distributed clusters, the first one without accelerators, and the second one with GPU accelerators. We have only implemented the StarPU version of the distributed algorithm yet, but the corresponding algorithms using PaRSEC JDF approach would be similar.

3.1 Solving distributed sparse linear system

As in shared memory, we still consider column-blocks as the elementary data for distributed algorithm. The difficulty here is to compute an efficient distribution of those panels and the associated tasks, such that each node has an equivalent amount of work. To solve this problem, it is necessary to understand the way communications are performed in the distributed algorithm. Let's consider that each column-block is owned by an MPI process. This process will be in charge of computing the associated operations. The panel factorization (i.e. `POTRF` and `TRSM`) task is de facto associated with the panel, as in shared memory, and processed by the owner. The update operations (i.e. `GEMM`) are involving two column-blocks which can be on two different processors. The operation must be performed by the owner of one of them.

Figure 3.1 illustrates the distributed version of the algorithm. Here, we have four column blocks distributed on two processors: P_1 in red, and P_2 in blue. The column block owner logically executes the panel factorization on it. One can notice on Figure 3.1 that the update from C_0 to C_2 involves data from two processors and can be mapped on any of them.

Two algorithms exist to implement the distributed version of a supernodal method. The computations of the `GEMM` updates can be either shared by all the "senders" (*fan-in*), or processed only by the "receiver", here P_2 (*fan-out*). Both algorithms are detailed in the next sections, as well as their implementation on top of the StarPU runtime.

3.1.1 *Fan-out* implementation

The *fan-out* algorithm consists in performing the `GEMM` updates on the processor that owns the destination panel. It requires the panel that generated the update to be exchanged before that. Using task-based runtime systems, it is the straightforward method. Indeed, one would just have to give the data distribution in addition to the sequential tasks flow to implement this algorithm. In this case, `GEMM` updates are performed on the node that possesses the target panel. To be able to perform the update, the updating column block must be communicated before the `GEMM` is applied. This is automatically performed by the runtime system.

In order to be able to perform communications, StarPU requires to know each piece of data that will be used in computations and each task that will be executed, locally or not, on local data. One can then submit the sequential algorithm on each MPI process,

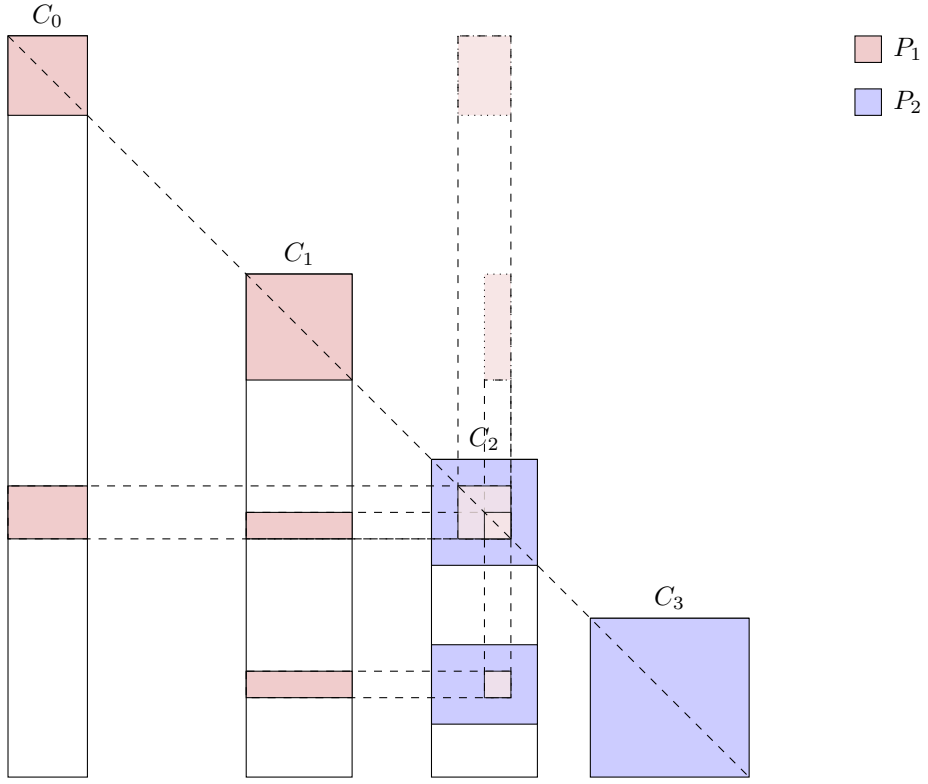


Figure 3.1 – Distributed example: four column blocks – C_0, C_1, C_2, and C_3 – distributed on two processors, P_1 and P_2.

and StarPU would discover the local tasks and the communications induced by the data mapping information. In our case, we do not want to submit the whole graph on all MPI process but only required tasks. Indeed, storing and inserting the whole graph would create an unnecessary overload for the memory and the computation.

In the Cholesky decomposition algorithm, the scheduler has to be aware of not only local data but also remote data that will be updated by local updates, and remote data from which updates will be applied on local column-blocks. We will call these data the halo. In our simple example, the halo corresponds to all the other processor's columns but, in a real case, it would represent a smaller part of the matrix.

In the *fan-out* version of the StarPU implementation, described by Algorithm 9, we have to declare both the local data and the halo data. Then for each local column block, incoming `GEMM` tasks, updating local data from distant column blocks, are submitted. Those updates will be executed on the local node after reception of the input data from the remote node. All communications from one incoming column block are submitted at once and an array (*seen*) is used to remember when incoming updates have been submitted to avoid doing it twice. After that, the incoming buffer can be flushed out of the MPI cache as they will not be used again. Then, the local panel is factorized

and updates from this panel are submitted. The targets of the updates are the column blocks which diagonal blocks are in the same rows than the off-diagonal blocks (b) of c. We call this column block the facing column block of the block b ($facing_cblk(b)$). Those `GEMM` tasks can update either local or remote column blocks. If the column blocks are remote, the runtime system sends the input data to the remote node and this node will perform the update. The c buffer can then be flushed out from the MPI cache.

To handle communication, StarPU detects remote data usage when a task is inserted. If the task will be executed locally, StarPU will allocate a buffer to receive the required data and post the MPI reception. In the case of a remote task, StarPU will send the local data involved whenever they are ready. A datum can be used several times by a remote task, and StarPU will use a cache mechanism to avoid sending the data multiple time. However, StarPU cannot discover if a datum will be reused or not. Indeed, StarPU cannot know whether the user will insert a new task using an already-received datum in the future. Thus, the user will have to indicate to the runtime when it can flush the cache corresponding to some data to free some memory. This is done using `starpu_mpi_cache_flush(MPI_Comm, data_handle)` routine. One can also use `starpu_mpi_cache_flush_all_data(MPI_comm)` to flush all the caches when the task insertion process is finished. The second one is easier to use for the user, but the information given to the runtime system is less accurate, and cache flushing can occur later.

This algorithm, applied on the four column-blocks illustration (Figure 3.1), leads to the *fan-out* tasks graph described in Figure 3.2(a). In the given example, processor P_1 sends column-blocks C_0 and C_1, once they are factorized, to processor P_2 before it can compute C_2 factorization. C_1 is also required to compute updates on C_3 before it can be factorized. The runtime system automatically detects this, and performs the communication when they are ready.

3.1.2 *Fan-in* implementation

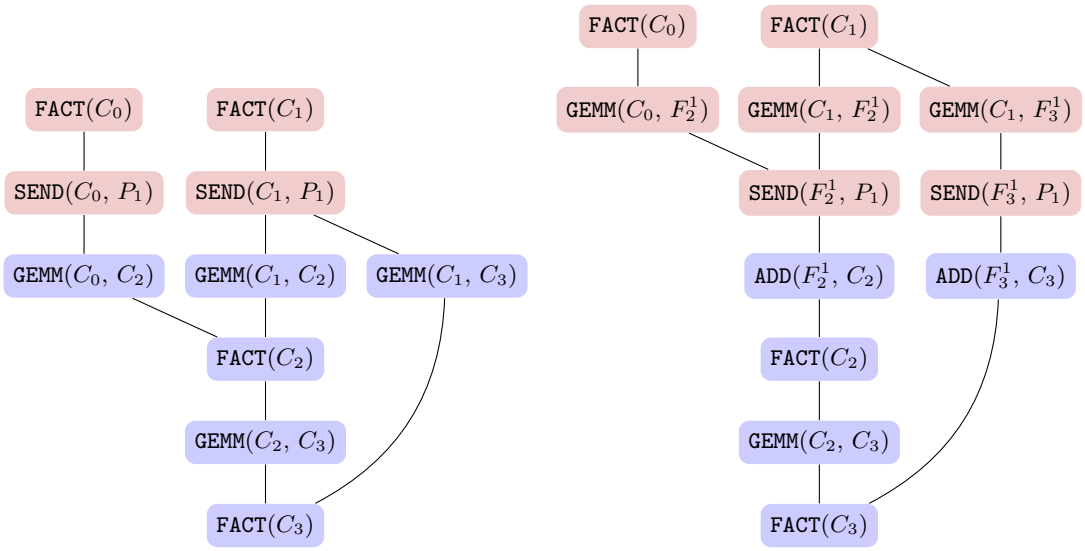
The *fan-in* implementation is the solution used by the original PaStiX scheduler. In the *fan-in* implementation, all updates from local column-blocks toward a same remote column-block are first computed locally. Then, this buffer is sent to the remote node for addition onto the targeted column-block. A new addition task has to be introduced into the task-graph to handle the addition of the *fan-in* buffers onto the corresponding panel. While in original PaStiX implementation the *fan-in* algorithm is implemented using independent blocks, we have decided to describe *fan-in* buffers as column-blocks in the runtime based implementation. This change may imply an overhead in the size of *fan-in* data as the width of the *fan-in* column block is the maximum width of the *fan-in* blocks that are used in original PaStiX implementation. The *fan-in* column-block pattern always fits inside the remote column-block, it is a sub-part of the destination panel.

Algorithm 10 describes the *fan-in* implementation. First the data are registered (lines 1 to 6). Then, for each local column block c, the incoming *fan-in* addition tasks are inserted to prepare the data reception. A new kernel is required to perform the

Algorithm 9 *Fan-out* Cholesky implementation with StarPU.

```
    ▷ Register all local column blocks
 1: For each local column block c Do
 2:     starpu_mpi_data_register(c, myrank)
 3: End For
    ▷ Register all local column block involved in local update or updated using local
    column blocks
 4: For each column block h in halo Do
 5:     starpu_mpi_data_register(h, owner(h))
 6: End For
    ▷ Submission of the factorization tasks
 7: For each local column block c Do
    ▷ Submit all local updates from remote column blocks on c
 8:     For each column block h in halo updating local column blok c Do
 9:         If not seen(h) Then
10:             For each local update from h on a local column block k Do
11:                 starpu_mpi_insert_task(gemm, h, k)
12:             End For
13:             starpu_mpi_data_flush(h)                    ▷ h will not be used again
14:             seen[h] ⇐ True
15:         End If
16:     End For
    ▷ Submit the panel factorization and updates
17:     starpu_mpi_insert_task(panel_factorization, c)
18:     For each block b ∈ c Do
19:         fc = facing_cblk(b)
20:         starpu_mpi_insert_task(gemm, c, fc)  ▷ This task is executed where fc is
    located.
21:     End For
22:     starpu_mpi_data_flush(c)                            ▷ c will not be used again
23: End For
24: starpu_task_wait_for_all()
```

addition on the local column-block after the *fan-in* column block has been received (line 9). Afterward, local column-blocks factorizations, and `GEMM` updates are inserted (lines 12 to 15). As for local column-block, the *fan-in* buffer will receive `GEMM` updates and thus will share the same "sparse `GEMM`" kernel. When all the contributions have been received the addition tasks can be submitted (line 17) to trigger the communication. In the *fan-in* case, we describe both *fan-in* buffer that will be received and that will be send. We do not need anymore to describe others processors column blocks because all communications will only use *fan-in* buffers. When we know that we will send a *fan-in* buffer we just have to use a *void* handle (named here $fake_cblk$) indicating the rank of



(a) *fan-out* algorithm's tasks graph. (b) *fan-in* algorithm's tasks graph.

Figure 3.2 – Task graph corresponding to the example presented in Figure 3.1. Red tasks are executed by processor P_1 and blue ones by P_2. F_i^p is the *fan-in* buffer associated to C_i on processor P.

the owner of the column block ($owner(fc)$).

Figure 3.3(b) represents the communications required to apply updates on a column-block from the previous example. Updates from C_0 and C_1 are gathered in the *fan-in* buffer F_2^1, which corresponds to all updates from P_1 on C_2. Afterward, F_2^1 is sent to P_2 which adds it to C_2. It corresponds to the original PaStiX communication scheme (Fig. 3.3(a)) except that, in PaStiX original implementation, *fan-in* buffers are split into blocks ($F_{2,0}^1$ and $F_{2,1}^1$) which are treated separately and correspond to the minimal contribution area. This can reduce a little bit more the memory consumption. Indeed, as it is shown in the example, a *fan-in* block in the block column can comprise fewer columns than the diagonal block. However, a quick study of the communication volume shows that the increase in the data exchanged is not that large. Table 3.1 present the comparative size of all *fan-in* buffers, with PaStiX original implementation and StarPU one. The overhead column presents the overhead of StarPU implementation, which uses column-blocks *fan-in* buffers, versus PaStiX ones, which use blocks. The Table present results for both 4 and 16 nodes from Avakas cluster (see 3.2.1, page 93). In practice, on all the presented test cases, the overhead is less than 1% of the original PaStiX implementation *fan-in* buffers size.

The *fan-in* tasks graph corresponding to the *fan-in* implementation in StarPU is presented in Figure 3.2(b). One can notice the new addition tasks added to the graph. `GEMM` updates are also moved from P_1 in red to P_2 in blue when going from *fan-out* algorithm to *fan-in* algorithm. On the given example, the *fan-out* version implies the

Algorithm 10 *Fan-in* Cholesky implementation with StarPU.

```
    ▷ Register all local column blocks
 1: For each local column blocks c Do
 2:     starpu_mpi_data_register(c, myrank)
 3: End For
    ▷ Register all fan-in column blocks
 4: For each fan-in column blocks f Do
 5:     starpu_mpi_data_register(f, owner(f))
 6: End For
    ▷ Submission of the factorization tasks
 7: For each local column block c Do
    ▷ Submit all local updates from remote fan-in column blocks on c.
 8:     For each received fan-in column blocks f updating c Do
 9:         starpu_mpi_insert_task(add, f, c)
10:         starpu_mpi_data_flush(f)                    ▷ c will not be used again
11:     End For
12:     starpu_mpi_insert_task(panel_factorization, c)
13:     For each block b ∈ c Do
14:         fc = facing_cblk(b)
15:         starpu_mpi_insert_task(gemm, c, fc)
16:         If fc is a fan-in column block and
                all local updates on fc have been submitted Then
17:             starpu_mpi_insert_task(add, fc, fake_cblk[owner(fc)])
18:             starpu_mpi_data_flush(fc)               ▷ fc will not be used again
                                                ▷ Will be performed on remote node.
19:         End If
20:     End For
21: End For
22: starpu_task_wait_for_all()
```

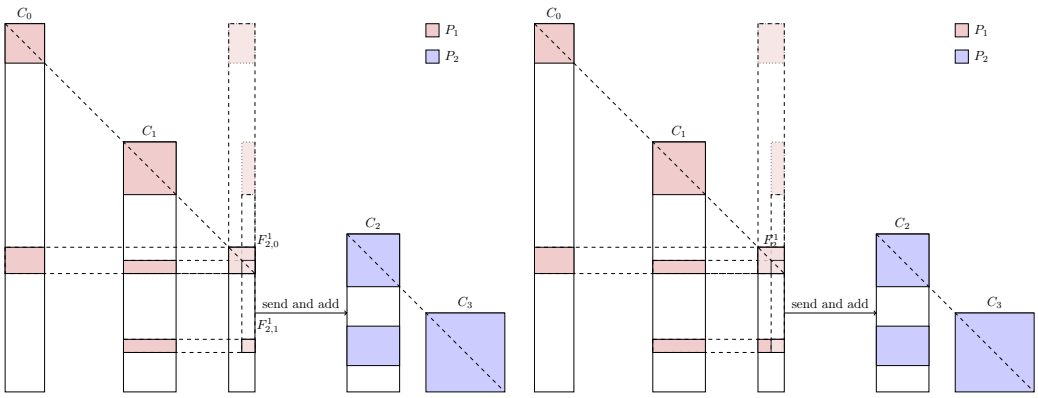
exchange of C_0 and C_1 while the *fan-in* version exchange only smaller *fan-in* buffers F_2^1 and F_3^1 that represent sub-part of C_2 and C_3. Here, as in most real cases, the *fan-in* version is more efficient as it reduces the communication volume.

3.1.3 Data mapping

StarPU requires the user to provide a data distribution to be able to decide when communications are required. In our case, the data distribution used with the StarPU implementation of the factorization algorithm is the one developed for PaStiX original scheduler. It is performed once, during the graph analysis. As PaStiX original scheduler uses a *fan-in* algorithm, the cost models used in the analysis step to compute the data distribution and the static task scheduling takes this property into account. The distribution of the column-blocks follows a proportional mapping of the elimination tree

Matrix	4 MPI nodes			16 MPI nodes		
	fan-in buffer size (Mo)		overhead	*fan-in* buffer size (Mo)		overhead
	PaStiX	StarPU		PaStiX	StarPU	
af_shell10	38.09	37.99	0.25%	194.8	194.6	0.10%
dielFilterV2clx	33.32	33.32	0%	148.5	148.2	0.25%
Flan_1565	128.1	128.1	0%	666.7	666.5	0.03%
audikw_1	284.4	284.4	0%	1287	1287	0%
matr5	331.4	331.4	0%	1988	1988	0%
Geo_1438	1017	1017	0%	4484	4484	0%
PmlDF	485.7	485.7	0%	2313	2313	0%
Hook_1498	485.1	485.1	0%	2406	2406	0%
Serena	1375	1375	0%	6384	6384	0%

Table 3.1 – Comparison of the *fan-in* memory overhead. Total memory used by *fan-in* buffer on whole cluster.



(a) with PaStiX (b) with StarPU

Figure 3.3 – *Fan-In* distributed example : four column blocks – C_0, C_1, C_2, and C_3 – distributed on two processors, P_1 and P_2.

computed at the graph processing step. First, each node of the tree is associated with the cost of the corresponding panel factorization. This cost includes the panel factorization, the matrix products for the different updates from the panel, and the addition of these contributions on either the target column-block or *fan-in* buffer. The main bias of this model is that, the addition is always associated with the source column-block. This association is correct in shared memory, but in a distributed context the additions of updates on a remote panel are computed by the target supernode's owner. This cannot be taken into account during the cost evaluation because we do not know yet if a column-block will be local or remote. We consider this bias negligible for the proportional mapping step which attributes only a set of candidates to each sub-tree. The simulation of the factorization that actually assigns a process to each panel corrects this default. In *fan-out*, the products should also be attached to the receiving column-blocks but, for the

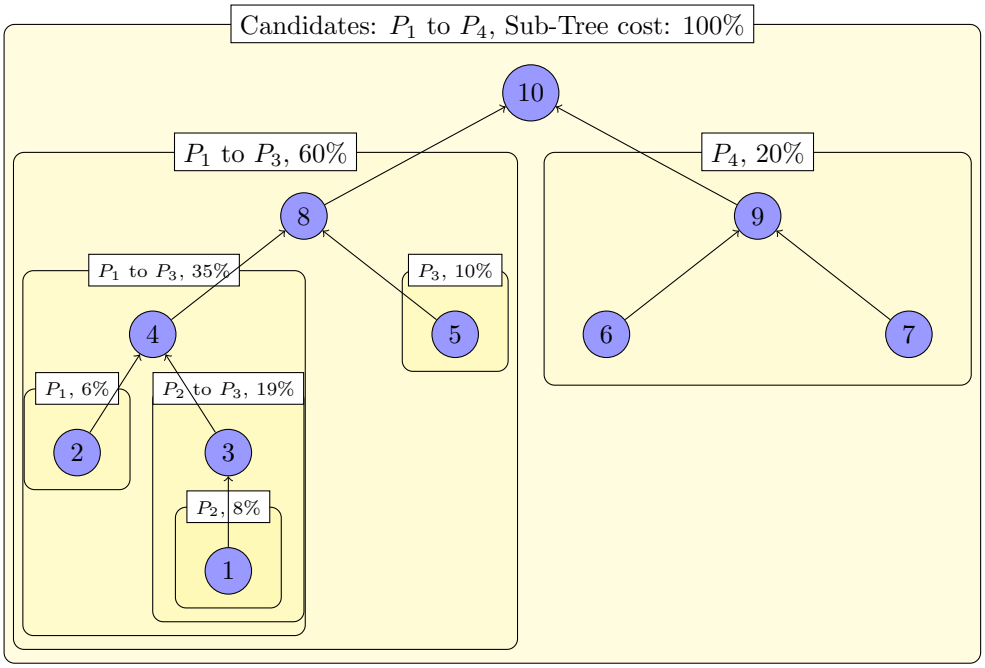


Figure 3.4 – Elimination tree with proportional mapping information.

same reason, it is attached to the source supernode. Thus, the cost estimation of each node corresponds to a shared memory execution and can lead to load unbalances.

Then, the processors are mapped onto each node depending on the cost of the associated sub-tree. The root is associated with all the processors. For each level of the elimination tree, the independent sub-trees are recursively associated with subsets of the current branch candidates following the relative computational cost of each sub-tree. Figure 3.4 presents the elimination tree with the proportional mapping information.

Finally, the static scheduling step associates each of the column-blocks with one of its candidates following the load of the candidates. This algorithm simulates the factorization using a computational and communication costs model. Each processor P_i is associated with a time t_i, corresponding to the amount of work it has been assigned to, and with a list of ready tasks it is associated with. The couple of processor and task that can be executed first is scheduled. If several couples can be scheduled, the task associated with the deepest node in the elimination tree is scheduled first because it is more likely to be critical to the release of other tasks. Figure 3.5 illustrates the simulation corresponding to Figure 3.4. Here, the task T_3 is chosen among the ready tasks and associated with the process P_2 because it can be executed first. This will unlock the task T_4 that will be added to the ready tasks lists of candidate processors P_1 and P_2.

In PaStiX, before the coupling with task-based runtime systems, multiple modes have been developed to target clusters of shared memory nodes. The first one is the

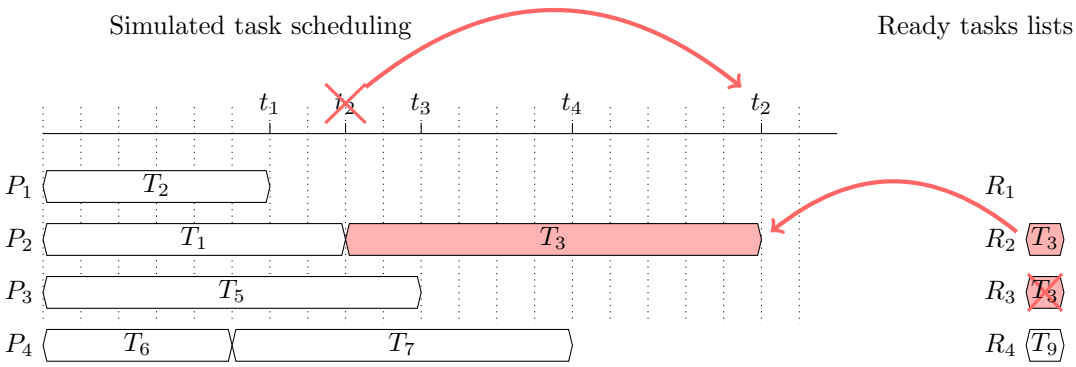


Figure 3.5 – Static scheduling computation via a simulated factorization.

thread multiple implementation were all threads are communicating over the network without restriction. This version requires the MPI library to provide an MPI thread multiple implementation of the API which is, unfortunately, not always the case even as of today. To be able to run PaStiX with nearly all MPI implementations, a *funneled* version of the algorithm has been developed. In this version, all communications are funneled through one communication thread. Besides handling the MPI communication, this thread is also in charge of the addition of the received *fan-in* buffers. The last version, developed by Mathieu Faverge during his PhD thesis [Fav09] to improve the scaling on clusters of NUMA nodes, is the dynamic scheduling implementation. As for the *funneled* version, it uses a dedicated thread to perform communications, but also allows for communication reordering to answer the requirement of dynamic scheduling. This implementation also authorizes job stealing between threads to correct the possible errors in the precomputed static scheduling.

3.2 Experiments

In previous section, we presented the two different implementations of the task-based distributed decomposition algorithm we implemented in PaStiX on top of StarPU. We also explained how data are mapped onto MPI processes and quickly described the different distributed versions of the original scheduler available in PaStiX. This section presents a performance comparison of the two task-based runtime system algorithm implementations with the different versions of the static and dynamic original schedulers. The first part of the study has been performed on a CPU only cluster. In the second part, this study extends to distributed heterogeneous experiments.

3.2.1 Distributed implementation on homogeneous nodes

Experiments were performed on the Avakas cluster from the University of Bordeaux. This cluster is composed of 264 nodes with 48 GB memory and two hexa-core Intel®

Xeon® x5675 @ 3,06 GHz each. All experiments use the full nodes with 12 threads and MPI between nodes.

Figure 3.6 presents different implementations of the original scheduler, all of them using *fan-in* algorithm:

- the funneled version, with crosshatch dots patterns;
- the thread multiple version, with white filling;
- the dynamic scheduling option, with diagonal lines.

One can observe that while the time scaling of the funneled version is the worst one when the size of the problem is rather small (e.g. `af_shell10`, `dieFielterV2clx`, `Flan_1565`, `audi_kw1`), it outperforms the thread multiple one when the number of operations grows (e.g. `PmlDF`, `Serena`). The dynamic scheduler option provides the best performances on all cases. Indeed, the static scheduler gets less efficient when the number of cores increases, and the dynamic scheduler algorithm tends to absorb this unbalancing. The differences between the *thread multiple* and the *thread funneled* versions mostly come from the additions of *fan-in* buffers that are moved to the communication thread.

Figure 3.7 compares the original thread multiple implementation (white bars), with the two StarPU implementations: *fan-in* (light colored) and *fan-out* (dotted and colored). We chose to compare our task-based implementation with the *thread multiple* on first because the algorithm remains the same in the two cases, the additions are performed by the computing threads (which are also communicating). On the contrary, in the two others versions of PaStiX original scheduler, the additions are moved to an additional communication thread. This scaling study shows that, surprisingly, the *fan-out* case gets better performance on the matrices involving the largest number of operations. With this method, updates are performed as soon as possible, and no extra-temporary buffers are required. The separation of the updates may allow more communication overlapping which could explain the performance growth in some cases. Using the generic task-based scheduler, one can achieve comparable performance to the static scheduler. It can even be faster on some cases, such as `audi_kw2`, `matr5`, or `pmlDF`, when the *fan-out* implementation is more efficient.

Using the dynamic scheduling option, on figure 3.8, we can obtain a better scaling and outperform the task-based runtime implementations in any case. On can see that largest problems exhibit the limited scaling of the task-based implementation when the number of tasks increases. We will have to interact with the runtime developers team to identify and overcome the sources of this limited scaling.

The 10 Millions unknowns test case is a larger matrix with 10 423 737 unknowns and 89 072 871 non zeros in A. The matrix is in complex double precision, and its factorization is done using the LDL^T algorithm. The L matrix contains 6 739 610 079 non-zeros, and the factorization involves 1.72e+14 operations. The factorization of this matrix on four nodes of twelve cores involves 10 681 100 tasks, from which 331 866 are column block factorization and the rest are sparse `GEMM` updates, which can create a large overhead for the scheduler. The results with this test case are presented in Figure

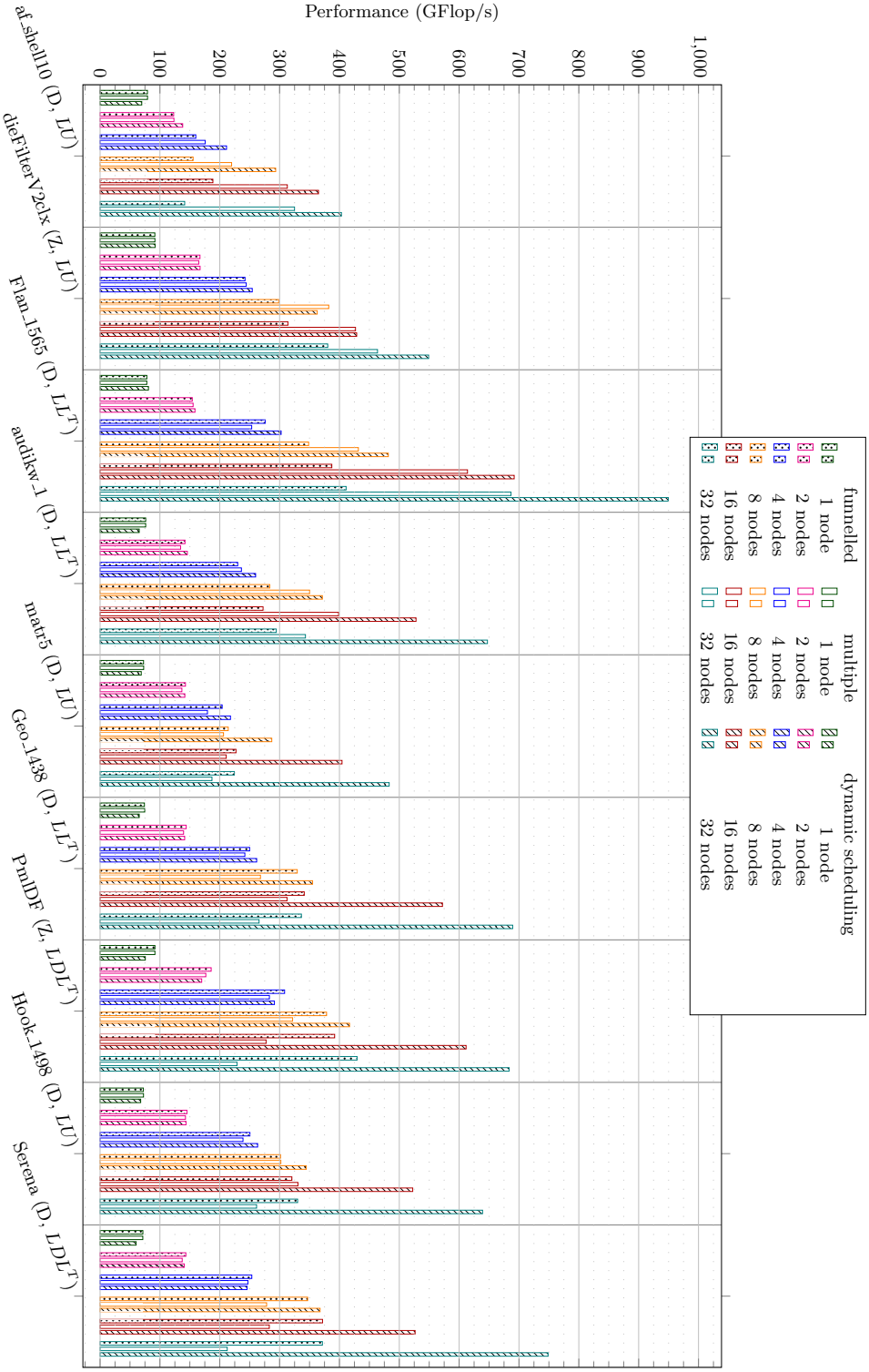


Figure 3.6 – Comparison of distributed implementations of PaSTiX original scheduler. All these experiments have been executed on 12-cores nodes from Avakas cluster from the University of Bordeaux, using 12 threads per node.

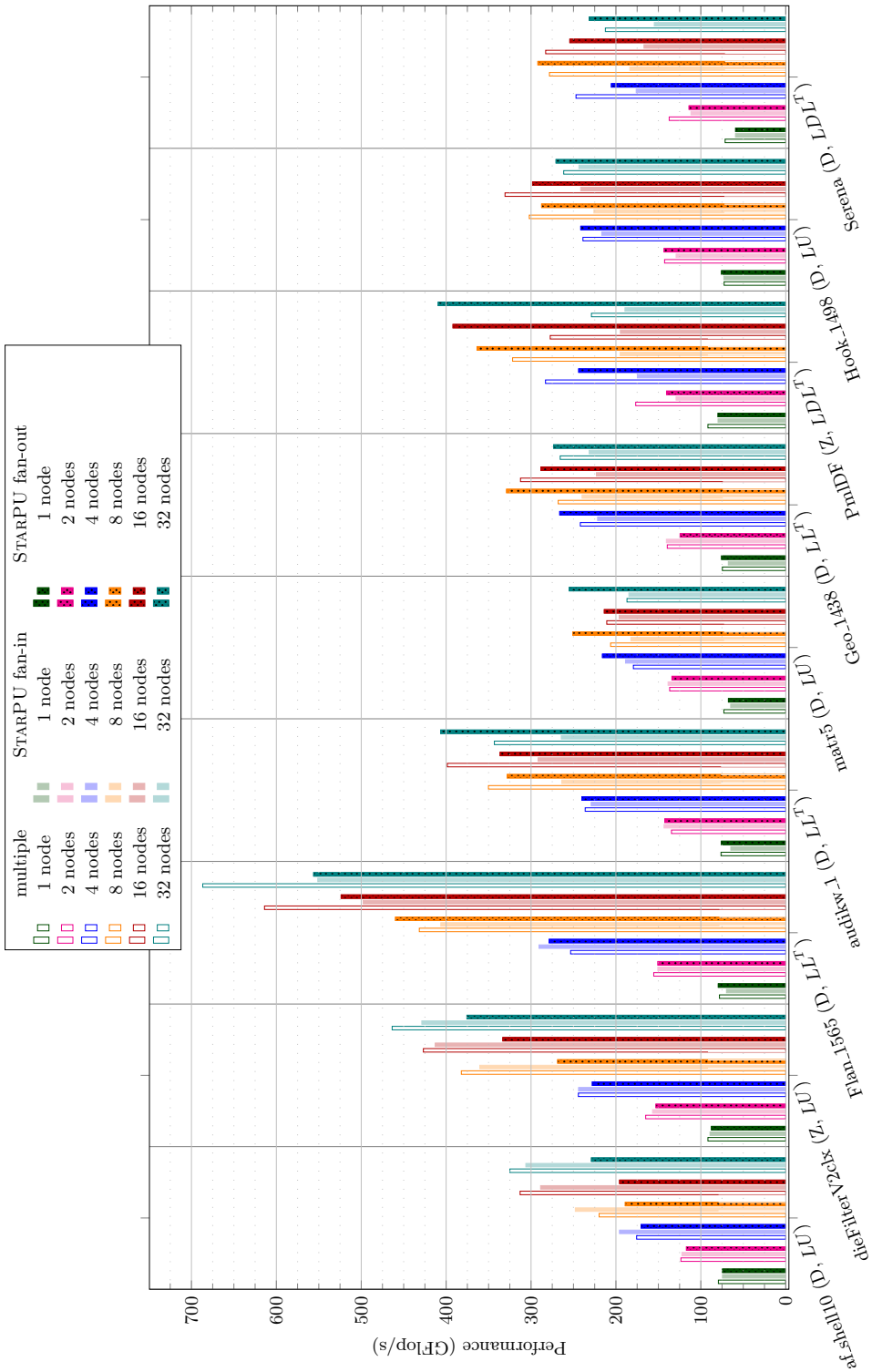


Figure 3.7 – Comparison of the distributed implementations using thread multiple version of the original scheduler and StarPU in *fan-in* and *fan-out* modes. All these experiments have been executed on 12-cores nodes from the Avakas cluster at the University of Bordeaux, using 12 threads per node.

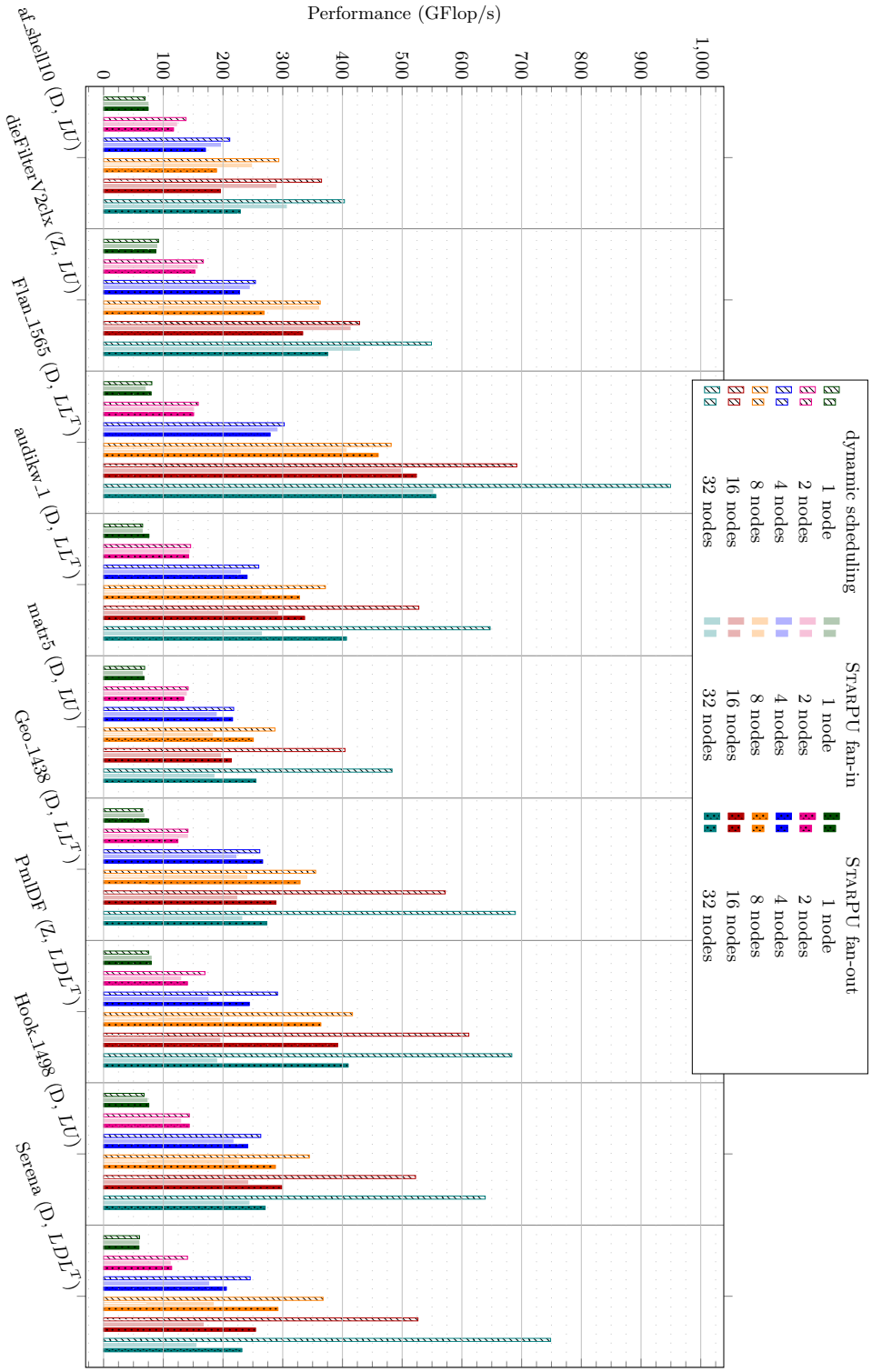


Figure 3.8 – Comparison of distributed implementations dynamic scheduling version of the original scheduler and StarPU in *fan-in* and *fan-out* modes. All these experiments have been executed on 12-cores nodes from Avakas cluster from the University of Bordeaux, using 12 threads per node.

3.9 and shows a better scalability of the PaStiX original implementations over the task-based runtime ones. This Figure does not present results with less than 4 nodes because the matrix would not fit into the memory, the matrix requires 31.2 GB per node when distributed on 4 nodes. The task-based runtime system implementations are managing a too large problem, with too many small tasks to obtain good performances. The *fan-out* implementation did not even run on less than 16 nodes because of a lack of memory. Then, it scales better than the *fan-in* one. Indeed, in StarPU, the communications buffers are allocated as soon as the receiving tasks are inserted which creates a large memory overhead. If these buffers fit in memory, the *fan-out* version can be more efficient because communications are performed earlier and additions can be applied whenever the thread is idle. Several solutions are currently studied to reduce the memory overhead:

- One could use a submission window that would limit the number of submitted tasks. After a certain number of tasks are given to the runtime the inserting function would become synchronous and wait until tasks are consumed. Doing so, the number of inserted communicating tasks would be limited and the size of the communication buffers would be reduced. In that case, we would have to move our receiving task insertion before the insertion of each panel factorization. The submission window is not yet implemented in StarPU but this feature is used in some runtime systems as Quark. In our case, we could also submit only ready tasks, this would limit the number of submitted tasks but would give less information to the runtime.

- The task-based runtime system could use a rendezvous communication scheme to allocate the data only once the sender has the data ready to send. The data would be available slightly later, decreasing the performances, but the amount of required memory would be reduced. This option can only be implemented inside the task-based runtime system and is not developed yet inside StarPU.

These results are encouraging but we still have to exchange with the runtime system developers team to improve the task-based implementation. We also need to investigate the possibility to dynamically choose between *fan-in* or *fan-out* algorithm, for each column block, to obtain the best performances from these two approaches.

3.2.2 Distributed implementation on heterogeneous nodes

Once the distributed version has been implemented using task-based runtime systems, we can benefit from cluster of heterogeneous node. Experiments mixing MPI and heterogeneous feature obtained with StarPU in PaStiX were conducted on the Curie cluster from the TGCC[8]. The hybrid partition of this cluster contains 144 nodes, each one composed of 2 quad-cores Westmere-EP at 2.67 GHz associated with 2 NVIDIA M2090 GPUs. Each node is equipped with 24 GB of memory and each GPU with 6 GB.

[8]TGCC: Très grand centre de calul `http://www-hpc.cea.fr/fr/complexe/tgcc.htm`

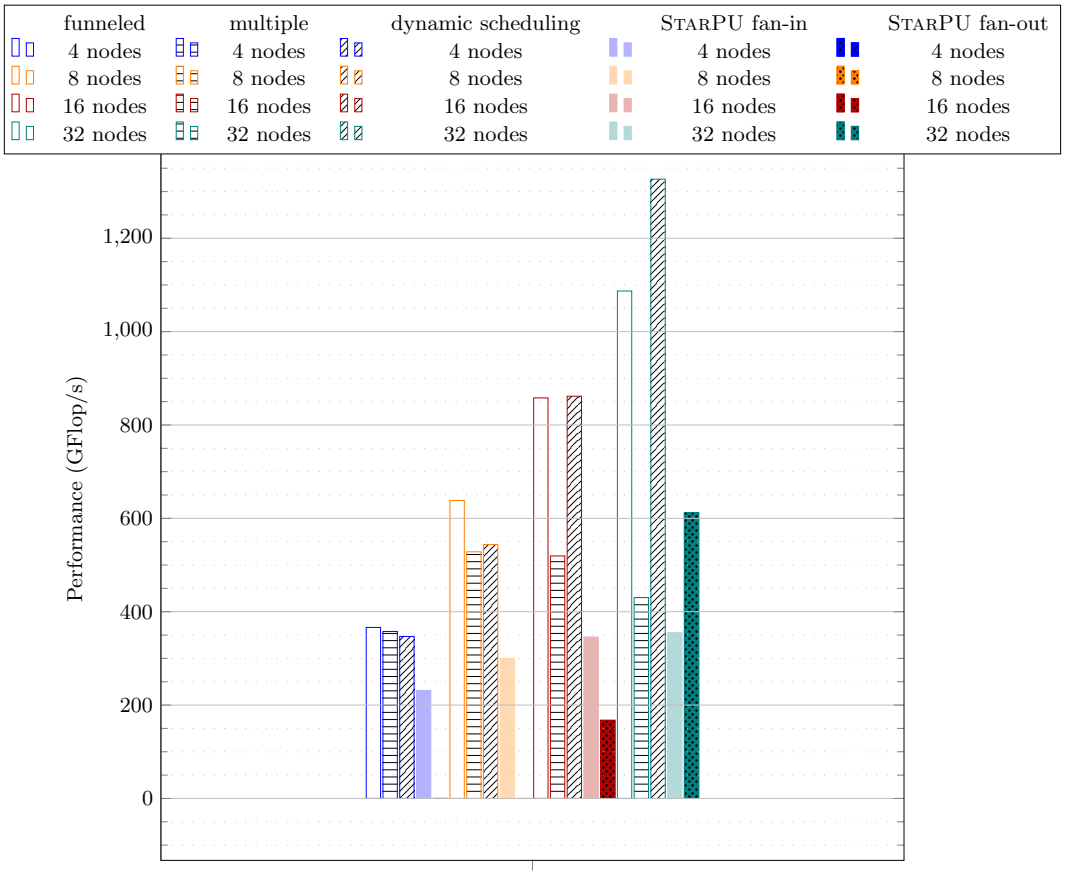


Figure 3.9 – Comparison of distributed implementations on the 10 Millions test case.

Figure 3.10 and 3.11 present results with respectively 4 nodes and 16 nodes from the Curie cluster. In blue we have results with 8 CPU cores only for comparison. Red bars are representing results with one GPU, while green ones are experiments with 2 accelerators. The comparison is done with the funneled implementation of PaStiX original scheduler as multiple communication threads are not supported by the provided MPI library. On four nodes, on `Geo_1438`, the *fan-in* algorithm and two GPUs the factorization can be accelerated by 1.46 against the original PaStiX scheduler (without GPU). The *fan-out* with two GPUs accelerates the factorization of `PmlDF` matrix by 1.39 against the original version (without GPU). On some cases, particularly with 16 nodes, using a GPU is worthless (e.g. on `audikw_1`). This is due to our distribution algorithm which tries to fill the GPU as much as it can whereas there are not enough tasks for every worker. We should adapt our algorithm by taking into account the number of workers and the number of tasks to schedule. This reveals that the scheduler does not consume immediately the received data that fill the memory of the node. We tried setting very high priority to tasks involving communications but without success. This can also explain the good performance obtained with the *fan-out* implementation that should be less efficient than the *fan-in* one. A tradeoff between memory and performance could be found by consuming received buffers when the memory is getting low. One can conclude from these experiments that, with the current implementation of the solver and the runtime system, the replacement of a classical core with a GPU can accelerate the computation but we still have some work to do with the runtime team to obtain a significant acceleration. It would be also interesting to study the distribution of the large blocks and adapt the distribution algorithm the to obtain blocks large enough to obtain good performances on the GPU.

3.3 Discussion

In this chapter, we have detailed how our task-based heterogeneous shared memory implementation of the matrix decomposition algorithms can be adapted to distributed nodes. We have presented two different implementations, *fan-in* and *fan-out*. We compared the results with different versions of the original scheduler, and the results are comparable to the performance of the original static scheduler but we still have to catch up with the state of the art dynamic scheduler. The advantage of this new solution compared to the dynamic scheduler is that all the complexity of the dynamic handling has been deported to the runtime and separated from the algorithm, making it easier to modify. The other advantage is the possibility to use accelerators which is not supported by the original dynamic scheduler. This feature is only relevant with a small number of nodes, when all nodes are fed with large blocks and can benefit from the accelerators. The distribution algorithm will have to be adapted to generate a distribution more adapted to heterogeneous systems.

Another feature brought by task-based runtime systems that could be worth trying in a direct sparse linear solver context is out-of-core computing. An attempt of out-of-core implementation of PaStiX had been done manually earlier but it was quite

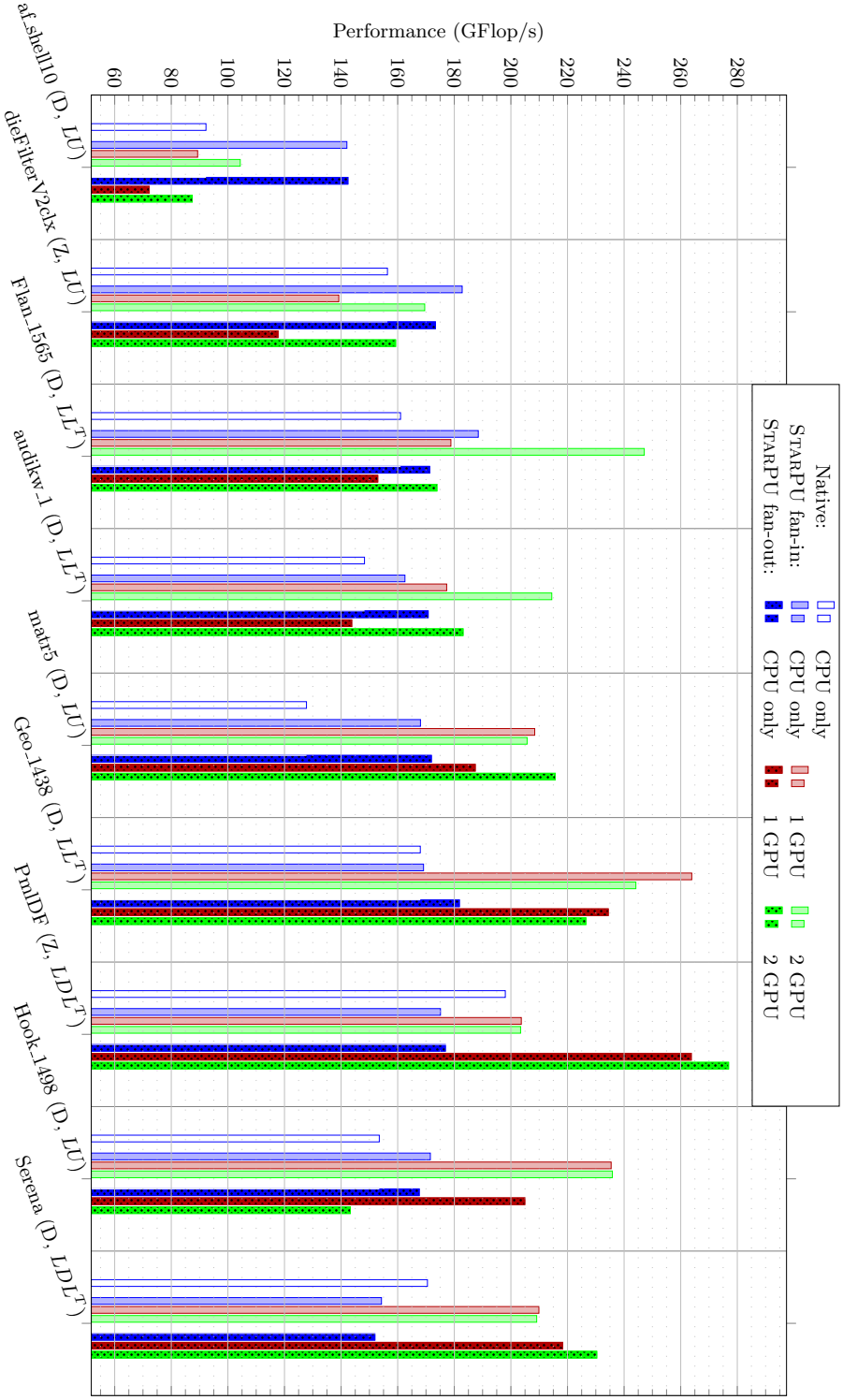


Figure 3.10 – Distributed heterogeneous scaling study on 4 nodes comparing the thread multiple original implementation and the two task-based ones. All these experiments have been executed on 8-cores nodes from Curie cluster from the French TGCC, using 8 threads per node. Each thread controlling either a CPU or a GPU

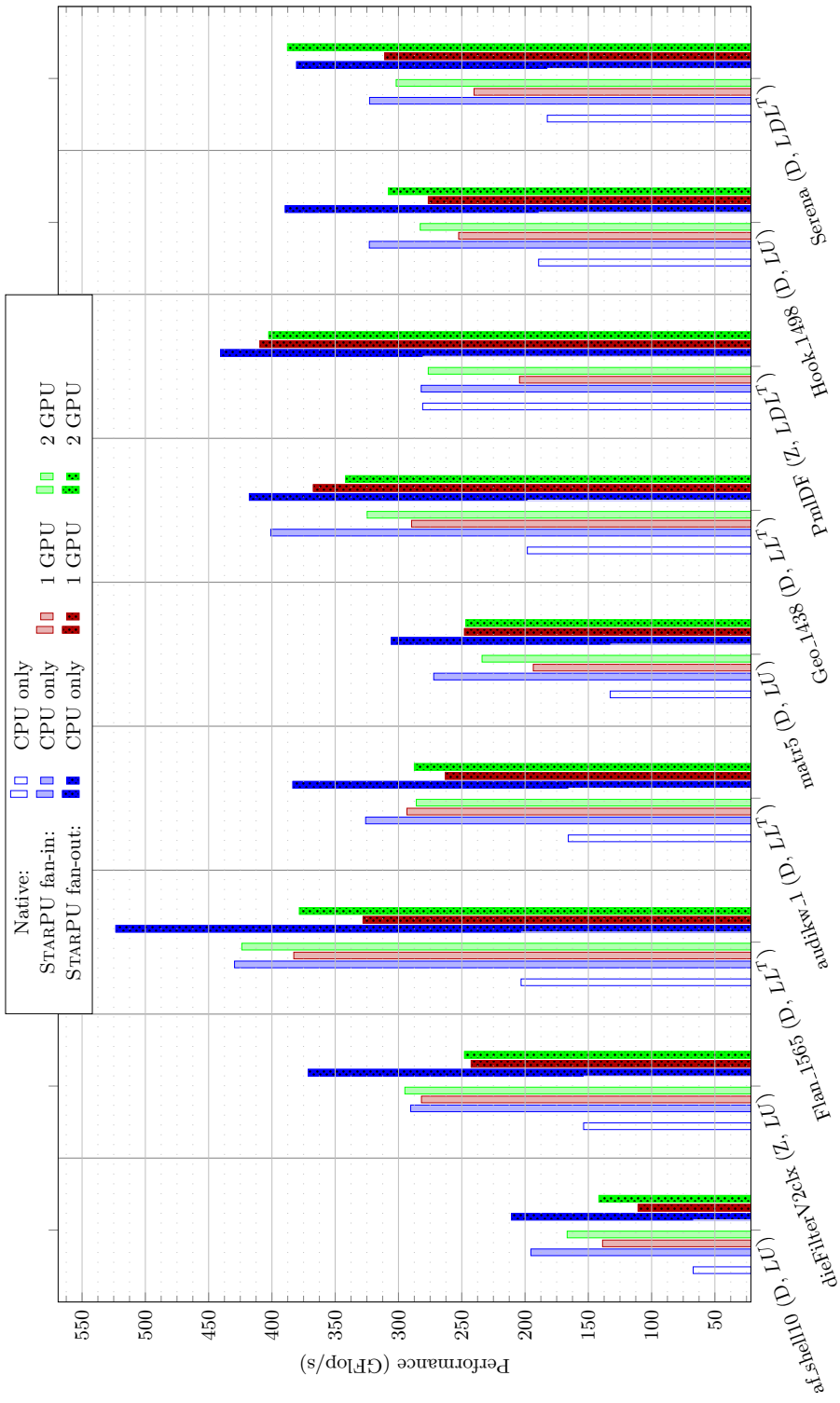


Figure 3.11 – Distributed heterogeneous scaling study on 16 nodes comparing the thread multiple original implementation and the two task-based ones. All these experiments have been executed on 8-cores nodes from Curie cluster from the French TGCC, using 8 threads per node. Each thread controlling either a CPU or a GPU

complex and has not been maintained. The usage of task-based runtime systems brings that feature with nearly no cost and a study of this possibility would be interesting and would help saving memory during the factorization. Other key features (e.g. energy efficient computation) could be brought "freely" through the runtime system and one can see here the benefits of separating the algorithm from the runtime system using task-based runtime systems.

Chapter 4

Integration in a controlled plasma fusion simulation code: JOREK

Contents

4.1 Description of the framework 104
4.1.1 Set of equations 105
4.1.2 Spatial discretization 105
4.1.3 Time integration scheme 106
4.1.4 Equilibrium 106
4.1.5 Sparse solver and preconditioning 107
4.2 Assembly step in JOREK 108
4.3 Optimized distributed matrix assembly 112
4.3.1 Generic distributed finite element assembly oriented API . . . 113
4.3.2 Comparison with PETSc 115
4.4 Integration into JOREK 118
4.4.1 Implementation 118
4.4.2 Timing and memory scaling study 119
4.5 Discussion 121

In the context of the ANR[9] ANEMOS project, which interests into controlled plasma fusion inside simulation, we worked on the integration of PaStiX direct solver in JOREK simulation code [HC07; CH08]. JOREK is a magnetohydrodynamic (MHD) simulation code that modelizes burning plasma inside tokamaks[10] (see Figure 4.1). JOREK simulation requires an efficient sparse solver to solve a linear system at each time step, and a large amount of memory to store simulation data. Thus, it is mandatory

[9] Agence National pour la Recherche, a agency french that provides funding for project-based research
[10] A tokamak is a device using magnetic fields to confine plasma in the shape of a torus

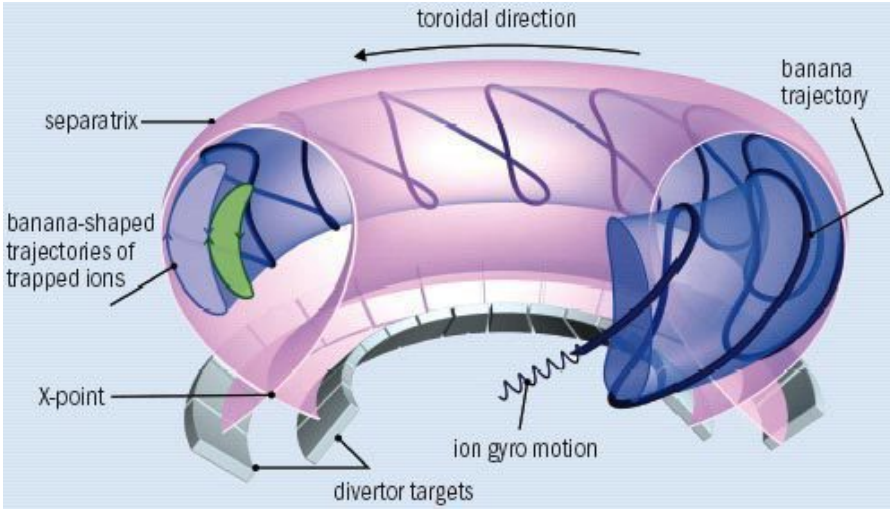


Figure 4.1 – A tokamak.

to provide a solver with the lower memory footprint as possible. The way the system is provided to the solver might also be of large impact. Generally, the input problem is gathered before calling the sparse linear solver library. This chapter presents JOREK and the integration of PaStiX as a part of the linear system solving process used to solve the problem resulting from the simulation. Then, we study the effect of providing an interface adapted to finite element simulations in order to exploit the distributed interface of the solver. This interface is first studied independently, then the effect inside JOREK simulation code is analyzed. The interface needs to provide functionalities to assemble the matrix in a distributed way. We explain its integration inside the JOREK numerical simulation code. One can notice that while this work takes place in the context of PaStiX and JOREK, this interface API is generic, and the study on the matrix assembly is applicable to many numerical simulations that require assembling a distributed matrix.

4.1 Description of the framework

MHD stability and the avoidance of plasma disruption – rapid loss of plasma energy and abrupt termination of the plasma current caused by global MHD instability growth – are key considerations to the attainment of burning plasma conditions in ITER (a large fusion reactor [11]). Realization of adequate MHD stability and careful control of the plasma operation will be critical ITER operation issues: MHD instabilities can damage components of the tokamak walls.

Numerical simulations play an important role in the investigation of the non-linear behavior of these instabilities and can help for interpreting experimental observations

[11] `http://www.iter.org`

and make predictions. In the framework of non-linear MHD codes, targeting realistic simulation requires:

- to model a complex geometry,
- to handle a large gap between the different time scales relevant to plasmas,
- to address full 3D simulation.

Computational time needed to run 3D MHD code named JOREK requires parallel computing in order to get reduced restitution time for the user [CH08; HC07].

In this section we present the equations involved in JOREK simulation, the discretization that is used to represent the tokamak, and the integration scheme. Then, we describe the linear solver that has been implemented to solve them.

4.1.1 Set of equations

JOREK is an MHD three dimensional fluid code that takes into account realistic tokamak geometry. Some of the variables modelled in JOREK code are: the poloidal flux Ψ, the electric potential u, the toroidal current density j, the toroidal vorticity ω, the density ρ, the temperature T, and the velocity along magnetic field lines $v_{parallel}$. Depending on the chosen model, the number of unknowns and the equations allowing the problem closure are setup. At every time-step, this set of reduced MHD equations is solved in weak form as a large sparse implicit system. The fully implicit method leads to very large sparse matrices. There are some benefits to this approach: there is no *a priori* limit on the time step, the numerical core adapts easily on the physics modelled (compared to semi-implicit methods that rely on additional hypothesis). There are also some disadvantages: high computational costs and high memory consumption for the parallel direct sparse solver (PaStiX or others).

4.1.2 Spatial discretization

High spatial resolution in the poloidal plane is needed to resolve the MHD instabilities at high Reynolds and/or Lundquist numbers. Bezier patches (2D cubic Bezier finite elements, see Figure 4.2) are used to discretize variables in this plane. Hence, several physical variables and their derivatives have a continuous representation over a poloidal cross section. The periodic toroidal direction is treated via a Fourier sine/cosine expansion.

The weak formulation of the MHD problem writes:

$$\frac{\partial C(\vec{u})}{\partial t} = D(\vec{u}), \text{where} u(t = \vec{t_0}) = u_0 \tag{4.1}$$

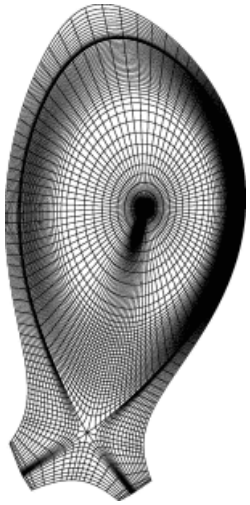


Figure 4.2 – Bezier 2D representation of a tokamak's plan.

4.1.3 Time integration scheme

We note u_n the set of variables. Applying a fully implicit second-order linearized Crank-Nicholson scheme for temporal discretization leads to solve the following system of equations:

$$\left(\frac{\partial C(\vec{u_n})}{\partial u} - \frac{1}{2}\delta_t \frac{\partial D(\vec{u_n})}{\partial u}\right) \delta\vec{u} = D(\vec{u_n})\, \delta t \tag{4.2}$$

Where $\delta\vec{u} = \vec{u_{n+1}} - \vec{u_n}$ is the vector of unknowns. In the following, we will refer to A as the sparse matrix that should be solved, and b the right hand side of the problem.

$$\text{Let us denote } \; A = \frac{\partial C(\vec{u_n})}{\partial u} - \frac{1}{2}\delta_t \frac{\partial D(\vec{u_n})}{\partial u}, \quad b = D(\vec{u_n})\, \delta t, \quad \text{then} \tag{4.3}$$

$$A\, \delta\vec{u} = b \tag{4.4}$$

4.1.4 Equilibrium

Each JOREK simulation begins with the solving of the static magnetic equilibrium equation (so-called Grad-Shafranov equation) in the two dimensions of the cross section plane. A key-point is the ability to handle magnetic equilibria which include an X-point. High accuracy is needed to have a correct representation of this equilibrium and avoid spurious instabilities whenever the whole 3D time stepping simulation is launched.

JOREK is able to build a Bezier finite elements grid aligned with the flux surfaces both inside and outside the separatrix[12] (*i.e.* the flux surface containing the X-point).

[12] Also named divertor, a separatrix is a mechanism for magnetically limiting a plasma, and hence for controlling the nuclear fusion in a tokamak

This strategy allows one to improve the accuracy of the equilibrium representation. The flux surfaces are represented by sets of 1D Bezier curves determined from the numerical solution of the equilibrium. The Grad-Shafranov solver is based on a Picard's iteration scheme.

After the Grad-Shafranov solving step, a supplementary phase is required: the time-evolution equations are solved only for the $n=0$ mode (the first toroidal harmonic, *i.e.* purely axisymmetric harmonic) over a short duration. First, very small time-steps are taken, then they are gradually increased. This process allows the plasma equilibrium flows to establish safely in simulations involving a X-point.

4.1.5 Sparse solver and preconditioning

JOREK simulation requires solving large problem and also needs a very high accuracy to represent precisely the instabilities that occur in the tokamaks. In order to minimize the memory requirement of the fully implicit solver and to access larger domain sizes, a direct resolution on a sub-part of the system is used as a preconditioner **??** coupled with a GMRES iterative solver, with the implementation provided by CERFACS [Fra+05], have been included a few years ago. Preconditioning transforms the original linear system $A\,x=b$ into an equivalent problem which is easier to solve by an iterative technique. A good preconditioner P is an approximation for A which can be efficiently inverted, chosen in a way that using $P^{-1}\,A$ or $A\,P^{-1}$ instead of A leads to a better convergence behaviour and more accurate solution. Usually, GMRES iterative solver is applied in collaboration with a preconditioner. The preconditioner typically incorporates information about the specific problem under consideration.

The JOREK *physics-based* preconditioner has been constructed by using the diagonal block for each of the n_{tor} Fourier modes (or toroidal planes) in the toroidal direction of the matrix A (see Equation 4.4). The preconditioner represents the linear part of each harmonic but neglects the interaction between harmonics (similar to a block-Jacobi preconditioning on a reordered matrix). So, in order to get a block-diagonal matrix with m independent submatrices on the diagonal, we set to zero the coefficients from the off-diagonal blocks of the original matrix A. The preconditioner P consists in the composition of m independent linear systems $(P_i^\star)_{i\in[1,m]}$, with $m = \frac{n_{tor}+1}{2}$. Considering interactions between the harmonics as nil, we can obtain a good preconditioner. Indeed, as the instabilities are small, the interactions between harmonics are small.

Practically, the set of processes is split in m independent MPI communicators, each of them treats only one single linear system $P_i^\star$ with a parallel sparse matrix direct solver (PaStiX or others). This preconditioned parallel approach avoids large costs in terms of memory consumption compared to the first approach that considers the whole linear system to solve (it saves the memory needed by the sparse solver to store the decomposition of the whole matrix A - *e.g.* L, U factors). However, the whole linear system A has also to be built (in parallel) in order to perform the matrix-vector multiplication needed by the GMRES. But, the cost in terms of memory and in terms of computation is far less than invoking the parallel sparse solver on the large linear system A.

This strategy improves the scalability (m independent systems), and the parallelization performance of the code. The bad point is that, in some specific circumstances, the iterative scheme may not converge.

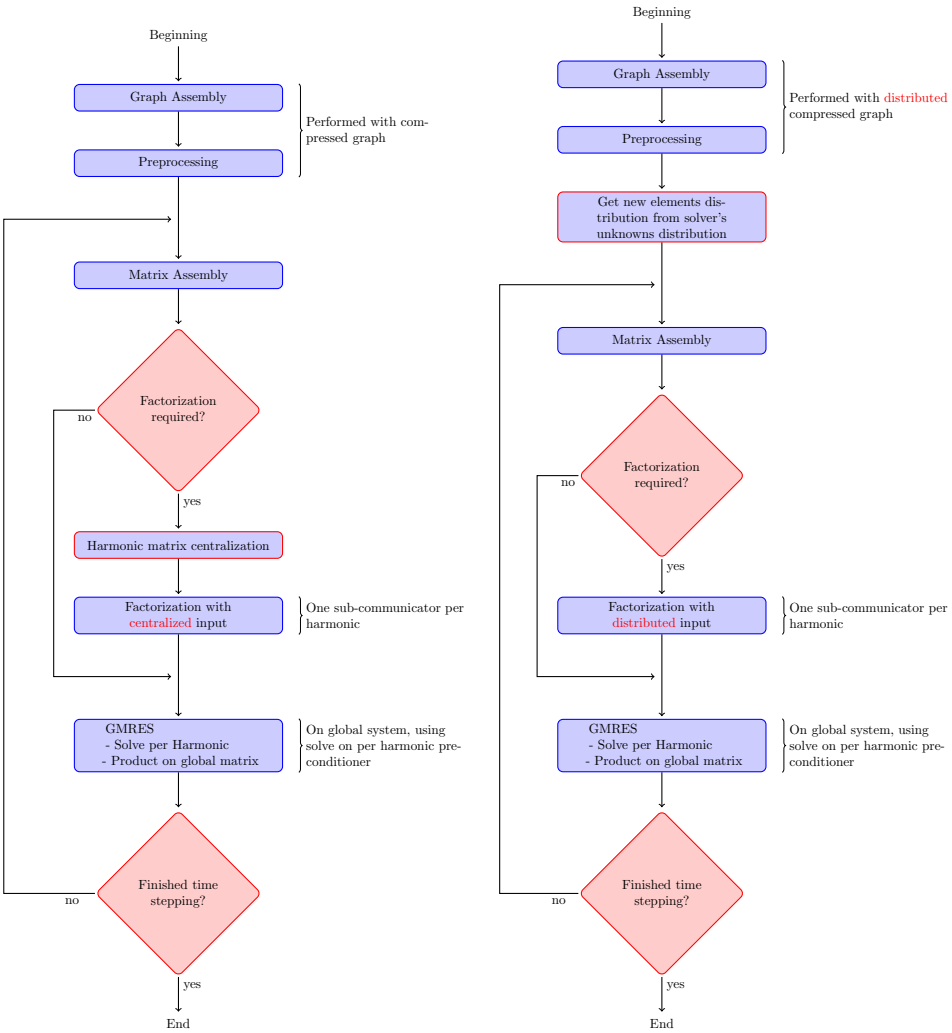
Figure 4.3(a) presents the solver's steps. First, the non-zero pattern of the matrix is built in a compressed form. Indeed, several ($n_{tor} \times n_{var}$) degrees of freedom are present for each mesh node and we need only one entry to represent all of them in the graph. Thus, the graph is smaller and the graph partitioning and symbolic factorization steps are faster to compute. Moreover, the graph is identical for each harmonic (as well as for the global matrix), and each call, on each of the m sub-communicators gives the same results. After that, one can assembly the global matrix, and if the factorization is required, the centralized harmonic matrix is gathered and factorized. Finally, the preconditioned GMRES algorithm can be run to obtain the solution.

4.2 Assembly step in JOREK

JOREK simulation uses 2D Bezier finite element to represent the sparse linear system. From these finite elements, one can build elementary matrices that are used to construct several matrices. First, the global problem, which is solved using a GMRES . Secondly, the harmonics matrices which are sub-part of the problem and the harmonic problems, which are solved using a sparse direct solver. The assembly algorithm, presented in Algorithm 13 (in Annex C), builds the global distributed matrix from which the harmonic matrices are deduced. As parallel domain decomposition, each process owns a list of local elements corresponding to the local rows of the matrix. The rows are distributed in a balanced way such that a row is owned by one and only one process. An element is local if one of its vertices corresponds to a local row. For each of the local elements, the MPI process builds the element matrix and inserts it into the assembled matrix. The built matrix is composed of blocks of $n_{tor} * n_{var}$ unknowns that can become rather large when the number of toroidal planes increases. Thus, we can compress the graph and consider only one entry per block to reduce significantly the preprocessing time.

Figure 4.4 describes a part of an element matrix (only two vertices are represented here for more readability). Each elementary matrix is composed of four vertices. For each vertex, there are $n_{order} + 1$ blocks of $n_{var} = 6$ variables, and each variable leads to one unknown per toroidal plane. The first toroidal plane corresponds to the first harmonic (in red in the elementary matrix), and each other harmonic is composed of two toroidal planes (second harmonic is in blue on the figure). Coupling variables, in yellow, will only be considered for the global matrix, used by the GMRES. Each harmonic is handled by one sub-communicator. Thus, the elementary matrices must be computed on each communicator, or communicated across communicators. The second option is chosen because of the heavy cost of computing an elementary matrix compared to its communication.

When solving the system, each harmonic is factorized to build a good preconditioner and inter-harmonic coupling is only taken into in the global GMRES algorithm. On Figure 4.5 we can see a mesh of elements (Figure 4.5(b)), describing the connectivity,



(a) Original steps with centralized calls to the direct solver

(b) Steps with new the distributed assembly

Figure 4.3 – JOREK main steps diagram. The global system is solved for each step of the time stepping using the preconditionned GMRES algorithm. Factorization is required only if the preconditioner is not accurate enough (ie. if the GMRES algorithm did not converge fast enough).

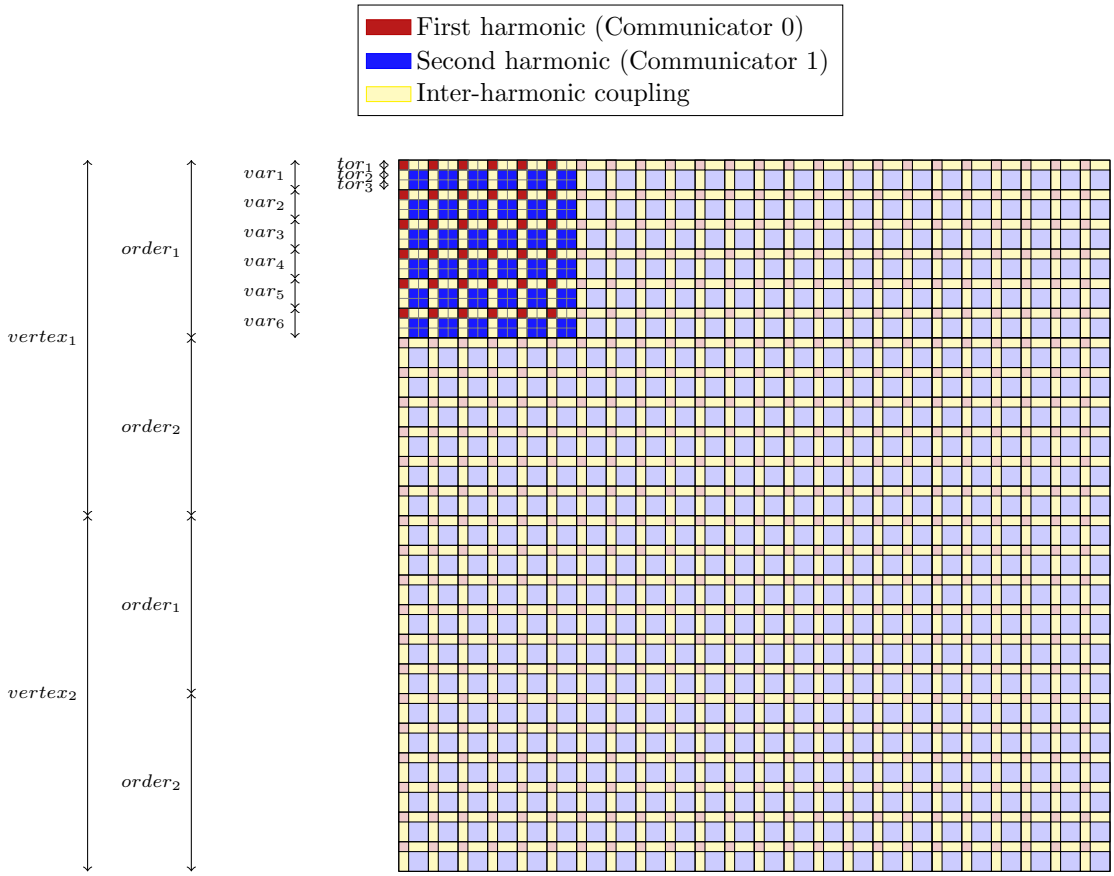
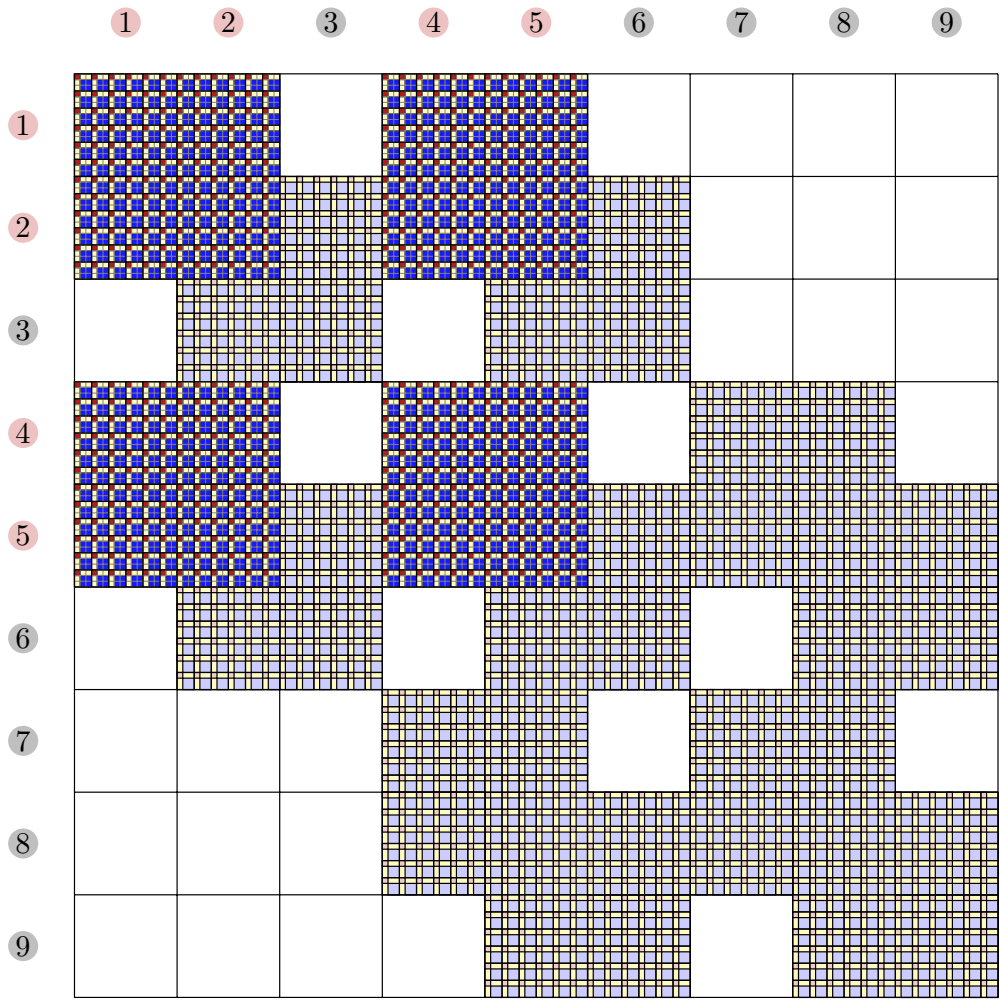


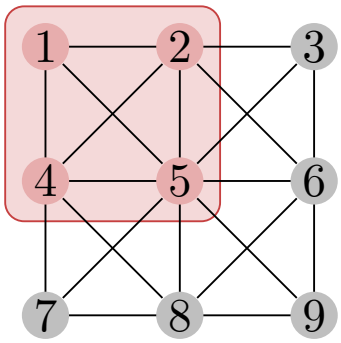
Figure 4.4 – Part of JOREK's elementary matrix, with 4 vertices, 3 toroidal planes, 6 variables and order 1.

and the corresponding global matrix on Figure 4.5(a). One can distinguish the harmonic matrices: in red the part of the matrix corresponding to the first harmonic (Figure 4.5(c)), and in blue the second one (Figure 4.5(d)). Of course, there could be more harmonics ($n_{tor} = 3$ up to 33 in typical simulations) but only two are represented in this graph for simplification. Furthermore, one can notice, on the harmonic figures (Figure 4.5(c) and Figure 4.5(d)), that only one entry per block of size $n_{var} \times n_{var}$ is represented for simplification. This two matrices are used to compute a preconditioner of the global system which also includes the coupling terms represented in yellow. The element {1, 2, 4, 5} is highlighted on all the figures comprised in Figure 4.5. The four highlighted blocks in Figure 4.5(a) correspond to the elementary matrix presented in Figure4.4. Once the red (respectively blue) coefficients have been extracted they correspond to the highlighted part in Figure 4.5(c) (respectively Figure 4.5(d)).

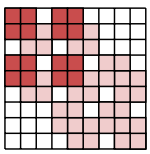
The assembly of the matrix results in a distributed matrix representing the system to solve. Locally, each process owns a set of rows of the full matrix. The rows are evenly



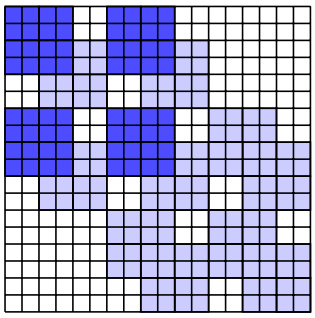
(a) Global matrix with all harmonics.



(b) Mesh of four elements and nine vertices.



(c) First harmonic.



(d) Second harmonic.

Figure 4.5 – JOREK's assembled matrix. On the two harmonic matrices we only show one entry for each block of six-by-six variables for more readability. First element, with vertices {1,2,4,5} is highlighted.

distributed among the MPI processes, and an elementary matrix is considered local and, thus, computed locally if at least one of its column is local.

This matrix, representing the whole problem, is used to compute the matrix-vector product operation of the GMRES algorithm. Once this global matrix has been computed, data are exchanged, using `MPI_Alltoallv` to obtain a centralized matrix per harmonic on each sub-communicator. These matrices are only built when the factorization is required, they are then transformed in the format required by the solver if necessary. In the case of the PaStiX solver, the Compress Sparse Column (CSC) format is used (see Annex D).

The fact that those matrices are centralized can create a bottleneck in term of memory usage for the simulation. This grows with the problem size, and is not reduced when the number of process increases. One goal in this study is to reduce this bottleneck by avoiding the centralization of the harmonic matrices.

4.3 Optimized distributed matrix assembly

Implementing an efficient distributed element matrices assembly is often a burden in simulation codes. Indeed, it is a memory bound operation that might require many data exchanges. However, it might reduce the memory peak by a large factor and potentially reduce the computational time.

Further more, the assembly step is present in numerous finite elements simulations, and the assembly functions are very similar. Thus, this step could be factorized to obtain a generic and optimized API for distributed assembly that minimizes the burden of the programmer. This is what we are addressing in this section. Specifically, in JOREK, this would allow one to use the distributed interface of the sparse direct solver without having to write a complex assembly routine.

Investigations about building a distributed matrix have already been performed in the context of PETSc linear algebra packaging tool [ASW07]. PETSc is a suite of data structures and routines to solve differential equations. This meta-package provides access to a large number of solvers including sparse direct solvers such as PaStiX. Using a hash-table based matrix where each coordinate (x, y) is associated to a key $k = x \times c + y$ (where c is the number of column), PETSc's developers can provide an efficient matrix assembly library. The main drawback of PETSc method is that, in order to be able to address multiple linear algebra tools, it stores its own version of the matrix that might be different from the structure used in the solver and thus, adds some computational overhead. The advantage is that it gives the possibility to test a large panel of solving techniques and libraries without modifying the code..

In JOREK code, using an external library to perform the parallel assembly, one can reduce the amount of code dedicated to the assembly part and focus on the specific algorithms and physics involved in the simulation.

4.3.1 Generic distributed finite element assembly oriented API

To help the user building efficiently the distributed matrix for the solver (here PaStiX), we proposed a new generic API: MURGE[13]. We designed this API to be able to couple any simulation code with any linear algebra solver. It proposes to suppress the PETSc structure overhead and to interact with the solver via several steps (as described in listing 5 from annex C). A quick performance comparison with the PETSc parallel linear system solving toolkit is presented in subsection 4.3.2.

To use a distributed sparse solver, MURGE API proposes to follow the following steps:

a) Initialization: allocates the required data structures for the linear algebra library and set the solver parameters;

b) Graph setting: assembles the graph of the matrix (ie. the matrix without the values). Once it has been performed, the linear algebra solver preprocessing steps (ordering, symbolic factorization, and data distribution in the case of PaStiX) can be triggered;

c) Remapping: retrieves the matrix distribution adapted to the solver and compute a new corresponding mesh distribution;

d) Matrix and right-hand-side setting: fills the matrix and right-hand-side with coefficients to solve the system;

e) Getting the solution: queries the solution vector;

f) Clean: deallocates all internal data structures;

The classical way to get the matrix (resp. graph) assembly is to set a block of values (resp. entries) corresponding to an elementary matrix at once. One can also use multiple degrees of freedom (DoF) per unknown and thus obtain a compressed graph, reducing memory cost, and preprocessing time. The advantage of using this API is that the communications and transformations required to obtain a distributed matrix in the format expected by the sparse linear solver library are optimized and hidden to the user.

Two options are proposed to the user:

- the first option is to ignore the solver's distribution and directly provide the matrix using the original element distribution. The user code is then simple at the cost of some extra communications to redistribute the matrix and fit the solver's requirement;

- the second option is to use an element mapping corresponding to the needs of the solver. We proposed a solution, simplifying the user's code while obtaining an element distribution corresponding to the solver's one. The user can use the `MURGE_GetLocalElementList()` function that will perform the graph construction

[13] `http://murge.gforge.inria.fr`

and compute the new elements distribution for the user. This function requires the number of elements in the graph, a function to retrieve the list of vertices of each element following its index, and a distribution strategy. It provides to the user a new list of local elements.

One can either use one of the following strategy:

`MURGE_DUPLICATE_ELEMENTS`: considers one element is local if at least one of its vertex is local;

`MURGE_DISTRIBUTE_ELEMENTS`: provides an even elements distribution. Elements are first distributed such that the process owning the largest number of vertices of each element owns it. Then, the list is balanced by moving the least "local" elements of overloaded processes to processes with less elements that own the largest part of the element.

- First, we have the PETSc solution presented in Figure 4.6(a). The user provides the matrix that will be assembled in PETSc framework and then converted into the solver format. After that conversion the solver's library can be called and will perform the preprocessing steps (ordering, symbolic factorization, and data distribution in the case of PaStiX). Then the matrix is redistributed to fit the solver's requirements, and the factorization and solving steps are called.

- Secondly, Figure 4.6(b) present the MURGE API, which aims at building directly the matrix in the format required by the solver's library. In that case, the conversion steps are not required anymore.

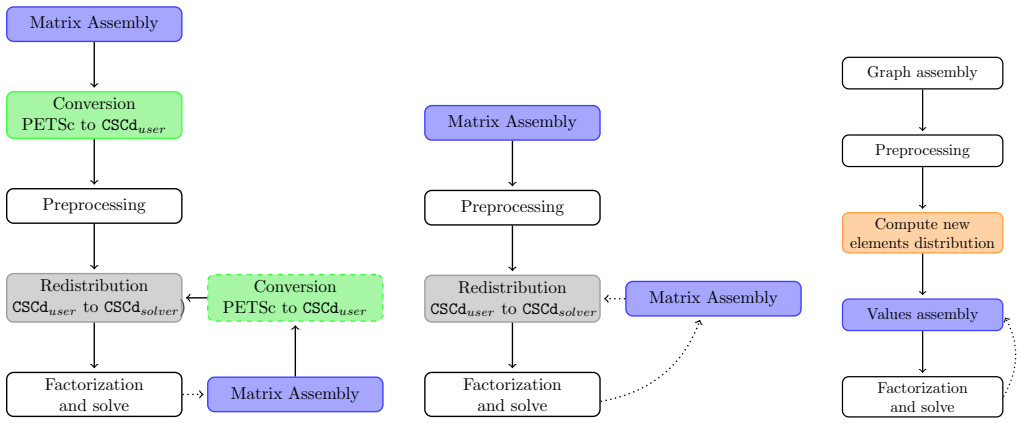
- Finally, Figure 4.6(c) present the possibility to build separately the graph in MURGE to be able to obtain the solver distribution before building the matrix using a domain decomposition of the mesh corresponding to the solver's matrix one. In that case, the matrix redistribution can be suppressed. This is automatically the case when one uses the `MURGE_GetLocalElementList()`.

In all those situation, if another factorization is required, new values would be inserted via a second assembly. In the case of PETSc, it would also require the conversion to be performed on the newly assembled matrix, and the factorization, and solve would be called again. Only the solution presented in Figure 4.6(c) permit to avoid the redistribution step.

During the matrix assembly phase multiple use cases also exist:

- one can provide entries one by one, which is convenient to insert the boundary conditions;

- one can supply entries as a block element by element. That version is well suited to the finite element matrix assembly;



(a) Assembly in PETSc.

(b) Assembly in MURGE.

(c) Assembly in MURGE in the solver's distribution.

Figure 4.6 – Different assembly methods.

- one can enter a block of entries corresponding to several elements at once. This possibility can reduce the overhead due to the function call.

In important remark is that the API also provides a sparse matrix-vector product such that one can build an iterative solver, possibly using the linear solver as a preconditioner, without storing the distributed matrix in its own structures.

In next subsection, we will evaluate the performance of the different assembly methods and compare it with PETSc implementation.

4.3.2 Comparison with PETSc

This subsection presents a comparison of the computational costs of the PETSc assembly routines versus the PaStiX implementation of MURGE. Those two implementations of the matrix assembly step are compared using a similar testing example that builds and solves the matrix associated to a 3D Laplacian problem discretized with linear finite elements. The Laplacian is computed on a cubic 3D grid with M^3 elements with and one degree of freedom per vertex. This gives a matrix of size $(M+1)^3$ with nnz entries, where:

$$nnz = 3^3 * (M+1)^3 - (3^3 - 1) * ((M+1)^3 - (M-1)^3)$$

. Several implementations of the 3D Laplacian test case API have been evaluated:

- *simple* uses MURGE API and enters all the local entries one by one;
- *blocked* enters the local entries by block of values, called elementary matrices, associated to each local element;

- *getELementList* uses `MURGE_GetLocalElementList()` function to obtain a local, evenly distributed, local elements list that is used for the elementary matrices insertion;
- *PETSc* is similar to the *blocked* one where entries are entered by elementary matrices, but using PETSc API.

The *getElementList* is the only version that uses an element distribution corresponding to the solver and can then avoid the redistribution as shown earlier in Figure 4.6(c). In all cases, the distributed implementation of PaStiX is called in a similar way for a fair comparison.[14]

Figure 4.7 presents timings during the execution of the presented test cases. We present results for 1, 2, 4, 8, 16, and 32 nodes with one process per 12-core node. These experiments have been performed on nodes from Avakas cluster at University of Bordeaux (see 3.2.1, page 93), and uses $M = 50$ elements on each edge of the cube. Several timings are shown in the figure (the colors correspond to the ones in Figure 4.6):

- the blue part corresponds to the assembly of the matrix,
- the red part represents the matrix conversion inside PETSc,
- the yellow part is the time constructing the new elements list (with *getElementList* test case),
- the grey part represents the rest of the overhead of the function used to obtain the solution.

One can notice that all different MURGE implementations obtain similar timing. PETSc implementation seems to be penalized by its generecity that allows it to address a large number of solvers within one packager. Indeed, both the assembly and the overhead of the API are larger. Moreover the conversion time, which is, hopefully, not too long, is only present on the PETSc case. However, when the number of MPI nodes increases, PETSc library can catch up with MURGE API implementation. In all cases, if the assembly scales whereas the redistribution time increases with the number of nodes. The time to obtain the solution, in PETSc case, is longer with one node. This may be due to the particular treatment it receives in the PETSc wrapper. One has to notice that, with the `getElementList` method, the time getting solution part is paid only once for a given matrix structure (i.e. a given graph), which can save time compared to the redistribution time, which is paid before each factorization, if the number of nodes increases or if the matrix is factorized several times.

Thus, we have shown here the benefits in term of development cost and performance that one can expect from a generic API to build a distributed matrix and call a sparse linear solver. This has been applied to the case of PaStiX direct solver but could be extended to any sparse linear solver that uses a distributed sparse matrix as input.

[14] We had to modify the PaStiX wrapper in PETSc to be able to call the distributed version of PaStiX and obtain a more fair comparison. This patch will be proposed to the PETSc developer team.

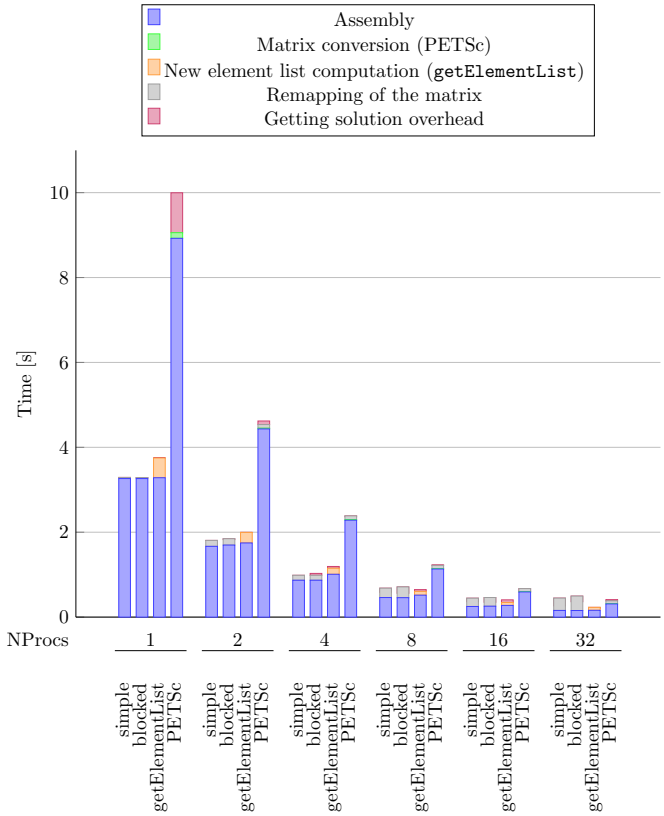


Figure 4.7 – Timing scaling study comparing PETSc and MURGE assembly.

4.4 Integration into JOREK

The previous section validated the MURGE API using a simple 3D Laplacian example. This section describes the integration of this API in the parallel MHD simulation code: JOREK. Memory consumption is the main bottleneck in JOREK. In order to reduce the memory peak when the system is solved, and improve the memory scaling, we replace the existing centralized interface by the API presented in previous section. Indeed, the actual interface gathers the entire harmonic sub-matrices (in blue and red on Figure 4.5) on each sub-process involved in its factorization. With the new API, the centralized harmonic matrices do not have to be stored, but can be distributed among all nodes of a single harmonic communicator. The next subsection describes the modification on the global steps of the simulation, and the new assembly, using MURGE API. A second subsection presents timing and memory studies inside the JOREK simulation code.

4.4.1 Implementation

At the beginning of this thesis, JOREK was centralizing the harmonics matrices before calling a sparse linear algebra library (either MUMPS or PaStiX) to solve the system. This leads to a bottleneck as the centralized storage of the matrix cannot scale. Thus, a distributed interface to the solver has to be used to obtain memory scaling. In this case, two approaches can be foreseen:

- The simplest solution uses the original mesh distribution and let the sparse linear algebra solver remap the matrix to obtain the distribution it requires factorizing the matrix;
- A more efficient method consists in using the solver's distribution to remap the mesh as seen previously in Figure 4.6(c). This solution is designed to avoid communications as the computation of contribution from any cell of the mesh can be computed by any MPI process indifferently.

Figure 4.3(b) presents the new solving steps in JOREK with the integration of the distributed interface to the PaStiX solver. It uses MURGE with the `MURGE_GetLocalElementsList()` function, presented in Figure 4.6(c), adopting a new elements list adapted to the solver's distribution. Thus, the centralization of the harmonics matrix step is removed, and the calls to PaStiX use a distributed interface, reducing the memory footprint. The data distribution can be either chosen evenly balanced, with `MURGE_DISTRIBUTE_ELEMENTS`, or the element can be duplicated, when one use `MURGE_DUPLICATE_ELEMENTS`.

The complex part when using distributed calls to the solver in JOREK, even using MURGE API, resides in the assembly of the matrices (see Algorithm 15 in Annex C). Indeed, in JOREK, we have to assemble multiple matrices in one assembly. Each elementary matrix (as shown in Figure 4.5) corresponds to entries that have to be accumulated into the global problem matrix, but also to the harmonics. These harmonics are either local, or belong to another communicator. Thus, for each computed elementary matrix,

the computing process has to insert it into the global problem, but also to extract the part belonging to each harmonic. These parts either correspond to the local harmonic (and can be inserted using MURGE API), or to another harmonic. In that case, the computing process will have to send it to a process of the other harmonic communicator.

To simplify the matrix assembly, we have to obtain the same elements distribution on each harmonic sub-communicator. Indeed, if the processes with the same rank in each harmonic sub-communicator own the same list of local elements, we can divide the local elements list among these processes (see Algorithm 15 in Annex C).

If we consider Figure 4.5 with a global communicator with four process, we have two processes per harmonic: P_1^1 and P_2^1 on harmonic one and P_1^2 and P_2^2 for harmonic two. Let us suppose that process one from each sub-communicator (P_1^1 and P_1^2) owns columns one to four and process two (P_2^1 and P_2^2) owns columns five to nine. Then, the element composed of vertices one, two, four, and five, highlighted in Figure 4.5 is shared between the two processes on each sub-communicator with three columns on process one and one on process two. Then, the element belongs to processes P_1^1 and P_1^2. Indeed, P_1^1 and P_1^2 will share exactly the same list of local elements. We will arbitrary decide that one of the two processes will compute the elementary matrix and communicate it to the other process. Each process will compute half of the elements and communicate it to the other processes of same rank in other harmonic communicators. All the communications required for the assembly of the harmonics matrices are performed, if factorization is required, during the assembly step.

The element distribution depends on the graph preprocessing steps that determine the distribution of the factorized matrix. The result of the preprocessing steps depends on the pattern of the graph, which is identical for each harmonic (as one can see on Figure 4.5), but also on the number of degree of freedom. This number of degree of freedom is smaller for the first harmonic that corresponds to only one toroidal plane (whereas other harmonics correspond to a pair of toroidal planes). We have to ensure to obtain the same matrix distribution for each harmonic.

This procedure allows to remove the Harmonic matrix centralization step. Thus, using the new assembly, one can expect a better memory scaling from the distributed calls to the solver's library.

4.4.2 Timing and memory scaling study

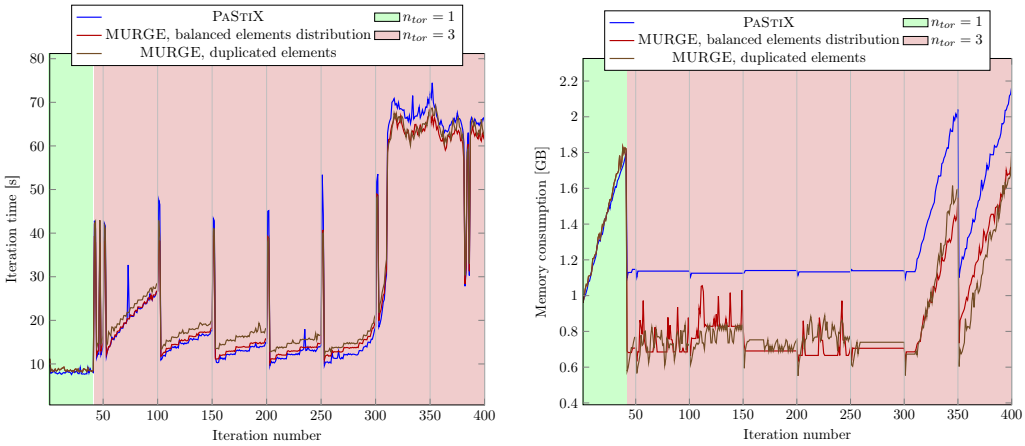
This section presents some experiments on the integration of MURGE inside JOREK. All the experiments have been performed on the hybrid partition from the Curie cluster at TGCC in France. One MPI process is executed on each node of eight cores. Inside the nodes, the parallelism is handled with one thread per core using OpenMP in JOREK and POSIX Threads in PaStiX.

Figure 4.8(a) presents the amount of time spent in each time step iterations during an execution of JOREK on a medium size problem: model 302. The original PaStiX integration, modified to use blocks of $n_{tor} \times n_{var}$ degrees of freedoms, is compared with MURGE integration. Two implementations, using the MURGE API, are presented here. They both use the `MURGE_GetLocalElementsList()` from MURGE to retrieve

the elements list. The first one, in red, duplicates elements when their columns are spread onto multiple processes from the harmonic communicator. For example, on model 302, 501 elementary matrices out of 4074 are computed twice on different processes). The second one, in brown, attributes each element only once on each sub-communicator. Thus, the elementary matrices need to be exchanged. The figure does not present the "Grad-Shafranov" equilibrium which uses a different code which is not considered here. The second phase of the simulation, in green, uses only one toroidal plane whereas the second phase uses three toroidal planes. During the second phase, the GMRES solver is disabled and the matrix is solved using a direct method. This phase is used when a X-point is involved, it requires the high precision of the direct method combined with small increasing time steps to find a dynamic equilibrium. MURGE interface is not used in this phase, this is why the results remain identical.
During the second phase, we can distinguish two types of iterations: the short ones, when no factorization is performed (i.e. the iterative solver is used with preconditioning); and the longer ones, when the iterative solver cannot converge to a satisfying solution. Then, a factorization of the harmonic matrices is required to improve the convergence.
As expected, one can notice that when a factorization occurs, the MURGE API accelerates the computations, thanks to the improved assembly of the matrix. Communications are minimized inside the API compared to the `MPI_Alltoallv` that is used when using original PaStiX interface.
Shorter iterations are comparable with MURGE. In the duplicated elements case, the larger number of elementary matrices computed and the imbalance in the elements distribution explain the additional time. With the second option, the global iteration time is competitive with the PaStiX centralized one. MURGE API cannot improve the communication avoided distributed assembly of the global matrix but improvement in the multi-threaded matrix-product could be investigated to improve the speed of these iterations.

Figure 4.8(b) presents the memory consumption of the three implementations on the same test case. About 15% to 30% of the memory can be saved during the simulation when $n_tor = 3$. The peak memory, at the end of the simulation, is then reduced by 20%. The duplicated elements version consumes about the same amount of memory than the one with no replication.

Figure 4.9 and Figure 4.10 present results of the same experiment on a larger test case: model 303. Here the initialization phase with one toroidal plane last for 120 iterations, then they use respectively three and seven toroidal planes in a second phase. With three toroidal planes (Figure 4.9), one can make similar observations on this model as on the model 302: iterations without factorization are taking the same time with the centralized API and with MURGE, while the iterations with a factorization are improved. We can save 25% of the memory during the second phase but as the peak was reached during the first phase it is not reduced. With seven toroidal planes (Figure 4.9), the GMRES only iterations are longer. A longer time is spent in the assembly loop that is more complex than with the centralized harmonics because of the differences in the elementary matrices distribution. We need to identify which step of the assembly loop is not optimal and



(a) Iteration time.

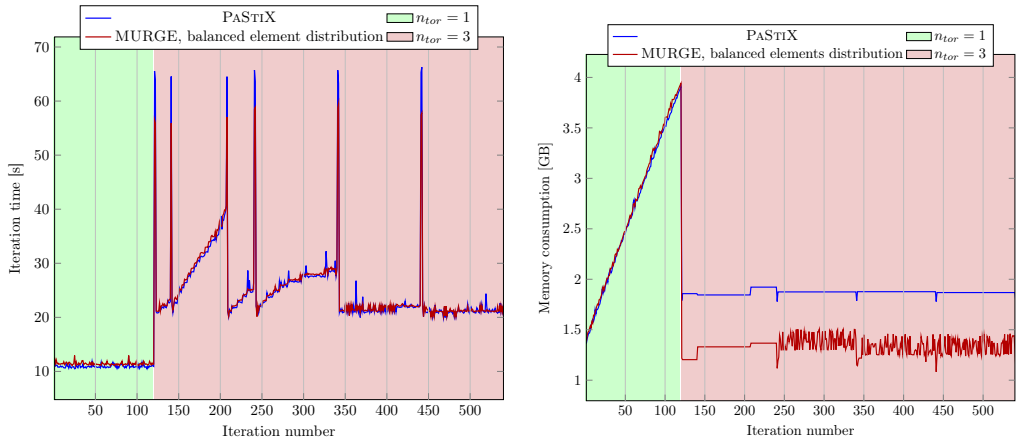
(b) Memory consumption.

Figure 4.8 – Comparison of iteration time and memory consumption for each time step of the simulation on model 302. On eight nodes of eight cores, on Curie cluster hybrid partition. Model 302 was run with default parameters, in particular $n_{tor} = 3$ (2 harmonics).

correct it. However, the memory consumption is also reduced and the peak memory can be reduced by 30%. Thus, using MURGE this execution can be run with a lower memory requirement and executed on smaller configuration. Thus, users can execute the simulation with a larger number of toroidal planes.

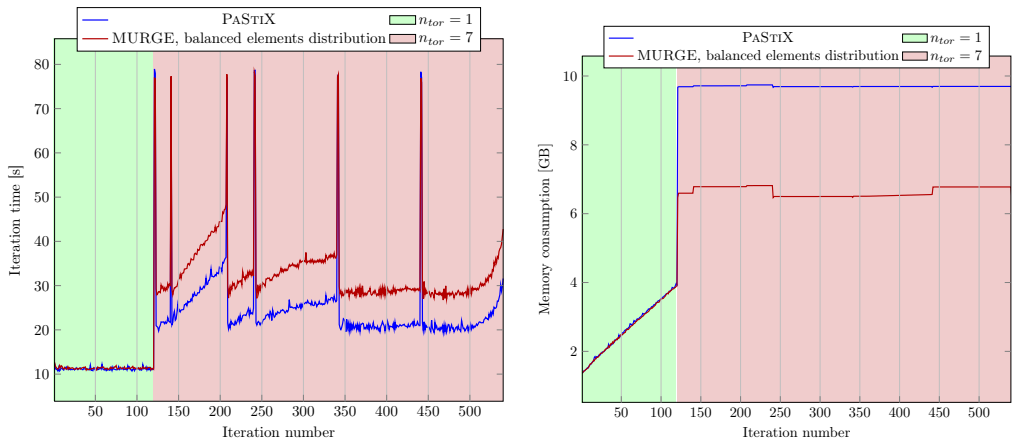
4.5 Discussion

In this chapter, we explained how we could improve the memory and time scaling of a numerical simulation using a generic matrix assembly API developed above the PaStiX solver. Using an implementation specific to the solver one can reach higher performance. This finite elements oriented API also simplify the development of a distributed matrix assembly for the numerical simulation developer. In a large scale simulation code, JOREK, we could achieve both good performance and less memory consumption through this API. All this study is related to PaStiX integration inside JOREK but the same API could be used to produce a distributed assembly for another solver or/and another finite elements based simulation code.



(a) Iteration time. (b) Memory consumption.

Figure 4.9 – Comparison of iteration time and memory consumption during for each time step of the simulation on model 303. On eight nodes of eight cores, on Curie cluster hybrid partition. Model 303 was run with default parameters, in particular $n_{tor} = 3$ (2 harmonics).



(a) Iteration time. (b) Memory consumption.

Figure 4.10 – Comparison of iteration time and memory consumption for each time step of the simulation on model 303. On eight nodes of eight cores, on Curie cluster hybrid partition. Model 303 was run with default parameters, except that $n_{tor} = 7$ (4 harmonics).

Conclusion and perspectives

To compensate for the limited clock frequency and the high consumption of basic cores, manufacturers proposed more specialized, highly parallel, accelerators. Nowadays, half of the first ten machines in the top500 ranking [Meu+14] are heterogeneous clusters. If dense linear algebra community already addressed efficiently these new heterogeneous architectures this was not the case of sparse linear algebra. We investigated the possibility to solve efficiently a sparse system, using direct methods, on a distributed cluster of heterogeneous nodes.

The main objective of this thesis was to evaluate the feasibility of an efficient sparse direct method on top of generic task-based runtime systems to target distributed heterogeneous nodes. These works were integrated in the PaStiX sparse linear solver library developed in Bordeaux. To implement such a heterogeneous algorithm, we decided to use the task-based runtime systems that were widely used in the context of dense linear algebra.

In Chapter 2, we described the implementation of a shared-memory sparse factorization algorithm using two task-based runtime systems; namely, StarPU and PaRSEC. We estimated the efficiency of those new implementation. We have shown that using task-based runtime systems in a shared memory context, one could reach similar performance to the finely tuned original PaStiX scheduler. Then, we added GPU support to our algorithm. To achieve that, we modified an existing matrix-matrix product GPU kernel to fit the sparse structures used in the sparse direct linear solver. We provided this new kernel to the runtime system and developed a scheduling strategy to help the runtime system mapping the operation on either CPU or GPU. Doing so, we could achieve a speedup of up to 2.86 on one heterogeneous node.

Chapter 3 extended this study to the distributed case. We implemented two versions of the distributed algorithm, *fan-in* and *fan-out*, using task-based runtime systems and compared it to the original PaStiX static and dynamic scheduler implementations. The task-based runtime systems implementation can outperform, under certain conditions, the default static scheduler. We obtained a speedup of up to 1.4 on 4 nodes of 12 cores and 2 GPU. However, the dynamic scheduler implemented in [Fav09] is still superior on a large number of nodes. Indeed, the communication management by the generic runtime system creates an overhead that has to be reduced.

Chapter 4 considered the distributed matrix assembly step of a simulation code. To address this issue, we proposed a new generic API that can be implemented on

top of most of the sparse solvers. We implemented it on top of the PaStiX solver and estimated its performance in comparison with the PETSc linear algebra assembly functionality. Using this API, we could replace the centralized matrix assembly in JOREK by a distributed one and save up to 30% of the memory. Once this memory bottleneck has been removed, JOREK is now able to target larger problem sizes.

The outcomes of this thesis are opening toward the usage and optimization of task-based runtime systems for sparse linear algebra. Optimizations regarding the graph partitioning step are currently investigated. The goal is to reorder the unknowns to gather the small blocks into larger ones that will lead to more efficient BLAS calls. This will improve the performance of the decomposition in the general case and particularly reduce the relative overhead of task-based runtime systems, and generate tasks more suited to accelerators. In Chapter 2, we showed that the static placement of tasks on GPUs is well tuned for shared memory execution on the test cases presented in this thesis. However, the placement would have to be modified when the size of the matrices differs in order or when it is distributed on multiple nodes. By enforcing the locality decision following arbitrary criteria, we limit the control of the algorithm. A finer API to control the placement of computation would provide better performance.

We could also separate the panel factorization into two kernels: the diagonal block factorization and the triangular system solving on the off-diagonal blocks. Doing so we could decide to execute the solution of the off-diagonal triangular system on the accelerators. For this operation, on GPU, the cuBLAS routine already exists, we only would have to provide it to the runtime system so that it could use the accelerator to perform these new kernels. This is also possible for the diagonal block factorization kernel although this one is usually less efficient on GPU, especially with the block sizes encountered in sparse direct solvers.

In this thesis, we only considered GPUs, but we should also address Intel Xeon Phi accelerators. Two approaches can be investigated: one could execute the whole sparse factorization on the Intel Xeon Phi, or this last one could be used to offload part of the computation through specialized kernels. The first approach is limited by the size of the memory available on the board. One could investigate the second by providing a new specialized kernel to the runtime system implementation of the solver.

In a distributed context, the task-based implementation of the factorization is also a good framework to experiment a mixed *fan-in/fan-out* [Ash+90] implementation of the factorization. It could be used, particularly, for a Schur complement computation exploited in modern hybrid solvers [GH09; RBH12]. Finally, using a window to limit the number of submitted tasks to the runtime system might improve the scaling of the MPI implementation on larger problems by reducing the number of communication buffers handled simultaneously.

Considering JOREK and the matrix assembly interface, we have to investigate the use of the generic runtime for this step as well as for the matrix-vector product in order to be able to exploit accelerators. Also, a tighter integration with PaStiX could insert values directly inside the factorized matrix structure, avoiding the usage of all the intermediate structures at the cost of a more complex code for the interface. Increasing

the integration of finite elements could improve the scaling of the simulation. Indeed, a tighter integration with the mesh structure could allow one to find a data mapping that will fit the solver's computation, the matrix mapping, and give a balanced distribution of the elements of the mesh adapted to the simulation. We could also investigate the possibility to use two dedicated communicators for the solver and for the assembly that would simplify the user assembly code in simulations using a complex solving method with multiple independent problems such as JOREK.

Thus, in this thesis, we showed that one can obtain an efficient parallel sparse direct solver on heterogeneous clusters using task-based runtime systems with an adapted task granularity. We also provided an easy-to-use API to perform an optimized distributed assembly for a sparse direct solver. Both have been integrated in a production code to improve its performance.

Bibliography

[ADD96] P. Amestoy, T. A. Davis, and I. S. Duff. *An Approximate Minimum Degree Ordering Algorithm.* In: *SIAM J. Matrix Anal. and Appl.* (1996), pp. 886–905 (cit. on pp. 20, 22).

[ADL98] P. Amestoy, I. S. Duff, and J.-Y. L'Excellent. *Multifrontal Parallel Distributed Symmetric and Unsymmetric Solvers.* In: *Comput. Methods Appl. Mech. Eng* 184 (1998), pp. 501–520 (cit. on pp. 24, 29).

[AEL90] A. Ashcraft, S. C. Eisenstat, and J. W.-H. Liu. *A fan-in algorithm for distributed sparse numerical factorization.* In: *SIAM Journal on Scientific and Statistical Computing* 11 (1990), pp. 593–599 (cit. on p. 27).

[Agu+09] 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.* In: *Journal of Physics: Conference Series.* Vol. 180. 00179. IOP Publishing, 2009, p. 012037 (cit. on pp. 15, 31).

[AMD] AMD. *The AMD Core Math Library.* `http://developer.amd.com/cpu/libraries/acml/Pages/default.aspx` (cit. on p. 17).

[Ame+01] P. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. *A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling.* In: *SIMAX* 23.1 (2001), pp. 15–41 (cit. on pp. 24, 26, 29).

[And+99] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammerling, A. McKenney, et al. *LAPACK Users' guide.* Vol. 9. Siam, 1999 (cit. on p. 18).

[Ash+90] C. Ashcraft, S. C. Eisenstat, J. W. Liu, and A. H. Sherman. *A Comparison of Three Column-Based Distributed Sparse Factorization Schemes.* DTIC Document, 1990 (cit. on p. 124).

[Ash93] A. Ashcraft. *The fan-both family of column-based distributed Cholesky factorization algorithms.* In: *Graph Theory and Sparse Matrix Computation, IMA* 56 (1993), pp. 159–190 (cit. on p. 27).

[ASW07] M. Aspnäs, A. Signell, and J. Westerholm. *Efficient assembly of sparse matrices using hashing.* In: *Applied Parallel Computing. State of the Art in Scientific Computing* (2007), pp. 900–907 (cit. on p. 112).

[Aug+11] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. *StarPU: a unified platform for task scheduling on heterogeneous multicore architectures.* In: *Concurrency and Computation: Practice and Experience* 23.2 (2011). ISSN: 1532-0634 (cit. on pp. 14, 15, 44).

[Aug11] C. Augonnet. "Scheduling Tasks over Multicore machines enhanced with acelerators: a Runtime System's Perspective". PhD thesis. Université Bordeaux 1, Dec. 9, 2011 (cit. on p. 14).

[Ayg+09] 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: *Euro-Par.* Ed. by H. J. Sips, D. H. J. Epema, and H.-X. Lin. Vol. 5704. Lecture Notes in Computer Science. Springer, 2009, pp. 851–862. ISBN: 978-3-642-03868-6 (cit. on p. 15).

[Bad+09] 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.* In: *Concurrency and Computation: Practice and Experience* 21.18 (2009), pp. 2438–2456 (cit. on p. 15).

[Bel+09] P. Bellens, J. M. Pérez, F. Cabarcas, A. Ramírez, R. M. Badia, and J. Labarta. *CellSs: Scheduling techniques to better exploit memory hierarchy.* In: *Scientific Programming* 17.1-2 (2009), pp. 77–95 (cit. on p. 15).

[Bie+10] P. Bientinesi, V. Eijkhout, K. Kim, J. Kurtz, and R. van de Geijn. *Sparse direct factorizations through unassembled hyper-matrices.* In: *Computer Methods in Applied Mechanics and Engineering* 199.9,12 (2010), pp. 430–438. ISSN: 0045-7825. DOI: `10.1016/j.cma.2009.07.012` (cit. on p. 34).

[Bla+96] L. S. Blackford, J. Choi, A. Cleary, A. Petitet, R. C. Whaley, J. W. Demmel, I. Dhillon, K. Stanley, J. Dongarra, S. Hammarling, G. Henry, and D. Walker. *ScaLAPACK: a portable linear algebra library for distributed memory computers - design issues and performance.* In: *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM).* Pittsburgh, Pennsylvania, United States: IEEE Computer Society, 1996, p. 5. ISBN: 0-89791-854-1. DOI: `http://doi.acm.org/10.1145/369028.369038` (cit. on p. 18).

[Bos+11] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra. *Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA.* In: *12th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC'11).* 2011 (cit. on pp. 32, 44).

[Bos+12] G. Bosilca, A. Bouteiller, A. Danalis, T. Hérault, P. Lemarinier, and J. Dongarra. *DAGuE: A generic distributed DAG engine for High Performance Computing.* In: *Parallel Computing* 38.1-2 (2012) (cit. on pp. 14, 15, 41).

[Bos14] G. Bosilca. *Dense linear algebra on distributed heterogeneous hardware with a symbolic dag approach.* In: *Scalable Computing and Communications: Theory and Practice* (2014) (cit. on p. 2).

[Bra86] A. Brandt. *Algebraic multigrid theory: The symmetric case.* In: *Applied mathematics and computation* 19.1 (1986), pp. 23–56 (cit. on p. 19).

[Bro+09] F. Broquedis, N. Furmento, B. Goglin, R. Namyst, and P.-A. Wacrenier. *Dynamic task and data placement over NUMA architectures: an OpenMP runtime perspective.* In: *Evolving OpenMP in an Age of Extreme Parallelism.* Springer, 2009, pp. 79–92 (cit. on p. 11).

[Bro+10] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. *hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications.* In: *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2010).* Pisa, Italia: IEEE Computer Society Press, Feb. 2010, pp. 180–186. DOI: 10.1109/PDP.2010.67 (cit. on p. 40).

[But+06] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov. *The Impact of Multicore on Math Software.* In: *PARA 2006* (2006) (cit. on p. 31).

[But+09] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. *A class of parallel tiled linear algebra algorithms for multicore architectures.* In: *Parallel Computing* 35.1 (2009), pp. 38–53 (cit. on pp. 31, 49).

[But13] A. Buttari. *Fine-grained multithreading for the multifrontal QR factorization of sparse matrices.* In: *SIAM Journal on Scientific Computing* 35.4 (2013), pp. C323–C345 (cit. on p. 32).

[Car+99] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. *Introduction to UPC and Language Specification.* Tech. rep. CCS-TR-99-157. George Mason University, May 1999 (cit. on p. 13).

[CH08] O. Czarny and G. Huysmans. *Bézier surfaces and finite elements for MHD simulations.* In: *Journal of computational physics* 227.16 (2008), pp. 7423–7445 (cit. on pp. 103, 105).

[Cha+08] E. Chan, F. G. V. Zee, P. Bientinesi, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. A. van de Geijn. *SuperMatrix: a multithreaded runtime scheduling system for algorithms-by-blocks.* In: *PPOPP.* Ed. by S. Chatterjee and M. L. Scott. ACM, 2008, pp. 123–132. ISBN: 978-1-59593-795-7 (cit. on p. 16).

[Cha01] R. Chandra. *Parallel programming in OpenMP.* Morgan Kaufmann, 2001 (cit. on p. 11).

[Che+08] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam. *Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/-downdate.* In: *ACM Transactions on Mathematical Software (TOMS)* 35.3 (2008), p. 22 (cit. on p. 29).

[Cho+96] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley. *A proposal for a set of parallel basic linear algebra subprograms.* In: *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science.* Springer, 1996, pp. 107–114 (cit. on p. 18).

[CJ99] M. Cosnard and E. Jeannot. *Compact DAG representation and its dynamic scheduling.* In: *Journal of Parallel and Distributed Computing* 58.3 (1999), pp. 487–514 (cit. on p. 15).

[CJY04] M. Cosnard, E. Jeannot, and T. Yang. *Compact DAG Representation and its Symbolic Scheduling.* In: *Journal of Parallel and Distributed Computing* 64.8 (2004), pp. 921–935 (cit. on p. 44).

[CM69] E. Cuthill and J. McKee. *Reducing the bandwidth of sparse symmetric matrices.* In: *Proceedings of the 1969 24th national conference.* New York, NY, USA: ACM, 1969, pp. 157–172. DOI: `http://doi.acm.org/10.1145/800195.805928` (cit. on p. 22).

[CR89] P. Charrier and J. Roman. *Algorithmique et calculs de complexité pour un solveur de type dissections emboitées.* In: *Numerische Mathematik* 55 (1989), pp. 463–476 (cit. on p. 22).

[Cra+12] T. Cramer, D. Schmidl, M. Klemm, and D. an Mey. *OpenMP Programming on Intel R Xeon Phi TM Coprocessors: An Early Performance Comparison.* In: (2012) (cit. on p. 14).

[CSB07] M. Christen, O. Schenk, and H. Burkhart. *General-purpose sparse matrix building blocks using the NVIDIA CUDA technology platform.* In: *First Workshop on General Purpose Processing on Graphics Processing Units.* Citeseer, 2007 (cit. on p. 32).

[Dav04] T. A. Davis. *Algorithm 832: UMFPACK V4. 3—an unsymmetric-pattern multifrontal method.* In: *ACM Transactions on Mathematical Software (TOMS)* 30.2 (2004), pp. 196–199 (cit. on p. 29).

[Dav06] T. A. Davis. *Direct methods for sparse linear systems.* Vol. 2. Siam, 2006 (cit. on p. 18).

[Dem+99] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. Liu. *A supernodal approach to sparse partial pivoting.* In: *SIAM Journal on Matrix Analysis and Applications* 20.3 (1999), pp. 720–755 (cit. on p. 30).

[DH11] T. A. Davis and Y. Hu. *The University of Florida sparse matrix collection.* In: *ACM Transactions on Mathematical Software (TOMS)* 38.1 (2011), p. 1 (cit. on p. 55).

[Dia+08] J. R. Diamond, B. Robatmili, S. W. Keckler, R. A. v. d. Geijn, K. Goto, and D. Burger. *High performance dense linear algebra on a spatially distributed processor.* In: *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming.* Salt Lake City, UT, USA: ACM, 2008, pp. 63–72. ISBN: 978-1-59593-795-7. DOI: `http://doi.acm.org/10.1145/1345206.1345218` (cit. on p. 17).

[Don+88] J. Dongarra, J. Du Croz, . Hammarling, and R. J. Hanson. *An extended set of FORTRAN basic linear algebra subprograms.* In: *ACM Trans. Math. Softw.* 14.1 (1988), pp. 1–17. ISSN: 0098-3500. DOI: `http://doi.acm.org/10.1145/42288.42291` (cit. on p. 17).

[Don+90] J. Dongarra, J. D. Croz, S. Hammarling, and I. S. Duff. *A set of level 3 basic linear algebra subprograms.* In: *ACM Trans. Math. Softw.* 16.1 (1990), pp. 1–17. ISSN: 0098-3500. DOI: `http://doi.acm.org/10.1145/77626.79170` (cit. on p. 17).

[Don03] J. Dongarra. *Freely available software for linear algebra on the web.* In: *URL: http://www. netlib. org/utk/people/JackDongarra/la-sw. html (april, 2003)* (2003) (cit. on p. 29).

[Duf06] I. S. Duff. *Sparse system solution and the HSL library.* In: *Some topics in industrial and applied mathematics* 8 (2006), pp. 78–94 (cit. on p. 30).

[Edd10] D. Eddelbuettel. *Benchmarking single-and multi-core BLAS implementations and GPUs for use with R.* 00003. Mathematica, 2010 (cit. on p. 17).

[Fav09] M. Faverge. "Ordonnancement hybride statique-dynamique en algèbre linéaire creuse pour de grands clusters de machines NUMA et multi-cœurs". PhD thesis. Université Sciences et Technologies - Bordeaux I, Dec. 7, 2009 (cit. on pp. 55, 92, 123).

[FLR98] M. Frigo, C. E. Leiserson, and K. H. Randall. *The Implementation of the Cilk-5 Multithreaded Language.* In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* Montreal, Canada, June 1998 (cit. on p. 13).

[FR08] M. Faverge and P. Ramet. *Dynamic Scheduling for sparse direct Solver on NUMA architectures.* Anglais. In: *PARA'08.* LNCS. Norway, 2008 (cit. on p. 30).

[FR95] L. Facq and J. Roman. *Algèbre linéaire creuse : distribution par bloc pour une factorisation parallèle de Cholesky.* In: *Parallélisme et applications irrégulières.* Ed. by G. Authié, J. M. Garcia, A. Ferreira, J. L. Roch, G. Villard, J. Roman, C. Roucairol, and B. V. editors. Hermès, 1995, pp. 135–147 (cit. on p. 40).

[Fra+05] V. Frayssé, L. Giraud, S. Gratton, and J. Langou. *Algorithm 842: A set of GMRES routines for real and complex arithmetics on high performance computers.* In: *ACM Transactions on Mathematical Software (TOMS)* 31.2 (2005), pp. 228–238 (cit. on p. 107).

[Gab+04] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, et al. *Open MPI: Goals, concept, and design of a next generation MPI implementation.* In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface.* 00848. Springer, 2004, pp. 97–104 (cit. on p. 13).

[GDL07] L. Grigori, J. A. Demmel, and X. S. Li. *Parallel Symbolic Factorization for Sparse LU with Static Pivoting.* In: *SIAM J. Sci. Comput.* 29.3 (2007), pp. 1289–1314. ISSN: 1064-8275. DOI: `http://dx.doi.org/10.1137/050638102` (cit. on p. 30).

[Geo+11] T. George, V. Saxena, A. Gupta, A. Singh, and A. Choudhury. *Multifrontal Factorization of Sparse SPD Matrices on GPUs.* In: *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International.* Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International. 2011, pp. 372–383. DOI: `10.1109/IPDPS.2011.44` (cit. on p. 33).

[GG08] K. Goto and R. A. v. d. Geijn. *Anatomy of high-performance matrix multiplication.* In: *ACM Trans. Math. Softw.* 34.3 (2008), pp. 1–25. ISSN: 0098-3500. DOI: `http://doi.acm.org/10.1145/1356052.1356053` (cit. on p. 17).

[GH08] J. Gaidamour and P. Hénon. *A parallel direct/iterative solver based on a Schur complement approach.* In: *Computational Science and Engineering, 2008. CSE'08. 11th IEEE International Conference on.* IEEE, 2008, pp. 98–105 (cit. on p. 19).

[GH09] L. Giraud and A. Haidar. *Parallel algebraic hybrid solvers for large 3D convection-diffusion problems.* In: *Numerical Algorithms* 51.2 (2009), pp. 151–177 (cit. on pp. 19, 124).

[GKJ98] A. Gupta, V. Kumar, and M. Joshi. *WSSMP:A High-Performance Shared- and Distributed-Memory Parallel Sparse Symmetric Linear Equation Solver.* In: *PARA'98 Workshop on Applied Parallel Computing in Large Scale Scientific and Industrial Problems* (June 1998) (cit. on p. 28).

[GKK97a] A. Gupta, G. Karypis, and V. Kumar. *Highly scalable parallel algorithms for sparse matrix factorization.* In: IEEE Trans. on Parallel and Distributed Systems 8.5 (May 1997), pp. 502–520 (cit. on pp. 24, 40).

[GKK97b] A. Gupta, G. Karypis, and V. Kumar. *Highly scalable parallel algorithms for sparse matrix factorization.* In: *Parallel and Distributed Systems, IEEE Transactions on* 8.5 (1997), pp. 502–520 (cit. on pp. 24, 40).

[GL81] A. George and J. W.-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1981 (cit. on pp. 20, 22).

[GLS99] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface.* Vol. 1. MIT press, 1999 (cit. on p. 13).

[GN89] G. A. Geist and E. Ng. *Task Scheduling for Parallel Sparse Cholesky Factorization.* In: *Internat. J. Parallel Programming* 18.4 (1989), pp. 291–314 (cit. on p. 40).

[Gro02] W. Gropp. *MPICH2: A new start for MPI implementations.* In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface.* 00124. Springer, 2002, pp. 7–7 (cit. on p. 13).

[Gup01] A. Gupta. *Recent progress in general sparse direct solvers.* In: *LNCS.* Vol. 2073. 2001, pp. 823–840 (cit. on p. 29).

[Gup07] A. Gupta. *A Shared- and distributed-memory parallel general sparse direct solver.* In: *Appl. Algebra Eng., Commun. Comput.* 18.3 (2007), pp. 263–277. ISSN: 0938-1279. DOI: `http://dx.doi.org/10.1007/s00200-007-0037-x` (cit. on p. 30).

[Hac85] W. Hackbusch. *Multi-grid methods and applications.* Vol. 4. Springer-Verlag Berlin, 1985 (cit. on p. 19).

[HC07] G. T. A. Huysmans and O. Czarny. *MHD stability in X-point geometry: simulation of ELMs.* In: *Nuclear fusion* 47.7 (2007), p. 659 (cit. on pp. 103, 105).

[Hén01] P. Hénon. "Distribution des Données et Régulation Statique des Calculs et des Communications pour la Résolution de Grands Systèmes Linéaires Creux par Méthode Directe". PhD thesis. Talence, France: LaBRI, Université Bordeaux I, Talence, Nov. 2001 (cit. on p. 39).

[Her+10] E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard. *Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations.* In: *Euro-Par (2).* Ed. by P. D'Ambra, M. R. Guarracino, and D. Talia. Vol. 6272. Lecture Notes in Computer Science. Springer, 2010, pp. 235–246. ISBN: 978-3-642-15290-0 (cit. on p. 15).

[HNP91] M. T. Heath, E. Ng, and B. W. Peyton. *Parallel algorithms for sparse linear systems.* In: *SIAM Rev.* 33.3 (1991), pp. 420–460. ISSN: 0036-1445. DOI: `http://dx.doi.org/10.1137/1033099` (cit. on p. 28).

[Hog10] J. D. Hogg. "High performance Cholesky and symmetric indefinite factorizations with applications". PhD thesis. 2010 (cit. on p. 30).

[HOS14] J. Hogg, E. Ovtchinnikov, and J. A. Scott. *A Sparse symmetric indefinite direct solver for GPU architectures.* In: *ACM Transactions on Mathematical Software (TOMS)* (2014) (cit. on p. 32).

[HRR02] P. Hénon, P. Ramet, and J. Roman. *PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems.* In: *Parallel Computing* 28.2 (Jan. 2002), pp. 301–321 (cit. on p. 29).

[HRR08] P. Hénon, P. Ramet, and J. Roman. *On finding approximate supernodes for an efficient ILU(k) factorization.* In: *Parallel Computing* 34 (2008), pp. 345–362 (cit. on p. 50).

[HRS10] J. Hogg, J. Reid, and J. Scott. *Design of a Multicore Sparse Cholesky Factorization Using DAGs*. In: *SIAM Journal on Scientific Computing* 32.6 (Jan. 1, 2010), pp. 3627–3649. ISSN: 1064-8275. DOI: 10.1137/090757216 (cit. on p. 30).

[HS13] J. Hogg and J. Scott. *New Parallel Sparse Direct Solvers for Multicore Architectures*. In: *Algorithms* 6.4 (2013), pp. 702–725 (cit. on p. 30).

[HSÇ12] T. D. R. Hartley, E. Saule, and Ü. V. Çatalyürek. *Improving performance of adaptive component-based dataflow middleware*. In: *Parallel Computing* 38.6-7 (2012), pp. 289–309 (cit. on p. 15).

[Hül+06] F. Hülsemann, M. Kowarschik, M. Mohr, and U. Rüde. *Parallel geometric multigrid*. In: *Numerical Solution of Partial Differential Equations on Parallel Computers*. Springer, 2006, pp. 165–208 (cit. on p. 19).

[Igu+12] F. D. Igual, E. Chan, E. S. Quintana-Ortí, G. Quintana-Ortí, R. A. Van De Geijn, and F. G. Van Zee. *The FLAME approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations*. In: *Journal of Parallel and Distributed Computing* 72.9 (2012). 00015, pp. 1134–1143 (cit. on pp. 16, 32).

[Int] Intel. *Intel Math Kernel Library*. http://software.intel.com/en-us/intel-mkl (cit. on p. 17).

[Jos+99] M. Joshi, G. Karypis, V. Kumar, A. Gupta, and G. F. *PSPASES : Scalable Parallel Direct Solver Library for Sparse Symmetric Positive Definite Linear Systems*. Tech. rep. University of Minnesota and IBM Thomas J. Watson Research Center, May 1999 (cit. on p. 28).

[JR13] J. Jeffers and J. Reinders. *Intel Xeon Phi coprocessor high performance programming*. Newnes, 2013 (cit. on p. 10).

[KD09] J. Kurzak and J. Dongarra. *Fully Dynamic Scheduler for Numerical Computing on Multicore Processors*. In: *LAPACK working note* lawn220 (2009) (cit. on p. 15).

[KK11] D. M. Kunzman and L. V. Kalé. *Programming heterogeneous clusters with accelerators using object-based programming*. In: *Scientific Programming* 19.1 (2011), pp. 47–62 (cit. on p. 15).

[KK93] L. V. Kale and S. Krishnan. *CHARM++: A Portable Concurrent Object Oriented System Based on C++*. In: *SIGPLAN Not.* 28.10 (Oct. 1993), pp. 91–108. ISSN: 0362-1340. DOI: 10.1145/167962.165874 (cit. on p. 15).

[KLV98] B. Kågström, P. Ling, and C. Van Loan. *GEMM-based level 3 BLAS: High-performance model implementations and performance evaluation benchmark*. In: *ACM Transactions on Mathematical Software (TOMS)* 24.3 (1998). 00000, pp. 268–302 (cit. on p. 17).

[KP10] G. P. Krawezik and G. Poole. *Accelerating the ANSYS direct sparse solver with GPUs*. In: *2010 Symposium on Application Accelerators in High Performance Computing (SAAHPC'10)*. 2010 (cit. on p. 33).

[KR12] S. J. Krieder and I. Raicu. *An overview of current and future computing accelerator architectures*. In: *1st Greater Chicago Area System Research Workshop Poster Session*. 2012 (cit. on p. 8).

[KTD12] J. Kurzak, S. Tomov, and J. Dongarra. *Autotuning GEMM Kernels for the Fermi GPU*. In: *IEEE Transactions on Parallel and Distributed Systems* 23.11 (2012), pp. 2045–2057. ISSN: 1045-9219 (cit. on pp. 60, 61).

[Kyu14] V. E. Kyungjoo Kim. *A Parallel Sparse Direct Solver via Hierarchical DAG Scheduling*. In: *{ACM} Transactions on Mathematical Software* 41.1 (Jan. 28, 2014) (cit. on p. 34).

[Law+79] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. *Basic Linear Algebra Subprograms for Fortran Usage*. In: *ACM Trans. Math. Softw.* 5.3 (1979), pp. 308–323. ISSN: 0098-3500. DOI: `http://doi.acm.org/10.1145/355841.355847` (cit. on p. 17).

[LD03] X. S. Li and J. W. Demmel. *SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems*. In: *ACM Trans. Mathematical Software* 29.2 (June 2003), pp. 110–140 (cit. on p. 29).

[LD98] X. S. Li and J. W. Demmel. *Making sparse Gaussian elimination scalable by static pivoting*. In: *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 1998, pp. 1–17 (cit. on p. 28).

[LHK09] C.-K. Luk, S. Hong, and H. Kim. *Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping*. In: *MICRO*. Ed. by D. H. Albonesi, M. Martonosi, D. I. August, and J. F. Martínez. ACM, 2009, pp. 45–55. ISBN: 978-1-60558-798-1 (cit. on p. 15).

[Li08] X. S. Li. *Evaluation of Sparse LU Factorization and Triangular Solution on Multicore Platforms*. In: *VECPAR*. Ed. by J. M. L. M. Palma, P. Amestoy, M. J. Daydé, M. Mattoso, and J. C. Lopes. Vol. 5336. Lecture Notes in Computer Science. Springer, 2008, pp. 287–300. ISBN: 978-3-540-92858-4 (cit. on p. 30).

[Liu90] J. W.-H. Liu. *The role of elimination trees in sparse factorization*. In: *SIAM J. Matrix Anal. Appl.* 11 (1990), pp. 134–172 (cit. on p. 21).

[LS13] J.-Y. L'Excellent and M. W. Sid-Lakhdar. *Introduction of shared-memory parallelism in a distributed-memory multifrontal solver*. Anglais. Rapport de recherche RR-8227. INRIA, Feb. 2013, p. 35 (cit. on p. 29).

[Luc+11] R. F. Lucas, G. Wagenbreth, D. M. Davis, and R. Grimes. *Multifrontal computations on GPUs and their multi-core hosts*. In: *High Performance Computing for Computational Science–VECPAR 2010*. Springer, 2011, pp. 71–82 (cit. on p. 33).

[Man80] T. A. Manteuffel. *An incomplete factorization technique for positive definite linear systems.* In: *Mathematics of Computation* 34.150 (1980), pp. 473–497. ISSN: 0025-5718, 1088-6842. DOI: 10.1090/S0025-5718-1980-0559197-0 (cit. on p. 19).

[Mbe+13] D. Mbengoue, D. Genet, C. Lachat, E. Martin, M. Mogé, V. Perrier, F. Renac, M. Ricchiuto, and F. Rue. *Comparison of high order algorithms in Aerosol and Aghora for compressible flows.* In: *ESAIM: Proceedings* 43 (Dec. 2013), pp. 1–16 (cit. on p. 15).

[Meu+14] H. Meuer, E. Strohmaier, J. Dongarra, and H. D. Simon. *Top 500 Supercomputer sites.* Nov. 2014. URL: http://www.top500.org/lists/2014/11/ (cit. on pp. 8, 123).

[Moo+65] G. E. Moore et al. *Cramming more components onto integrated circuits.* 1965 (cit. on p. 7).

[NP93] E. Ng and B. W. Peyton. *Block sparse Cholesky algorithms on advanced uniprocessor computers.* In: *SIAM J. Sci. Comput.* 14 (1993), pp. 1034–1056 (cit. on pp. 27, 40).

[NR98] R. W. Numrich and J. Reid. *Co-Array Fortran for parallel programming.* In: *ACM Sigplan Fortran Forum.* Vol. 17. 00544. ACM, 1998, pp. 1–31 (cit. on p. 13).

[NTD10] R. Nath, S. Tomov, and J. Dongarra. *An improved MAGMA GEMM for Fermi graphics processing units.* In: *International Journal of High Performance Computing Applications* 24.4 (2010), pp. 511–515 (cit. on p. 60).

[NTD11] R. Nath, S. Tomov, and J. Dongarra. *Accelerating GPU Kernels for Dense Linear Algebra.* In: *High Performance Computing for Computational Science – VECPAR 2010.* Ed. by J. M. L. M. Palma, M. Daydé, O. Marques, and J. C. Lopes. Lecture Notes in Computer Science 6449. Springer Berlin Heidelberg, Jan. 1, 2011, pp. 83–92. ISBN: 978-3-642-19327-9, 978-3-642-19328-6 (cit. on p. 31).

[NVI08] C. NVIDIA Inc. *Cublas library.* In: *NVIDIA Corporation, Santa Clara, California* 15 (2008) (cit. on pp. 31, 60).

[NVI11] I. NVIDIA. *Nvidia CUDA C programming guide.* In: *NVIDIA Corporation* 120 (2011) (cit. on p. 14).

[Owe+08] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. *GPU computing.* In: *Proceedings of the IEEE* 96.5 (2008), pp. 879–899 (cit. on p. 9).

[Pla+09] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. *Hierarchical task-based programming with StarSs.* In: *International Journal of High Performance Computing Applications* 23.3 (2009). 00111, pp. 284–299 (cit. on pp. 2, 15).

[PR97] F. Pellegrini and J. Roman. *Sparse matrix ordering with* SCOTCH. In: *Proceedings of HPCN'97*. Lecture Notes in Computer Science 1225. Springer Verlag, Apr. 1997, pp. 370–378 (cit. on p. 20).

[PRA99] F. Pellegrini, J. Roman, and P. Amestoy. *Hybridizing Nested Dissection and Halo Approximate Minimum Degree for Efficient Sparse Matrix Ordering*. In: *Proceedings of Irregular'99*. Lecture Notes in Computer Science 1586. Extended paper appeared in Concurrency: Practice and Experience, 12:69-84, 2000. Springer Verlag, Apr. 1999, pp. 986–995 (cit. on p. 20).

[Ram00] P. Ramet. *Optimisation de la communication et de la distribution des données pour des solveurs parallèles directs en algèbre linéaire dense et creuse*. Bordeaux 1, Jan. 1, 2000 (cit. on pp. 30, 39).

[RBH12] S. Rajamanickam, E. G. Boman, and M. A. Heroux. *ShyLU: A hybrid-hybrid solver for multicore platforms*. In: *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012, pp. 631–643 (cit. on pp. 19, 124).

[Rei07] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007 (cit. on p. 13).

[RG94] E. Rothberg and A. Gupta. *An Efficient Block-oriented Approach to Parallel Sparse Cholesky Factorization*. In: *SIAM J. Sci. Comput.* 15.6 (Nov. 1994), pp. 1413–1439 (cit. on p. 40).

[Ric96] H. Richardson. *High performance fortran: history, overview and current developments*. In: *Thinking Machines Corporation* 14 (1996) (cit. on p. 13).

[Rom94] J. Roman. *Partitionnement algorithmique des données pour la factorisation de Cholesky par bloc de grands systèmes linéaires creux sur des calculateurs MIMD*. In: *Lettre du transputer et des calculateurs parallèles* 6(24) (1994), pp. 115–120 (cit. on p. 40).

[Rot96] E. Rothberg. *Performance of Panel and Block Approaches to Sparse Cholesky Factorization on the iPSC/860 and Paragon Multicomputers*. In: *SIAM J. Sci. Comput.* 17(3) (May 1996), pp. 699–713 (cit. on p. 40).

[RS94] E. Rothberg and R. Schreiber. *Improved Load Distribution in Parallel Sparse Cholesky Factorization*. In: *Proceedings of Supercomputing'94*. IEEE. 1994, pp. 783–792 (cit. on p. 40).

[RSD14] S. C. Rennich, D. Stosic, and T. A. Davis. *Accelerating sparse cholesky factorization on GPUs*. In: *Proceedings of the Fourth Workshop on Irregular Applications: Architectures and Algorithms*. 00000. IEEE Press, 2014, pp. 9–16 (cit. on p. 33).

[RTL76] D. J. Rose, R. E. Tarjan, and G. S. Lueker. *Algorithmic aspects of vertex elimination on graphs*. In: *SIAM Journal on computing* 5.2 (1976), pp. 266–283 (cit. on p. 21).

[Saa94] Y. Saad. *ILUT: A dual threshold incomplete LU factorization.* In: *Numerical linear algebra with applications* 1.4 (1994), pp. 387–402 (cit. on p. 19).

[Saa96] Y. Saad. *Iterative Methods For Sparse Linear Systems.* Ed. PWS publishing Compagny, 1996 (cit. on p. 18).

[SCB08] O. Schenk, M. Christen, and H. Burkhart. *Algorithmic performance studies on graphics processing units.* In: *Journal of Parallel and Distributed Computing* 68.10 (2008), pp. 1360–1369 (cit. on p. 32).

[Sch93] R. Schreiber. *Scalability of sparse direct solvers.* Springer, 1993 (cit. on p. 40).

[SG04] O. Schenk and K. Gärtner. *Solving unsymmetric sparse systems of linear equations with PARDISO.* In: *Future Gener. Comput. Syst.* 20.3 (2004), pp. 475–487. ISSN: 0167-739X. DOI: `http://dx.doi.org/10.1016/j.future.2003.07.011` (cit. on p. 30).

[SGS10] J. E. Stone, D. Gohara, and G. Shi. *OpenCL: A parallel programming standard for heterogeneous computing systems.* In: *Computing in science & engineering* 12.3 (2010), p. 66 (cit. on p. 14).

[Sut05] H. Sutter. *The free lunch is over: A fundamental turn toward concurrency in software.* In: *Dr. Dobb's journal* 30.3 (2005), pp. 202–210 (cit. on p. 2).

[SVL14] P. Sao, R. Vuduc, and X. S. Li. *A Distributed CPU-GPU Sparse Direct Solver.* In: *Euro-Par 2014 Parallel Processing.* Ed. by F. Silva, I. Dutra, and V. S. Costa. Lecture Notes in Computer Science 8632. Springer International Publishing, 2014, pp. 487–498. ISBN: 978-3-319-09872-2, 978-3-319-09873-9 (cit. on p. 33).

[SYD09] 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 (cit. on p. 15).

[Tan+11] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, and N. Sun. *Fast implementation of DGEMM on Fermi GPU.* In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis.* SC '11. Seattle, Washington: ACM, 2011, 35:1–35:11. ISBN: 978-1-4503-0771-0 (cit. on p. 60).

[TCR03] S. Toledo, D. Chen, and V. Rotkin. *Taucs: A library of sparse linear solvers.* 2003 (cit. on p. 30).

[The05] I. The MathWorks. *MATLAB: the language of technical computing. Desktop tools and development environment, version 7.* Vol. 9. 00081. MathWorks, 2005 (cit. on p. 29).

[Tra+11] F. Trahay, Y. Ishikawa, F. Rue, R. Namyst, M. Faverge, and J. Dongarra. *EZTrace: a generic framework for performance analysis.* In: *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on.* IEEE, 2011, pp. 618–619 (cit. on p. 69).

[TW67] W. F. Tinney and J. W. Walker. *Direct solutions of sparse network equations by optimally ordered triangular factorization.* In: *J. Proc. IEEE* 55 (1967), pp. 1801–1809 (cit. on p. 22).

[VD08] V. Volkov and J. W. Demmel. *Benchmarking GPUs to tune dense linear algebra.* In: *Proceedings of the 2008 ACM/IEEE conference on Supercomputing.* SC '08. Austin, Texas: IEEE Press, 2008, 31:1–31:11. ISBN: 978-1-4244-2835-9 (cit. on pp. 50, 60).

[Vud+10] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney, and A. Shringarpure. *On the limits of GPU acceleration.* In: *Proceedings of the 2nd USENIX conference on Hot topics in parallelism.* USENIX Association, 2010, pp. 13–13 (cit. on p. 32).

[WD97] R. C. Whaley and J. Dongarra. *Automatically Tuned Linear Algebra Software.* Tech. rep. 131. LAPACK Working Note, Dec. 1997 (cit. on p. 17).

[WPD00] R. C. Whaley, A. Petitet, and J. Dongarra. *Automated Empirical Optimization of Software and the ATLAS Project.* Tech. rep. 147. LAPACK Working Note, Sept. 2000 (cit. on p. 17).

[Yam12] I. Yamazaki. *PDSLin User Guide.* In: (2012) (cit. on p. 19).

[Yar12] A. YarKhan. "Dynamic Task Execution on Shared and Distributed Memory Architectures". PhD thesis. Innovative Computing Laboratory, University of Tennessee, Dec. 2012 (cit. on p. 14).

[YDR13] S. N. Yeralan, T. A. Davis, and S. Ranka. *Sparse QR Factorization on GPU Architectures.* In: *Technical Report, University of Florida* (Nov. 2013) (cit. on p. 32).

[YL12] I. Yamazaki and X. S. Li. *New scheduling strategies and hybrid programming for a parallel right-looking sparse lu factorization algorithm on multicore cluster systems.* In: *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International.* IEEE, 2012, pp. 619–630 (cit. on p. 30).

[YTD12] I. Yamazaki, S. Tomov, and J. Dongarra. *One-sided Dense Matrix Factorizations on a Multicore with Multiple GPU Accelerators.* In: *Procedia Computer Science* 9.Complete (2012), pp. 37–46 (cit. on p. 59).

[YWP11] C. D. Yu, W. Wang, and D. Pierce. *A CPU–GPU hybrid approach for the unsymmetric multifrontal method.* In: *Parallel Computing.* 6th International Workshop on Parallel Matrix Algorithms and Applications (PMAA'10) 37.12 (2011), pp. 759–770. ISSN: 0167-8191. DOI: `10.1016/j.parco.2011.09.002` (cit. on p. 33).

Appendix A

Publications

Publication in conference with proceedings

[Lac+14] X. Lacoste, M. Faverge, G. Bosilca, P. Ramet, and S. Thibault. *Taking Advantage of Hybrid Systems for Sparse Direct Solvers via Task-Based Runtimes*. In: *Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. IPDPSW '14. Phoenix, USA: IEEE Computer Society, 2014, pp. 29–38. ISBN: 978-1-4799-4116-2. DOI: 10.1109/IPDPSW.2014.9.

Publications in conference with selection committee

[Bos+12] G. Bosilca, M. Faverge, X. Lacoste, I. Yamazaki, and P. Ramet. *Toward a supernodal sparse direct solver over DAG runtimes*. In: *Proceedings of PMAA'2012*. Londres, UK, June 2012.

[FLR08] M. Faverge, X. Lacoste, and P. Ramet. *A NUMA Aware Scheduler for a Parallel Sparse Direct Solver*. In: *Proceedings of PMAA'2008*. Neuchatel, Swiss, June 2008.

[LFR13] X. Lacoste, M. Faverge, and P. Ramet. *Sparse Linear Algebra over DAG Runtimes*. In: *SIAM Conference on Computation Science and Engineering*. Boston, USA, Feb. 2013.

[LFR15] X. Lacoste, M. Faverge, and P. Ramet. *A task-based sparse direct solver suited for large scale hierarchical/heterogeneous architectures*. In: *Mini-Symposium on "Task-based Scientific Computing Applications" at SIAM CSE'15 conference*. Salt Lake City, USA, Mar. 2015.

[LR12] X. Lacoste and P. Ramet. *Sparse direct solver on top of large-scale multicore systems with GPU accelerators*. In: *SIAM Conference on Applied Linear Algebra*. Valence, Spain, June 2012.

Other conferences

[Agu+12] E. Agullo, G. Bosilca, B. Bramas, C. Castagnede, O. Coulaud, E. Darve, J. Dongarra, M. Faverge, N. Furmento, L. Giraud, X. Lacoste, J. Langou, H. Ltaief, M. Messner, R. Namyst, P. Ramet, T. Takahashi, S. Thibault, S. Tomov, and I. Yamazaki. *Matrices over Runtime Systems at Exascale.* Published: SuperComputing'2012, Salt Lake City, USA. Nov. 2012. 1332-1332.

[Lac13] X. Lacoste. *Work stealing and granularity optimizations for a sparse solver on manycores.* In: *Sparse Days.* Toulouse, France, June 2013.

[LFR12a] X. Lacoste, M. Faverge, and P. Ramet. *Scheduling for Sparse Solver on Manycore Architectures.* Published: Workshop INRIA-CNPq, HOSCAR meeting, Petropolis, Brazil. Sept. 2012.

[LFR12b] X. Lacoste, M. Faverge, and P. Ramet. *Sparse direct solvers with accelerators over DAG runtimes.* Published: Workshop INRIA-CNPq, HOSCAR meeting, Sophia-Antipolis, France. July 2012.

[LFR13] X. Lacoste, M. Faverge, and P. Ramet. *Sparse Linear Algebra over DAG Runtimes.* Published: SOLHAR meeting, Bordeaux, France. Nov. 2013.

Technical reports

[FLR10] M. Faverge, X. Lacoste, and P. Ramet. *A NUMA Aware Scheduler for a Parallel Sparse Direct Solver.* 2010.

[Lac+12] X. Lacoste, P. Ramet, M. Faverge, Y. Ichitaro, and J. Dongarra. *Sparse direct solvers with accelerators over DAG runtimes.* Research Report RR-7972. INRIA, 2012, p. 11.

[Lac+14] X. Lacoste, M. Faverge, P. Ramet, S. Thibault, and G. Bosilca. *Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes.* Research Report RR-8446. INRIA, Jan. 2014, p. 25.

Appendix B

Integration in Algo'Tech software

During my PhD, I interacted with Algo'Tech, a SME from Bidart in the south west of France. This company is developing a CAD software to design electric diagram. The team also developed an electric solver to validate the diagrams and is extending this solution to electromagnetic problems. Including electromagnetic into their simulation tool increases the problem size considerably. Thus, they decided to move from their homemade direct solver to an HPC solution via the HPC-PME[15] program. In this context, we worked together to find a high performance solution to their problems.

B.1 Algo'Tech software simulation tool

Algo'Tech software company developed a simulation tool to validate the electric diagrams that have been designed via their CAD tool. Their simulation program performs a direct factorization for each electric frequency in a given range and with a given step. Among the different systems to solve, the pattern of the matrix remains constant and the values of the matrix are numerically close. The solving of the linear systems are also totally independent of each other.

The main drawback of the initial *LU* solver implemented in Algo'Tech's simulation was its lack of symbolic factorization. The structure of the factorized matrix was discovered during the factorization, and the insertion of new value into the structure during factorization had a prohibitive cost. Some investigation had been done on reordering, giving a decent improvement. Compared to our solution the solver did not propose parallelism, but this is not a real requirement as the frequency loop is highly parallel. PaStiX solver also benefits from BLAS 3 operations that were not available in the homemade solver.

[15] `http://www.initiative-hpc-pme.org/`

B.2 Optimizations

The first step of our work with Algo'Tech was to replace their home made direct solver with our high performance library. Then, we could investigate the parallel execution of these independent linear systems solving in a multi-threaded context. The third optimization investigated was the offload of the linear systems solving loop on a high performance platform. An other possibility that was proposed but not yet developed was a numerical optimization of the solving loop.

B.2.1 PaStiX integration

The first logical step for us was to integrate our highly optimized library into their code in Delphi. To achieve this goal we had to finalize the windows port of PaStiX library and to develop a Delphi interface to PaStiX. As the pattern of the matrix is constant, using PaStiX, we could perform the preprocessing step only once for the whole frequencies loop.

B.2.2 Parallelization

In the simulation tool, we could investigate the possibility of parallelization using multi-threading. Three solutions were possible:

- use multi-threaded BLAS;
- use threads in PaStiX;
- use threads in the frequencies loop.

As the blocks built by PaStiX are very small, using thread in the BLAS library is worthless and we'd rather use threads inside PaStiX. We can obtain more parallelism using threads in the frequencies loop because frequencies loop iterations are totally independent. Thus, we deactivated the multi-threading both in the BLAS library and in PaStiX library and use a parallel for model on the frequencies loop (Alg. 11).

B.2.3 Cloud computing

The second approach we investigated was the offload of the whole frequencies loop on a high performance cluster. This was done on the Bull ExtremeFactory machine. We send the three matrices A B and C via SSH and then a program executing the factorization loop is executed with a given range of values for ω. The resulting solution vectors are then copied back and analyzed by the simulation software. To reduce the cost of data transfer, the result vectors are packed into an archive before the transfer is performed.

With this approach, we can obtain a highly parallel solution but in our investigation the problems were still too small to benefit from the power of the machine. Moreover, the ratio of data transfers compared to computation was too high. We think that we should offload more computations, such as the matrix building from the electric

Algorithm 11 Frequencies loop using a classical direct solver.

▷ Build pattern of the matrix using any non nil value of ω
1: $S \rightarrow A + i\omega_1 B + \frac{iC}{\omega_1}$
2: $Preprocess(S)$
▷ Can be executed in parallel
3: **For** each ω in $[\![\omega_{min}; \omega_{max}]\!]$ **Do**
4: $S \rightarrow A + i\omega B + i\omega C$
5: $LU \rightarrow Fact(S)$
6: $x \rightarrow Solve(LU, b_\omega)$
▷ Post process the solution to plot results
7: $Postprocess(x)$
8: **End For**

diagram, to reach better reactivity. However, with more complex diagrams this solution can accelerate the computation.

B.2.4 Numerical optimization

Another possible optimization we proposed but did not investigate yet was a numerical optimization. Indeed, we have seen that the different solved problems are close to each other numerically. Thus, one could perform the factorization for a given frequency and reuse it as a preconditioner to solve iteratively next frequencies' problems (Alg. 12). The factorization would be performed only when the iterative solver does not converge quick enough. This would reduce the complexity of this algorithm as the complexity of an iterative step of the solver is only the complexity of a matrix vector product ($\mathrm{O}(nnz)$) compared to the complexity of a matrix decomposition ($\mathrm{O}(n^2)$).

B.3 Conclusion

In this section we have shown that PaStiX library is not only restricted to the research community but our experience can also address smaller problems efficiently and be profitable to SMEs.
As we have seen in this section, some investigations still need to be done in the solving method and improvements could still be obtained by offloading a larger part of the simulation to the cluster, increasing the computation/communication ratio and thus the scalability.

Algorithm 12 Frequencies loop with direct preconditionned iterative solver.

▷ Build pattern of the matrix using any non nil value of ω
$S \rightarrow A + i\omega_1 B + \frac{iC}{\omega_1}$
$Preprocess(S)$
▷ Can be executed in parallel
For each ω in $[\![\omega_{min}; \omega_{max}]\!]$ **Do**
  $S \rightarrow A + i\omega B + \frac{iC}{\omega}$
  $LU \rightarrow Fact(S)$
  $niter \rightarrow 0$
  While $niter < maxIter$ **Do**
    $S' \rightarrow A + i\omega B + \frac{iC}{\omega}$
    $(x, niter) \rightarrow IterativeSolve(LU, S', b_\omega)$
  End While
▷ Post process the solution to plot results
  $Postprocess(x)$
End For

Appendix C

Murge and Jorek code samples

In this annex, we present algorithms that illustrate the chapter 4. First, we detail a MURGE API classical use case. Then, we present the original matrix assembly algorithm in JOREK and, finally, the new version using MURGE.

Listing 5 presents the different steps one has to follow to call a sparse linear solver using MURGE API:

1. Initialization: allocates the required data structures for the linear algebra library and set the solver parameters (lines 1 to 10);
2. Graph setting: assembles the graph of the matrix (ie. the matrix without the values). Once it has been performed, the linear algebra solver preprocessing steps (ordering, symbolic factorization, and data distribution in the case of PaStiX) can be triggered (lines 13 to 20);
3. Remapping: retrieves the matrix distribution adapted to the solver and compute a new corresponding mesh distribution (lines 23 to 27);
4. Matrix and right-hand-side setting: fills the matrix and right-hand-side with coefficients to perform the solving steps (lines 30 to 44);
5. Getting the solution: queries the solution vector (line 46);
6. Clean: deallocates all internal data structures (lines 49 and 50)).

Listing 5 MURGE simple test case.

```
1   MURGE_Initialize(1); /* Initialize for 1 instance */
2   id = 0;              /* id of the linear system */
3
4   /* Initialize Default solver options   */
5   MURGE_SetDefaultOptions(id, 0);
6
7   /* Set options */
```

```
8   MURGE_SetCommunicator(id, MPI_COMM_WORLD);
9   MURGE_SetOptionINT(id, MURGE_IPARAM_BASEVAL, 1);
10  MURGE_SetOptionINT(id, MURGE_IPARAM_SYM, sym);
11
12  /* Set the graph */
13  MURGE_GraphBegin(id, localn, lnnz);
14  for (i = 0; i < nLocalElements; i++) {
15    e = localElements[i];
16    /* get the list of vertices of e in idx array */
17    GetVertices(e, idx);
18    MURGE_GraphBlockEdge(id, idx, idx);
19  }
20  MURGE_GraphEnd(id);
21
22  /* Get solver distribution */
23  MURGE_GetLocalUnknowsNumber(id, &n);
24  localUnknowns=malloc(n*sizeof(INTS));
25  MURGE_GetLocalUnknowsList(id, localUnknowns);
26  reBuildLocalElements(n, localUnknowns,
27                       &localElements, &lnnz);
28
29  /* Assemble the matrix and right-hand-side */
30  MURGE_AssemblyBegin(id, n, lnnz,
31                      MURGE_ASSEMBLY_OVW,
32                      MURGE_ASSEMBLY_OVW,
33                      MURGE_ASSEMBLY_FOOL, sym);
34  for (i = 0; i < nLocalElements; i++) {
35    e = localElement[i];
36    GetVertices(e, idx);
37    m=GetMatrix(e);
38    MURGE_AssemblySetBlockValues(is, idx, idx, m);
39    b=GetRhs(e);
40    for (j 0; j < nidx; j++)
41      rhs[idx[j]] = b[j];
42  }
43  MURGE_AssemblyEnd(id);
44  MURGE_SetGlobalRHS(id, rhs, 0, MURGE_ASSEMBLY_OVW);
45  /* Get the solution */
46  MURGE_GetGlobalSolution(id, xx));
47
48  /* Free Solver internal structures for problem id  */
49  MURGE_CALL(MURGE_Clean(id));
50  MURGE_CALL(MURGE_Finalize());
```

Algorithm 13 presents the matrix assembly in JOREK before the introduction of MURGE API. The distributed matrix corresponding to the global problem is simply assembled using the local matrix assembly. Then, the boundary conditions can be inserted. After the global matrix is complete, the harmonic matrices are gathered on each sub-communicator and transformed so that the direct solver can use it.

Algorithm 14 and Algorithm 15 present the matrix assembly in JOREK using MURGE API. Here, the global and harmonic matrices are both built in a distributed way. After the assembly loop, entries corresponding to non local harmonics are sent and conversely entries are received from other harmonics. After that, the direct solver is called in a distributed way, without conversion of the matrix.

Algorithm 13 Assembly algorithm using original PaSTiX interface.

```
1:  For each local element e Do
2:      EltMatrix ⇐ buildElementMatrix(e)
3:      For each vertex_col ∈ [[1, vertexNbr]] Do
4:          For order_col ∈ [[1, orderNbr + 1]] Do
5:              index_col ⇐ getIndex(vertex_col, order_col)
6:              If index_col is a local column Then
7:                  For each vertex_row ∈ [[1, vertexNbr]] Do
8:                      For each order_row ∈ [[1..orderNbr + 1]] Do
9:                          index_row ⇐ getIndex(vertex_row, order_row)
10:                         For each dof_row ∈ [[1, n_tor * n_var]] Do
11:                             For each dof_col ∈ [[1, n_tor * n_var]] Do
12:                                 value ⇐ getValue(EltMatrix, order_row, dof_row,
13:                                                  order_col, dof_col)
14:                                 GlobalDistributedMatrix.Insert(index_row, dof_row,
15:                                                                index_col, dof_col,
16:                                                                value)
17:                             End For
18:                         End For
19:                     End For
20:                 End For
21:             End If
22:         End For
23:     End For
24: End For
    ▷ Set boundary conditions
25: BoundaryCounditions(Distributed)
    ▷ Redistribute values accross harmonic communicator
26: For each (row, col, value) in DistributedMatrix Do
27:     tor_row ⇐ getTor(row)
28:     tor_col ⇐ getTor(col)
    ▷ If we are on a diagonal block it correspond to an harmonic
29:     If tor_row = tor_col Then
30:         DistHarmonicMatrix[tor_col].Insert(row, col, value)
31:     End If
32: End For
33: HarmonicMatrix ⇐ MPIGatherMatrix(DistHarmonicMatrix)
```

Algorithm 14 Assembly algorithm using MURGE API (1/2).

```
▷ Initiate assembly phase
MURGE_AssemblyBegin(murge_id_prod, n, nnz)
MURGE_AssemblyBegin(murge_id_harm, n, nnz)
For each local element e Do
    EltMatrix ⇐ buildElementMatrix(e)
    For each vertex_col ∈ [[1, vertexNbr]] Do
        For order_col ∈ [[1, orderNbr + 1]] Do
            index_col ⇐ getIndex(vertex_col, order_col)
            If index_col is a local column Then
                For each vertex_row ∈ [[1, vertexNbr]] Do
                    For each order_row ∈ [[1..orderNbr + 1]] Do
                        index_row ⇐ getIndex(vertex_row, order_row)
▷ Register the global problem elementary matrix
                        value ⇐ getValue(EltMatrix, order_row, dof_row, order_col, dof_col)
                        MURGE_AssemblySetNodeValues(
                            murge_id_prod, index_row, index_col, EltMatrix)
                        For each harmonic h Do
                            EltMatrixHarm ⇐
                                getHarmonicElementaryMatrix(EltMatrix, index_row,
            index_col, h)
                            If h == myHarm Then
▷ Register the local harmonic elementary matrix
                                MURGE_AssemblySetNodeValues(
                                    murge_id_harm, index_row, index_col, EltMatrixHarm)
                            Else
▷ Prepare to send data to other harmonics
                                ToSend[h].append(index_row, index_col, EltMatrixHarm)
                            End If
                        End For
                    End For
                End For
            End If
        End For
    End For
End For
▷ Communicate constructed matrices accross harmonics ... (see Algorithm 15)
```

Algorithm 15 Assembly algorithm using MURGE API (2/2).

```
    ▷ Finite element assembly loop ... (see Algorithm 14)
    ▷ Communicate constructed matrices accross harmonics
31: For each harmonic h Do
    ▷ Send data to harmonic h
32:     Send(ToSend[h], h, MPI_COMM_ACCROSS_HARM)
    ▷ Receive data from harmonic h
33:     Recv(ToAdd, h, MPI_COMM_ACCROSS_HARM)
34:     For i ∈ [[1, Size(ToAdd)]] Do
    ▷ Register the local harmonic elementary matrix
35:         MURGE_AssemblySetNodeValues(
36:             murge_id_harm, ToAdd[i].row, ToAdd[i].col, ToAdd[i].values)
37:     End For
38: End For
    ▷ End of the assembly step
39: MURGE_AssemblyEnd(murge_id_prod)
40: MURGE_AssemblyEnd(murge_id_harm)
```

Appendix D

Sparse matrix storage formats

In this annex, we describe the different format involved in our sparse solver. First the Compressed Sparse Column (CSC) format is the one used as an input in the centralized PaStiX library API. Then, an extension to distributed matrices is presented. Finally the dynamic CSC is used in MURGE API to obtain a more efficient structure for the assembly. Figure D.1 presents the CSC sparse matrix format:

n**:** number of column in the matrix;

colptr : starting index of each column in *rows* and *values* (column i goes from $rows[colptr[i] - colptr[0]]$ to $rows[colptr[i+1] - colptr[0]]$);

rows row indexes, sorted by column;

values corresponding values, sorted by column.

Our implementation of MURGE API aims at building a distributed Compressed Sparse Column matrix (called CSCD, see Figure D.2) and uses this one to call PaStiX. This approach is particular to solvers using a CSCD as input, but the API is generic and could also be implemented to build any kind of matrix.

Figure D.1 – An example of CSC matrix.

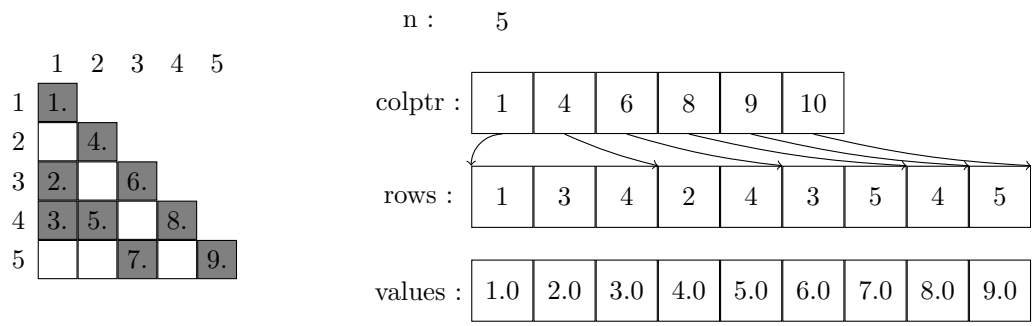
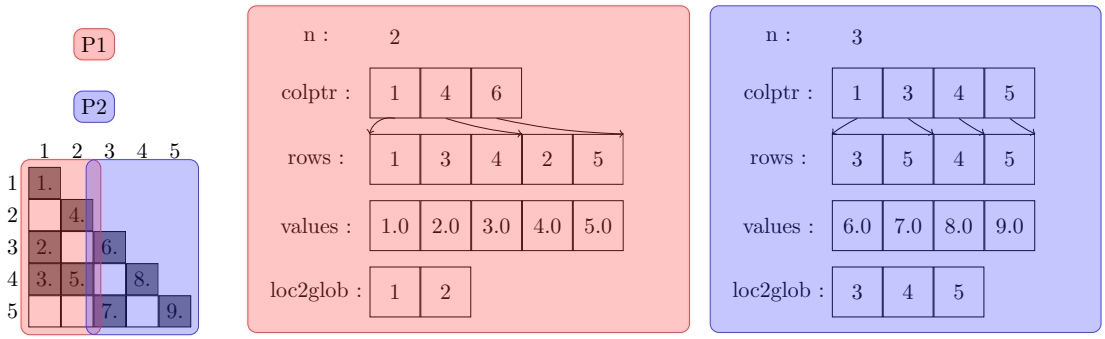


Figure D.2 – An example of CSCD matrix. The matrix is distributed on two processes, represented in blue and red.



The CSCD format is an extension of the CSC format in which local elements are stored in a CSC format. A `loc2glob` array extends the information to match the local columns into the global problem (local column i has a global number $loc2glob[i]$). To fit PaStiX needs, a column can only belongs to one MPI process. However, access in a CSCD structure is quite fast as we just have to pass through the non-zeros of one column to read or overwrite one entry (O(nnz/n) comparisons). On the other hand, inserting a new entry in this structure is not very efficient as all entries after the new one as to be moved. Indeed in a CSC matrix, all the rows (and also the values) are stored in the same array, one column after the other. In the worst case, when an entry is inserted in the first column, the whole rows and values entries have to move (O(nnz)). For an efficient assembly in MURGE, we need to use a more dynamic structure to store our matrix.

For more efficiency during the matrix assembly, we introduced a new dynamic CSC (listing 6) data structure.

This structure contains:

- the number of columns in the matrix (n),
- the number of expected non zeros in the matrix (nz),
- the number of degrees of freedom per unknown (dof),
- the number of entries in each column ($colsizes$),
- and one vector per column to store rows and values of the matrix.

This way, adding an entry in one column only alter the corresponding column of the matrix.

Once the assembly of the matrix is finished, we can convert it easily to a CSCD matrix that is used with the PaStiX original distributed interface. Building such a distributed matrix removes the memory bottleneck but requires performing communications. The MURGE API proposes to hide all this communication process to the user and implements them efficiently.

Listing 6 Dynamic CSC structure.

```
typdef struct dynamic_csc_s {
  int        n;
  long int   nz;
  int        dof;
  int       *colsizes;
  int      **rows;
  double   **values;
} dynamic_csc_t;
```

In PaStiX implementation of MURGE, entries are stored in a dynamic CSC. Then, when the end of assembly is reached, non local columns are sent to their owners. This way we obtain a dynamic CSC that contains only the local columns, and we can build a CSCD suited to PaStiX. If the preprocessing has been performed PaStiX internal column distribution gives the column locality of the CSCD. Else, the local columns are chosen following the entered (i, j) couples. If multiple processes own a same column, the one with more rows entries, or with the lower rank is they store the same number, is elected. Thus, if the user distribution is not evenly distributed our CSCd will also be badly balanced. We have tried a balancing algorithm to correct this problem but the cost of such an algorithm is too heavy compared to the assembly cost. Thus, we assume that the original assembly is well balanced. This is usually the case due to the graph partitioning of the problem.