N° d'ordre : 3922

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Mathieu FAVERGE**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Ordonnancement hybride statique-dynamique en algèbre linéaire creuse pour de grands clusters de machines NUMA et multi-cœurs

---

Soutenue le : 7 décembre 2009

Après avis des rapporteurs :

M.	Jean-François	MÉHAUT	Professeur, UJF Grenoble
M.	Frédéric	VIVIEN	Chargé de recherche, INRIA

Devant la commission d'examen composée de :

M.	Luc	GIRAUD	Directeur de recherche, INRIA	Président
M.	Guido	HUYSMANS	Chercheur, CEA Cadarache	Examinateur
M.	Jean-Yves	L'EXCELLENT	Chargé de recherche, INRIA	Examinateur
M.	Jean-François	MÉHAUT	Professeur, UJF Grenoble	Rapporteur
M.	Raymond	NAMYST	Professeur, Univ. de Bordeaux I	Directeur de thèse
M.	Jean	ROMAN	Professeur, ENSEIRB	Directeur de thèse

– 2009 –

Remerciements

Je souhaite tout d'abord remercier ceux qui ont acceptés d'être rapporteurs sur ce manuscrit : Jean-François Méhaut, qui m'a permis de faire le tour de la France avec les réunions ANR NUMASIS, et Jean-Yves L'Excellent, que je tiens tout particulièrement à remercier pour l'attention avec laquelle il a relu ma thèse, alors qu'il craignait de ne pas tout comprendre :) et pour les discussions que nous avons eues qui m'ont permises de peaufiner ce manuscrit. Je remercie également Frédéric Vivien qui a accepté de co-signer le rapport de Jean-Yves, et sans qui, il n'aurait pas pu rapporter ma thèse.

Je souhaite également remercier tous les autres membres de mon jury : mes deux directeurs de thèse Jean Roman et Raymond Namyst qui m'ont proposés spontanément ce sujet de thèse et qui m'ont fait confiance pour ces trois années. Merci également à Luc Giraud d'avoir accepté de présider mon jury de thèse et pour les remarques qu'il a apportées à mon manuscrit.

Enfin merci aux deux personnes qui m'ont encadré pendant ces trois années : Olivier Aumage, avec qui je n'ai malheureusement pas travaillé autant que je l'aurais souhaité, mais qui a toujours été là pour corriger mes *fôte d'orthogaphe*, et merci aussi de ne pas m'avoir demandé de recoder les fourmis et les matchs de foot dans PaStiX ;). Merci à Pierre Ramet pour ces trois super années de thèse passées ensemble, que ce soit dans le même bureau, en conférence ou pour m'avoir laissé exploser le code de PaStiX en plusieurs fichiers (à l'aide de Xavier), ou m'avoir montré l'utilité des macros. Mais également pour tout les bons moments passés ensemble en dehors du boulot : sur les terrains de squash ou sur un vélo où il était devant comme sur les pistes de skis ou de kart où il était loin derrière;), pour toujours finir autour d'un barbeuk improvisé ou d'une bière "locale". Enfin et surtout, merci à lui pour avoir passé 4 WE enfermé dans les préfas à relire ma thèse, mais également merci à Marieve de me l'avoir laissé pendant ces quatre week-ends :).

Je tiens également à remercier :

Xavier, pour le boulot énorme qu'il a fait sur PaStiX, pour sa réactivité à résoudre les problèmes et qui m'a souvent facilité la tâche dans les développements de ma thèse.

Tous mes relecteurs, parce que c'était pas toujours évident de me relire, par ordre de relecture, ma chérie, Olivier, Pierre, ma mère, mes grands-parents, ma mère, ma mère, mon oncle Patchi (Juste parce qu'il voulait pas être cité :)), ma mère, ma mère, ... Je sais pas combien de fois elle l'a relue, parfois sans comprendre ce qu'elle lisait, mais en tout cas un grand MERCI.

Guillaume sans qui je n'aurais pas eu de superbes maçons, ainsi qu'un R2D2 dans ma présentation.

L'INRIA pour ses multiples déménagements et l'incendie qui m'ont fait avoir une quantité de co-bureaux phénoménales en trois ans.

Tout mes co-bureaux qui m'ont écouté parler bien plus souvent qu'ils ne le souhaitaient : Benji, Cheche, Adam, Jérémie, Gaple, Jirka, Robin, Guillaume L. pour ses réservations au Café Bleu, Pierre, Hocine, toute l'équipe Bacchus dans le A29, et les derniers à m'avoir supporté durant la rédaction et les mois qui ont suivis : Christelle, Guillaume, Pascal et Xavier qui m'ont également remis au Rubik's Cube et avec qui on a fait de jolies soirées Wii.

Ceux que j'ai oublié ou avec qui il a quand même manqué un ou deux déménagements de la part de l'INRIA pour qu'on soit dans le même bureau Pascal, Mario, Abdou, Nico, Dam's, Algiane, Seb, Cédric, Arnaud, Jun Ho, Francois, ...

L'équipe Runtime également qui était ma deuxième équipe et avec qui j'ai passé de super moments : Raymond pour m'avoir laissé tomber tout habillé dans sa piscine, ses petits montres pour m'avoir dit que c'était pas bien, Gonnet, Francois, Juliette, Babeth, Cecile, Cyril, Brice et Emilie pour les soirées passées ensemble, Christophe pour m'avoir poussé dans la piscine, Broq, Jéjé, Steph, Nat, Pierre-André, Marie-Christine, Alex, Guillaume, ...

Brice et Nathalie pour toujours avoir accepté de prendre le temps pour relire mon anglais abominable.

Tout le groupe GANA du CEA, pour le stage de dernière année passé avec eux, mais également pour les soirées d'escalade avec Agnès et Michel. Merci aussi à David de toujours nous demander plus, tout en nous montrant constamment qu'il est satisfait de ce qu'on lui donne et qui par conséquent donnait un sens et une motivation à notre travail.

Arthur, Clément, Johnny, Jule, Kevin, Olivier et Pascal pour avoir vite fait du super boulot en PFA et m'avoir permis d'améliorer mes résultats sur des cas importants.

Ces remerciements ne seraient pas complets sans ma famille et mes amis : Le groupe i2pfa1, ainsi que les pièces rapportées : Bruno, Lolo, Kevin, Mathieu, Tom, Morgane, Antoine, Jérémie et Laure pour toutes les soirées passées ensemble depuis maintenant 6 ans et les séjours à l'étranger ou au ski qu'on a pu faire.

Les lyonnais qui m'ont fait le plaisir de faire le déplacement jusqu'à Bordeaux pour venir assister à ma soutenance : Camille, Lionel et Rémi, ainsi que l'e3 resté sur Bordeaux : Hervé.

Le café bleu, pour ses galettes et ses crêpes choco-citron. Les joueurs de squash et le club de Talence pour ses tournois où je finis toujours en milieu de tableau et jamais mieux ;). Le picotin et le Dream's pour ses soirées.

Ad, Jérémie et Bruno pour être restés avec moi jusqu'à pas d'heure le dernier soir où je rédigeais ma thèse et que je n'arrivais pas obtenir les résultats souhaités.

Bruno et Lolo pour tous les objets qu'ils ont réussi, parfois je ne sais pas comment, à glisser dans ma boîte aux lettres.

Mon frère pour être venu apprendre l'anglais à Bordeaux avec Boubou. Il ne te reste plus qu'à venir dans le Tennessee :).

Tous ceux que je n'ai pas cité.

Enfin merci à mes parents, grand-parents, beaux-parents et à Ad d'avoir organisé un super pot de thèse sans que j'aie grand chose à faire et merci surtout à Ad de m'avoir supporté pendant la rédaction, et tout le reste du temps aussi d'après Christelle.

Ordonnancement hybride statique-dynamique en algèbre linéaire creuse pour de grands clusters de machines NUMA et multi-cœurs

Résumé :

Les nouvelles architectures de calcul intensif intègrent de plus en plus de microprocesseurs qui eux-mêmes intègrent un nombre croissant de cœurs de calcul. Cette multiplication des unités de calcul dans les architectures ont fait apparaître des topologies fortement hiérarchiques. Ces architectures sont dites NUMA. Les algorithmes de simulation numérique et les solveurs de systèmes linéaires qui en sont une brique de base doivent s'adapter à ces nouvelles architectures dont les accès mémoire sont dissymétriques. Nous proposons dans cette thèse d'introduire un ordonnancement dynamique adapté aux architectures NUMA dans le solveur PaStiX. Les structures de données du solveur, ainsi que les schémas de communication ont dû être modifiés pour répondre aux besoins de ces architectures et de l'ordonnancement dynamique. Nous nous sommes également intéressés à l'adaptation dynamique du grain de calcul pour exploiter au mieux les architectures multi-cœurs et la mémoire partagée. Ces développements sont ensuite validés sur un ensemble de cas tests sur différentes architectures.

Mots clés :

Parallélisme, ordonnancement dynamique, systèmes linéaires creux, méthodes directes, architectures NUMA.

Discipline :

Informatique

LaBRI (UMR CNRS 5800) et Projets INRIA Bordeaux - Sud-Ouest Bacchus[1] et Runtime[2]
Bâtiment A29 bis
351, cours de la libération
33405 Talence Cedex, FRANCE

[1] http://bacchus.bordeaux.inria.fr/
[2] http://runtime.bordeaux.inria.fr/

Static-Dynamic Hybrid Scheduling in sparse linear algebra for large clusters of NUMA and multi-cores architectures

Abstract :

New supercomputers incorporate many microprocessors which include themselves one or many computational cores. These new architectures induce strongly hierarchical topologies. These are called NUMA architectures. Sparse direct solvers are a basic building block of many numerical simulation algorithms. They need to be adapted to these new architectures with Non Uniform Memory Accesses. We propose to introduce a dynamic scheduling designed for NUMA architectures in the PaStiX solver. The data structures of the solver, as well as the patterns of communication have been modified to meet the needs of these architectures and dynamic scheduling. We are also interested in the dynamic adaptation of the computation grain to use efficiently multi-core architectures and shared memory. Experiments on several numerical test cases will be presented to prove the efficiency of the approach on different architectures.

Keywords :

Parallelism, dynamic scheduling, sparse direct solver, sparse linear system, NUMA architectures

Discipline :

Computer science

LaBRI (UMR CNRS 5800) and INRIA Bordeaux - Sud-Ouest Projects Bacchus[3] and Runtime[4]
Bâtiment A29 bis
351, cours de la libération
33405 Talence Cedex, FRANCE

[3]http://bacchus.bordeaux.inria.fr/
[4]http://runtime.bordeaux.inria.fr/

Table des matières

Introduction **1**

1 État de l'art **5**

1.1 La résolution de systèmes linéaires creux . . . 6
1.2 Les solveurs directs . . . 8
1.2.1 Problème de remplissage : renumérotation des inconnues . . . 8
1.2.2 Factorisation symbolique . . . 10
1.2.3 L'algorithme séquentiel de factorisation . . . 11
1.2.4 Différentes méthodes de factorisation parallèle . . . 13
1.2.5 Résolution des systèmes triangulaires . . . 13
1.3 Problématique des architectures récentes . . . 14
1.3.1 Machines multi-processeurs symétriques . . . 14
1.3.2 Multi-cœurs et accès mémoire hiérarchiques . . . 16
1.3.3 Solutions pour l'utilisation de ces architectures . . . 18
1.4 Comment les solveurs répondent à cette problématique . . . 21
1.5 Choix d'un solveur de référence pour l'étude . . . 22
1.6 Description de l'algorithme utilisé par PaStiX . . . 23
1.6.1 Distribution et ordonnancement . . . 23
1.6.2 Factorisation numérique . . . 24
1.7 Discussion . . . 26

2 Optimisation de l'utilisation mémoire pour architectures hiérarchiques **29**

2.1 Accès mémoire non-uniformes . . . 30
2.2 Problèmes de contention . . . 34
2.3 Améliorations de la gestion mémoire . . . 37
2.4 Présentation des cas tests et des clusters utilisés pour l'étude . . . 39
2.5 Evaluation sur le solveur PaStiX . . . 42

3 Gestion des communications adaptées à un ordonnancement dynamique 43

3.1 Les schémas de communication dans le HPC . . . 44
3.2 Utilisation d'un thread de progression . . . 45
3.3 Une solution plus portable . . . 48
3.4 Évaluation des solutions proposées sur le solveur PaStiX . . . 51

4 Algorithme d'ordonnancement dynamique pour architectures NUMA 55

4.1 Description de la solution statique . . . 56
4.1.1 Repartitionnement et affectation des processeurs candidats . . . 56
4.1.2 Algorithme de distribution et d'ordonnancement . . . 59
4.2 Description de la solution dynamique . . . 61
4.2.1 Nouvel algorithme de distribution des données . . . 62
4.2.2 Ordonnancement dynamique des calculs . . . 64
4.3 Validation . . . 69
4.3.1 Version en mémoire partagée . . . 69
4.3.2 Version hybride MPI/Threads . . . 70

5 Ordonnancement dynamique à grains plus fins pour le solveur PaStiX 75

5.1 Description des méthodes de distributions 1D et 2D . . . 76
5.1.1 Distribution 1D . . . 76
5.1.2 Distribution 2D . . . 78
5.1.3 Distibution 2D dynamique . . . 80
5.2 Validation . . . 83
5.2.1 Version en mémoire partagée . . . 83
5.2.2 Version hybride MPI/Threads . . . 84
5.3 Validation sur un cas challenge . . . 86

Conclusion et perspectives 93

Annexes 97

A ViTE : Un logiciel de visualisation de traces d'exécution 97

B Utilisation des options implémentées dans le solveur PaStiX 99

Bibliographie 101

Liste des publications 107

Introduction

Dans de nombreux domaines de recherche les scientifiques ont désormais recours à des simulations numériques plutôt qu'à des expériences coûteuses et difficiles à mettre en place. On peut citer à ce titre le domaine de l'aérodynamique pour la conception de technologies de pointe comme les avions ou les formules 1. Les séances en soufflerie sont remplacées progressivement par des simulations numériques d'écoulement de fluides. La sismologie est aussi un domaine très demandeur de simulation numérique pour prédire l'impact d'un tremblement de terre sur une zone géographique ou pour évaluer les risques de tremblements suite à une première secousse. On peut citer également l'électronique, la biologie, la météorologie, la physique ou l'économie. Cette liste est loin d'être exhaustive mais montre l'importance de ce domaine dans les technologies actuelles.

Une majorité de ces problèmes de simulation numérique peuvent se ramener à des problèmes d'algèbre linéaire creuse, c'est-à-dire à un système d'équations linéaires de taille N dont le nombre de coefficients non nuls est de l'ordre de $O(N)$. Cependant, de nombreux problèmes des domaines cités précédemment comptent plusieurs millions d'inconnues, voir plusieurs dizaines de millions. Différentes méthodes ont été conçues pour résoudre ces systèmes linéaires. Ces méthodes très coûteuses en temps de calcul ont bénéficié des avancées technologiques en terme de microprocesseur pour résoudre ces problèmes de plus en plus rapidement. L'arrivée du parallélisme dans le domaine du calcul hautes performances a permis à ces méthodes de résoudre des systèmes encore plus importants et plus rapidement. En effet, certains domaines comme la météorologie ont besoin de résoudre de très grands systèmes avec des temps de calcul réduits pour obtenir une simulation en temps réel et non a posteriori.

Aujourd'hui, les constructeurs intègrent désormais du parallélisme au sein même des microprocesseurs. Ce sont les nouveaux processeurs multi-cœurs. Ces puces ont été développées pour répondre au besoin de concevoir des microprocesseurs de plus en plus rapides à proposer au grand public. En effet, on a constaté que l'augmentation des fréquences d'horloge n'est plus possible car elle entraîne une augmentation de la consommation électrique et de la dissipation thermique des microprocesseurs. Pour remédier à ce problème, les constructeurs ont exploité les technologies de miniaturisation pour graver plusieurs processeurs côte à côte sur la même puce avec quelques liens pour les interconnecter. Désormais, la fréquence d'horloge reste bloquée à quelques GHz et le nombre de cœurs est progressivement augmenté. Deux ou quatre cœurs, il y a trois ans, contre six ou huit actuellement. Intel a déjà présenté, en décembre 2007, à l'*International Solid-State Circuits Conference* de San-Francisco, un prototype nommé Polaris, qui comportait 80 cœurs. La multiplication du nombre des processeurs et aujourd'hui la multiplication du nombre de cœurs de calcul dans les processeurs a introduit une sophistication des puces pour que ces cœurs puissent échanger efficacement et accéder rapidement à la mémoire. Cette sophistication a apporté une topologie fortement hiérarchique dans les nouvelles architectures utilisant ces microprocesseurs et des temps d'accès à la mémoire non-uniformes. On parle d'architectures NUMA (Non-Uniform Memory Access).

L'objectif principal de cette thèse est de proposer des solutions pour adapter les bibliothèques de résolution de systèmes linéaires creux à ces nouvelles architectures dites NUMA et de les valider au sein du solveur PaStiX développé à Bordeaux avec des problèmes issus des collaborations avec le CEA/Cesta, le BRGM et les autres partenaires des projets ANR Numasis et Solstice. Les problèmes de résolution de systèmes linéaires creux ont des graphes de dépendance de tâches sous forme de graphes directs acycliques qui peuvent se réduire à des arbres dans certains cas particuliers. Le but est de trouver un procédé qui permette d'ordonnancer efficacement et dynamiquement un tel arbre de tâches sur ces nouvelles architectures. Le second objectif de cette thèse est d'étudier les problèmes de couplage entre des ordonnancements dynamiques des calculs et une distribution statique des données. Pour cela, nous voulons exploiter la solution existante dans le solveur PaStiX basée sur des modèles de coût et une simulation en temps de l'algorithme principal. Cette solution permet actuellement de prédire un ordonnancement statique qui est un bon point de départ pour construire une solution d'ordonnancement dynamique, notamment pour la distribution préalable des données. En effet, la topologie hiérarchique des nouvelles architectures implique qu'il devient difficile d'établir des modèles de coûts des opérations élémentaires qui prennent en compte les accès hétérogènes et la contention de la machine.

Nous présenterons dans le premier chapitre les différentes méthodes mises au point pour la résolution des systèmes linéaires creux et, plus particulièrement, nous nous intéresserons au fonctionnement des méthodes directes utilisées dans cette thèse. Ce chapitre présentera ensuite l'état de l'art des architectures actuelles et les outils existants pour les exploiter. Nous terminerons ce chapitre par un bilan des solutions proposées par les bibliothèques de résolution de systèmes linéaires creux pour utiliser efficacement les clusters de machines multiprocesseurs actuelles et par une présentation détaillée du solveur PaStiX retenu pour cette étude.

Les chapitres 2 et 3 présenteront les travaux préalables à l'élaboration d'un ordonnancement dynamique sur cluster de machines NUMA. Les architectures NUMA ayant des accès mémoire hétérogènes, nous avons souhaité évaluer les variations des coûts d'accès à la mémoire et notamment des fonctions d'algèbre linéaire couramment utilisées dans les bibliothèques BLAS. Le chapitre 2 étudiera le comportement des accès mémoire sur ces architectures en fonction du placement des données et des calculs. Ces résultats conduiront aux modifications apportées sur les structures de données du solveur PaStiX pour répondre aux besoins de ces architectures. Le chapitre 3 s'intéressera plus particulièrement aux problématiques de progression des communications. Les codes de simulation exploitant les clusters d'ordinateurs reposent sur un bon recouvrement des communications par les calculs grâce à une progression asynchrone efficace des échanges proposée par les bibliothèques MPI. Dans un ordonnancement dynamique, cette progression est primordiale pour libérer plus rapidement les tâches de calcul et disposer d'un choix plus vaste dans l'ordonnancement. Dans le cadre de ce chapitre, nous proposons une solution utilisant un thread dédié pour faire progresser les communications de façon asynchrone dans l'ordonnancement statique et permettre une progression plus uniforme de celles-ci quelle que soit la bibliothèque MPI utilisée.

Une fois ces développements préalables présentés, nous aborderons dans le chapitre 4 les problèmes d'ordonnancement et de distribution des données. La première section du chapitre présentera la solution d'ordonnancement statique du solveur PaStiX basée sur des modèles de coût des opérations matricielles et vectorielles élémentaires et sur une simulation en temps de l'algorithme principal grâce à ces modèles. Nous proposons ensuite une solution d'ordonnancement dynamique basée sur cette distribution statique des données pour conserver au maximum les données au plus proche de l'unité de calcul qui les exploitera.

Le chapitre 5 soulèvera le problème du grain de calculs sur les machines multi-cœurs. Les performances du solveur PaStiX proviennent de la prédiction statique performante faite en amont de la factorisation numérique pour ordonnancer les calculs. Le problème de cette prédiction est qu'elle devient très coûteuse lorsque l'on augmente le nombre de tâches de calcul en modifiant la méthode de distribution des données. Nous proposons dans ce chapitre une solution qui exploite l'ordonnancement dynamique pour créer dynamiquement des tâches de calcul à grain plus fin. De cette façon, on évite le surcoût lors de la phase de prétraitement du solveur. Nous terminerons ce chapitre par une évaluation de l'ensemble de ces développements sur un cas challenge à 10 millions d'inconnues issu de notre collaboration avec le CEA/Cesta. Nous montrerons également sur ce cas challenge les différences d'ordonnancement des tâches de calcul sur des traces d'exécution en fonction de la méthode utilisée.

Nous conclurons par un résumé des résultats obtenus lors de cette thèse et sur les perspectives ouvertes par ces travaux et les futures évolutions architecturales des ordinateurs.

Chapitre 1

État de l'art

Sommaire

1.1 La résolution de systèmes linéaires creux 6

1.2 Les solveurs directs 8

1.2.1 Problème de remplissage : renumérotation des inconnues 8

1.2.2 Factorisation symbolique 10

1.2.3 L'algorithme séquentiel de factorisation 11

1.2.4 Différentes méthodes de factorisation parallèle 13

1.2.5 Résolution des systèmes triangulaires 13

1.3 Problématique des architectures récentes 14

1.3.1 Machines multi-processeurs symétriques 14

1.3.2 Multi-cœurs et accès mémoire hiérarchiques 16

1.3.3 Solutions pour l'utilisation de ces architectures 18

1.4 Comment les solveurs répondent à cette problématique 21

1.5 Choix d'un solveur de référence pour l'étude 22

1.6 Description de l'algorithme utilisé par PaStiX 23

1.6.1 Distribution et ordonnancement 23

1.6.2 Factorisation numérique 24

1.7 Discussion 26

La résolution de systèmes d'équations linéaires creux est une brique importante du calcul hautes performances et de la simulation numérique. Pour résoudre ces problèmes, les scientifiques ont mis au point de nombreuses méthodes de résolution adaptées à leurs besoins et à leurs contraintes. Nous présenterons succinctement ces différentes solutions avec leurs intérêts et leurs inconvénients, et de façon plus détaillée le fonctionnement des solveurs directs qui nous intéressent dans cette thèse. D'autre part, la demande en puissance de calcul n'a cessé d'augmenter au cours des dernières années. Notamment pour la résolution de systèmes linéaires qui est la partie des codes de simulation la plus gourmande en ressources mémoire et en temps de calculs. Ceci a conduit à une évolution rapide et complexe des architectures utilisées. Le niveau de parallélisme a augmenté avec la multiplication du nombre d'unités de calcul par ordinateur, et la complexité de ces derniers également avec l'apparition d'architectures hiérarchiques. Pour s'adapter à ces nouvelles architectures, nous présenterons les différentes possibilités offertes aux programmeurs pour adapter les codes à ces nouveaux environnements.

L'étude de problèmes de taille plus importante est souvent limitée par la résolution des systèmes linéaires à base de méthodes directes qui demandent trop de ressources. Ce sont ces solveurs que les scientifiques cherchent à optimiser en priorité. Nous verrons comment les solveurs directs s'adaptent aux architectures actuelles pour offrir la possibilité de résoudre des problèmes de taille de plus en plus importante dans un laps de temps raisonnable. PaStiX est un des solveurs directs existants qui permet déjà d'exploiter efficacement les architectures SMP présentes depuis quelques années dans le TOP500. Ce solveur direct a été développé à Bordeaux à l'origine par Pascal Hénon et Pierre Ramet lors de leurs thèses. Nous expliquerons pourquoi nous avons choisi d'orienter les travaux de cette thèse sur ce solveur et, afin de mieux comprendre les apports de ces travaux, nous détaillerons l'algorithme utilisé par ce solveur pour la factorisation de matrices creuses.

1.1 La résolution de systèmes linéaires creux

Quelque soit le domaine dans lequel la simulation numérique est utilisée, on retrouve souvent la nécessité de résoudre des systèmes linéaires de la forme $Ax = b$ avec A une matrice de grande taille et très creuse. Cette matrice est dite creuse car le nombre de termes non nuls qu'elle contient est de l'ordre $O(n)$ où n est la taille de la matrice, ce qui est faible par rapport au nombre de termes nuls. Ce faible remplissage vient de la construction de la matrice. Les coefficients de A représentent généralement l'influence qu'ont les points du maillage utilisés dans la simulation les uns sur les autres et un grand nombre de ces coefficients ont une valeur nulle. En effet dans l'exemple d'une simulation de météorologie mondiale, il apparaît évident que la météo de la ville de Knoxville au Tennessee n'a aucune influence sur celle de Bordeaux en France, le coefficient de la matrice liant ces deux villes est nul. Les variables b et x sont deux vecteurs de taille n avec x l'inconnue du problème. Ils peuvent avoir plusieurs significations, mais si on reprend notre exemple de météorologie, on peut considérer que b est la météo du jour J et x, celle du jour $J + 1$.

La solution de cette équation est donnée par la formule $x = A^{-1}b$. Cependant, A^{-1} est inutilisable car contrairement à A, elle est habituellement dense et par conséquent impossible à stocker en mémoire et trop coûteuse à calculer. Elle est également très instable et est déconseillée même avec des matrices denses. Pour contourner le problème de l'inversion de la matrice A, diverses solutions existent et permettent de résoudre des problèmes de plus en plus importants. En effet, les besoins et les contraintes des différents codes de calcul et architectures peuvent varier énormément : précision voulue, stabilité de la structure et/ou des données de la matrice au cours du temps, taille du problème, structure et complexité du problème, ... Pour résoudre ces problèmes, on distingue dans un premier temps deux méthodes principales de résolution : directe et itérative.

Les méthodes directes sont basées sur la décomposition de la matrice A en un produit de deux matrices triangulaires respectivement inférieure et supérieure LU. Pour les matrices symétriques définies positives, on utilise la factorisation de Cholesky : $A = LL^T$ avec L triangulaire inférieure, ou pour les matrices seulement symétriques $A = LDL^T$ avec D diagonale. Ces deux dernières factorisations réduisent de moitié la quantité de calculs à réaliser et la quantité de mémoire nécessaire par rapport à une décomposition LU. Elles sont souvent préférées à la décomposition LU qui est une solution plus générale pour ces caractéristiques. Les méthodes directes ont l'avantage de fournir une très grande précision sur la solution et une fois que la matrice est factorisée, on peut résoudre très rapidement le système pour différents seconds membres b. Ces techniques se parallélisent relativement simplement mais elles sont rapidement limitées par

l'espace mémoire nécessaire au stockage de la décomposition de A et par la demande importante en ressources de calcul.

Les méthodes itératives calculent la solution de l'équation $Ax = b$ à partir d'une valeur initiale de x qui est successivement raffinée pour se rapprocher de la solution cherchée. La plupart des méthodes itératives : Gauss-Seidel, Jacobi, Gradient Conjugué, GMRES sont décrites dans [90]. Contrairement aux méthodes directes, ces algorithmes sont peu coûteux en mémoire et nécessitent peu de calcul en comparaison de la factorisation nécessaire aux méthodes directes. Ainsi, il est possible de résoudre des problèmes de taille beaucoup plus importante dans des temps de calculs inférieurs à ceux d'un solveur direct. De plus dans de nombreux problèmes, il n'est pas nécessaire d'avoir une très grande précision sur le résultat, une précision de 10^{-5} ou 10^{-6} suffit. Les méthodes itératives sont alors plus intéressantes pour ces problèmes car elles peuvent atteindre cette précision en quelques itérations. Une précision plus importante peut également être obtenue par ces méthodes, mais souvent le coût du nombre d'itérations nécessaires est trop important pour qu'elles soient préférées aux solveurs directs. Il est possible également que dans certains cas la méthode ne converge pas vers la solution, particulièrement quand la précision demandée est importante. De plus, les méthodes itératives sont chacune spécifiques à des types de problèmes ou de structures de matrice et elles ont des comportements en termes de convergence très différents d'un problème à un autre. Enfin, la valeur initiale utilisée dans la résolution du système a aussi une influence importante sur la rapidité de convergence de chacune des méthodes. Les méthodes itératives sont souvent préférées aux méthodes directes sur les problèmes 3D car le remplissage sur ces problèmes est important et les méthodes directes sont plus coûteuses. En revanche sur les problèmes 2D, les méthodes directes sont quasiment imbattables car la matrice initiale possède peu de termes non nuls et le remplissage de la matrice factorisée est faible. La factorisation se fait alors très rapidement et la solution obtenue est meilleure que celle calculée avec une méthode itérative.

Des solutions ont été développées pour combiner les avantages des deux approches tout en essayant d'en éviter les inconvénients. Une des premières solutions consiste à réaliser une factorisation incomplète de la matrice en supprimant par exemple les coefficients dont la valeur est inférieure à un seuil choisi ou tout simplement en ne prenant pas en compte les termes de la matrice créés lors de la factorisation. Le résultat obtenu est ensuite utilisé comme *préconditionneur* du solveur itératif. Les techniques de préconditionnement consistent à introduire un nouveau terme dans le système pour faciliter sa résolution. Dans le cas, par exemple, d'un préconditionnement à gauche, on résout le système $P^{-1}Ax = P^{-1}b$. Cette solution permet une meilleure convergence de la méthode itérative car le vecteur initial est plus proche de la solution recherchée.

De nouvelles méthodes hybrides mixent ces approches à l'aide d'une décomposition de domaine [41]. Chaque sous-domaine est résolu par une méthode directe, puis le problème aux interfaces des domaines est résolu avec une méthode itérative préconditionnée par une factorisation incomplète des interfaces. Ces méthodes sont faciles à paralléliser en distribuant les sous-domaines sur les ressources de calculs et permettent d'obtenir une convergence plus rapide à précision demandée égale qu'avec une méthode itérative seule.

Enfin pour résoudre des problèmes de plus en plus grands et pour utiliser de plus en plus de processeurs, certains codes de calculs utilisent des méthodes multi-grilles. Le principe de ces méthodes est d'accélérer la convergence de la méthode itérative utilisée par des corrections de la solution globale calculées en résolvant une réduction du système d'origine. Ces méthodes comportent trois étapes : la réduction du problème qui est basée sur une réduction géométrique du maillage ou sur une réduction algébrique des coefficients de la matrice, le prolongement qui

reporte la solution calculée sur le système réduit au système étendu et le lisseur qui corrige les erreurs générées par le prolongement. Ces méthodes sont très appréciées car elles passent à l'échelle de façon linéaire et permettent d'exploiter facilement plusieurs milliers de processeurs en raison de leur faible taux de communications. Malgré cela, les techniques de restriction et de prolongation restent très spécifiques à un problème donné et il n'existe pas de solution efficace qui s'adapte à tous les types de problèmes.

Ainsi, de nombreuses solutions existent pour résoudre les systèmes linéaires. Nous avons vu que la plupart d'entre elles utilisent des méthodes itératives dont on améliore la convergence en préconditionnant le système par différentes méthodes. Cependant ces techniques restent souvent spécifiques à des problèmes donnés et ont des difficultés à converger sur les problèmes très mal conditionnés. Les méthodes directes restent par conséquent un bon choix sur ces problèmes comme la mécanique des structures, la magnétohydrodynamique (MHD) ou l'électromagnétisme. Nous allons voir dans la section suivante comment fonctionnent les méthodes directes et quelles sont les techniques utilisées pour les optimiser.

1.2 Les solveurs directs

Dans la suite de cette thèse nous allons principalement nous intéresser aux solveurs parallèles directs de systèmes linéaires creux. Ces solveurs requièrent une quantité importante de mémoire, car ils doivent stocker la matrice factorisée qui est plus volumineuse que la matrice initiale. Ils sont également très coûteux en ressources de calculs puisqu'ils font le calcul exact de la matrice factorisée contrairement aux autres solveurs qui ne calculent qu'une approximation de la solution. Malgré ces inconvénients, ces solveurs permettent d'obtenir la meilleure précision possible sur le résultat, il s'agit de la précision machine si le problème est bien conditionné. L'autre avantage de cette approche est le stockage de la matrice factorisée. En effet, dans les problèmes physiques où la matrice ne change pas au cours du temps, il suffit de la factoriser une fois pour résoudre le problème avec autant de seconds membres qu'on le souhaite. Ces solveurs sont très utilisés principalement sur les clusters avec beaucoup de mémoire à base d'Opteron ou de Xeon plutôt que sur les architectures BlueGene où la quantité mémoire par processeur est très faible. Ils reposent sur deux étapes de prétraitement importantes qui vont déterminer la quantité de calculs à réaliser lors de la factorisation. La première étape est la renumérotation des inconnues de la matrice, elle permet de limiter au maximum le remplissage induit par la factorisation. La seconde consiste à regrouper les coefficients de la matrice dans une structure par blocs pour permettre l'utilisation de routines BLAS de plus haut niveau [67]. Pour avoir une vue d'ensemble du domaine on se rapportera aux articles [35, 36, 37, 42, 45, 69, 83].

1.2.1 Problème de remplissage : renumérotation des inconnues

Un des points essentiels propre aux méthodes directes sur les matrices creuses est le problème du *remplissage* [45]. En effet, dans la factorisation de Cholesky qui consiste à factoriser une matrice initiale A sous la forme LL^T avec L une matrice triangulaire inférieure, L possède un nombre plus important de termes non nuls que A de par leur création au cours du processus de factorisation. Pour toute l'étude de cette thèse, nous nous placerons toujours dans le cas où la structure de la matrice est symétrique. Les algorithmes présentés par la suite s'adaptent ainsi facilement d'une factorisation de Cholesky à une décomposition LU en doublant les calculs de la partie triangulaire inférieure sur la partie triangulaire supérieure symétrique.

Un outil fondamental pour étudier ce remplissage est le *modèle de graphe* [45] associé à l'éli-

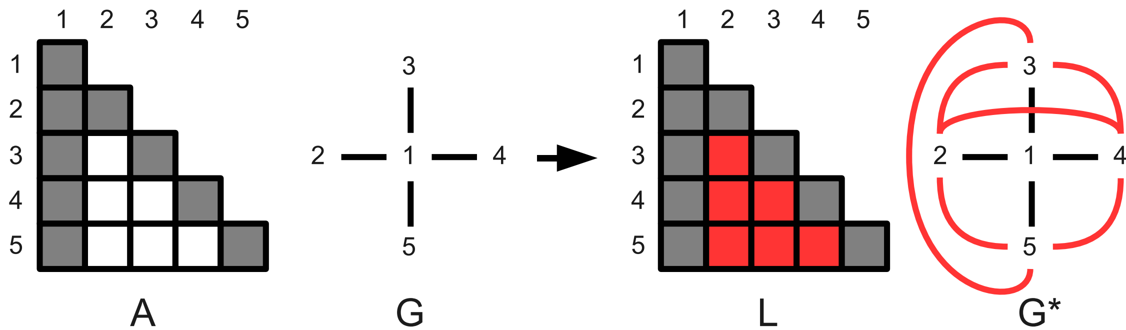
mination de Gauss. Ce remplissage étant directement lié à l'ordre d'élimination des inconnues, il s'agit donc de trouver une renumérotation de ces inconnues, c'est-à-dire une *renumérotation des sommets du graphe associé à la matrice initiale* [9, 45, 79, 80] qui minimise le remplissage et en même temps le nombre d'opérations à faire effectivement lors de la factorisation. La figure 1.1 illustre les problèmes liés au remplissage, représenté par les termes en rouge, et montre l'importance que peut avoir une bonne renumérotation sur la taille de la matrice factorisée et sur le nombre de calculs à réaliser.

Le graphe non orienté G associé à une matrice A de dimensions $n \times n$ symétrique ou à structure symétrique est un graphe à n sommets où il existe une arête (i, j) entre les sommets i et j si et seulement si $a_{ij} \neq 0$.

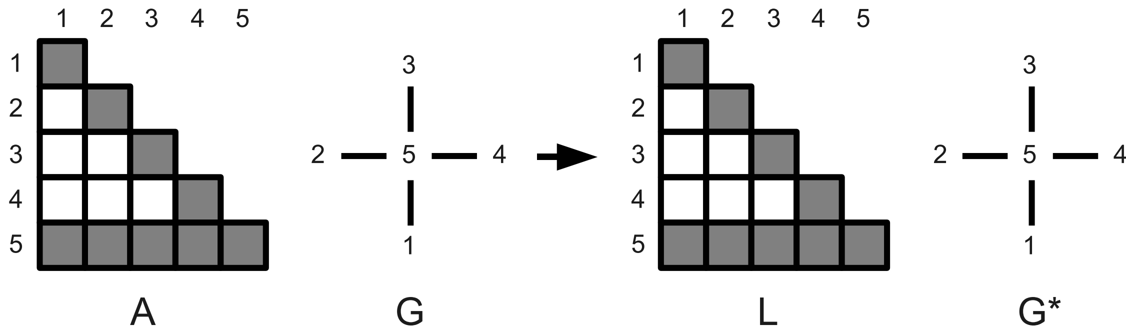
Le graphe d'élimination est le graphe G^* associé à la matrice L triangulaire inférieure obtenue par factorisation ; G^* a bien sûr le même nombre de sommets que G mais a (beaucoup) plus d'arêtes, car il contient toutes les arêtes correspondantes aux termes de remplissage (arêtes en rouge sur la figure 1.1(a)). Le remplissage de la matrice est calculé grâce au théorème de caractérisation suivant :

Théorème 1 (Théorème de caractérisation [86])

$$(i, j) \in G^* \Leftrightarrow \begin{cases} (i, j) \in G \\ ou \\ \exists \text{ un chemin } (j, k_1, \ldots, k_l, i) \text{ tel que } \forall p \in [\![1, l]\!], k_p < min(i, j) \end{cases}$$



(a) Sans renumérotation.



(b) Avec renumérotation.

FIG. 1.1 – Illustration de l'intérêt de la renumérotation des inconnues dans la factorisation de matrices creuses.

L'*arbre d'élimination* T [69] associé à la matrice factorisée est un arbre (dans le cas des matrices réductibles c'est une forêt), au sens classique de la théorie des graphes, à n sommets, et il existe une arête entre i et j si et seulement si la ligne du premier terme non nul dans la colonne j dans la matrice factorisée L se trouve être i.

L'arbre d'élimination joue un rôle fondamental pour la parallélisation des solveurs creux directs car il indique les dépendances dans les calculs : il ne peut y avoir de dépendances (et donc une séquentialité dans les calculs) entre deux éliminations d'inconnues que si les sommets associés sont sur une même branche de l'arbre d'élimination. Dans le chapitre 4, nous étudierons plus en détail les informations que l'arbre d'élimination peut fournir pour la distribution des données et l'ordonnancement des calculs.

Dans un cadre parallèle, le but est donc de trouver une renumérotation des sommets de G qui *minimise le remplissage* et qui *maximise l'indépendance dans les calculs de la factorisation*, c'est à dire conduisant à des arbres d'élimination *larges et de faible hauteur*. Les renumérotations performantes selon ces critères sont celles basées sur les techniques de "degré minimum" [9, 95] et de "dissections emboîtées" [45]. On peut noter que la numérotation classique de type Cuthill-McKee [26] est à écarter car elle occasionne plus de remplissage et conduit en plus à des arbres d'élimination très longs avec une faible indépendance dans les calculs.

1.2.2 Factorisation symbolique

Une autre étape fondamentale dans le déroulement des solveurs directs est celle dite de *factorisation symbolique par blocs* [24] qui va permettre de calculer algorithmiquement la structure creuse par blocs de la matrice L à partir de celle de A. Le but est d'optimiser la *factorisation numérique* en favorisant l'utilisation de routines BLAS matrices$\times$matrices et pour certains solveurs d'éviter la gestion effective, durant les calculs, des termes créés.

En termes de graphe, cela revient à calculer, pour une numérotation donnée et pour une partition P des sommets associée correspondant à un découpage en *blocs-colonnes* de la matrice, le *graphe quotient* G^*/P à partir de G et de cette partition P (voir figure 1.2). On appelle bloc-colonne, un ensemble de colonnes consécutives d'éléments de la matrice. L'intérêt de l'algorithme de factorisation symbolique par blocs repose sur le fait que pour les partitions P considérées dans ce travail, les opérations de passage au quotient et d'élimination commutent, c'est-à-dire qu'on a l'égalité $(G/P)^* = G^*/P$.

La complexité [24] en temps et en espace de cet algorithme croît alors comme le nombre total de blocs extra-diagonaux dans la structure ainsi construite pour L (voir figure 1.2). Ce nombre, et par conséquent la complexité en temps, dépend de la qualité de la numérotation vis-à-vis de la conservation du creux de A et de la partition P. Cette complexité est toujours très inférieure à celle de la factorisation numérique, ce qui justifie son intérêt. L'arbre d'élimination est alors un *arbre d'élimination par blocs* (arbre T sur la figure 1.2) et sa structure va décrire les dépendances dans les calculs par blocs du solveur.

Cet arbre d'élimination par blocs est un arbre recouvrant du graphe des échanges entre blocs-colonnes (voir figure 1.2) durant l'algorithme de factorisation par blocs (voir section 1.6 page 23). Ce graphe n'est autre que le graphe quotient G^*/P que l'on calcule donc symboliquement sous la forme $(G/P)^*$.

Dans un cadre parallèle, cette étape nous permettra la préparation, dans la phase de prétraitement, des données pour la distribution optimisée sur les processeurs.

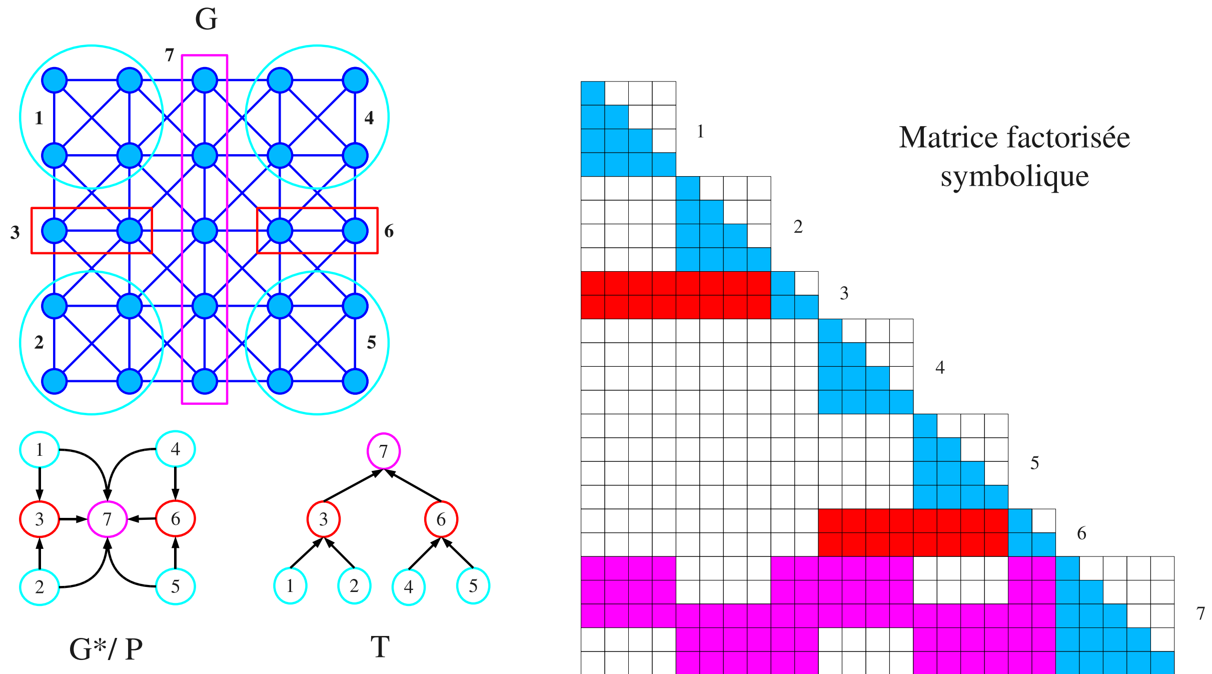


FIG. 1.2 – Un exemple de matrice factorisée L structurée par blocs-colonnes, avec le graphe G associé à la matrice A, le graphe quotient G^*/P et l'arbre d'élimination par blocs T associés. Chacun des 7 blocs-colonnes est constitué d'un bloc diagonal dense (en bleu) et de plusieurs blocs extra-diagonaux denses (en rouge et mauve).

1.2.3 L'algorithme séquentiel de factorisation

La structure par blocs de A obtenue avec l'arbre d'élimination dans l'étape précédente, nous permet de généraliser l'algorithme séquentiel élément par élément à un algorithme plus efficace qui va travailler directement sur les blocs (algorithme 1). L'utilisation des blocs, ou des blocs-colonnes de la matrice permet d'utiliser des routines d'algèbre linéaire qui sont optimisées pour le calcul sur de grands jeux de données.

L'algorithme présenté ici est celui de la factorisation LU pour une matrice A à structure symétrique. Dans le reste de cette thèse, nous utiliserons toujours une matrice A, dont on a symétrisé le graphe pour simplifier les notations dans les algorithmes utilisés. Cet algorithme se décline simplement pour les factorisations de Cholesky et Cholesky-Crout en quelques modifications. La première étape consiste à substituer $U_{k,(j)}$ par $L^T_{k,(j)} = L_{(j),k}$. Puis la symétrie de la matrice permet de supprimer la ligne 5 de l'algorithme et de réduire la boucle sur i à la ligne 8 à l'intervalle $[j, b_k]$. La quantité de calculs se retrouve ainsi divisée par deux.

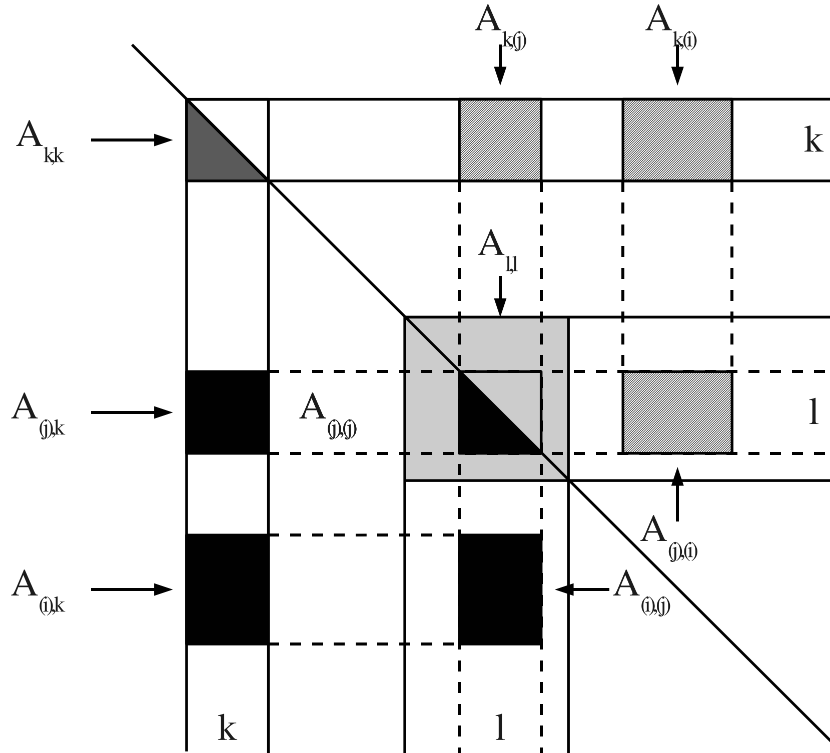


FIG. 1.3 – Illustration des notations utilisées dans l'algorithme de factorisation séquentielle.

Notations : Pour chaque bloc-colonne k, $1 \leq k \leq N$,
- $A_{k,k}$ est le bloc diagonal dense,
- b_k est le nombre de blocs extra-diagonaux dans le bloc-colonne k,
- $A_{(j),k}$ est le $j^{ième}$ bloc extra-diagonal dense avec $1 \leq j \leq b_k$, (j) étant un multi-indice décrivant un intervalle de lignes.

De plus, $A_{(i),(j)}$ est le bloc rectangulaire dense correspondant aux lignes du multi-indice (i) et aux colonnes du multi-indice (j). $A_{(j),(j)}$ correspond à la même notion mais pour un sous-bloc du bloc diagonal $A_{l,l}$ faisant face au bloc extra-diagonal $A_{(j),k}$.

Algorithme 1 Factorisation séquentielle par blocs : $A = LU$

1: **Pour** $k = 1$ à N **Faire**
2: Factoriser $A_{k,k}$ en $L_{k,k}.U_{k,k}$
3: **Pour** $i = 1$ à b_k **Faire**
4: Résoudre $L_{(i),k}.U_{k,k} = A_{(i),k}$
5: Résoudre $L_{k,k}.U_{k,(i)} = A_{k,(i)}$
6: **Fin de Pour**
7: **Pour** $j = 1$ à b_k **Faire**
8: **Pour** $i = 1$ à b_k **Faire**
9: $A_{(i),(j)} = A_{(i),(j)} - L_{(i),k}.U_{k,(j)}$
10: **Fin de Pour**
11: **Fin de Pour**
12: **Fin de Pour**

1.2.4 Différentes méthodes de factorisation parallèle

Il existe essentiellement deux types d'approche pour paralléliser l'algorithme séquentiel présenté précédemment. La parallélisation de l'algorithme de factorisation repose soit sur l'approche séquentielle *supernodale*, qui provient directement des algorithmes d'élimination par colonne réécrits par blocs et avec les variantes que l'on appelle *fan-in* ou *fan-out* [14, 78, 15], soit sur l'approche *multifrontale* [10, 11, 54, 55]. Ces approches diffèrent essentiellement sur la manière de répercuter les modifications (ou contributions) entre les blocs de la matrice (ligne 9 de l'algorithme 1) et ces différences se retrouvent dans les versions parallèles de ces algorithmes.

Dans l'approche multifrontale, on associe à chaque bloc-colonne une matrice *frontale* qui contient toutes les contributions du bloc-colonne générées lors du calcul de la ligne 9 de l'algorithme. Cette matrice frontale est ensuite transmise au père du bloc-colonne dans l'arbre d'élimination par bloc. Celui-ci somme toutes les matrices frontales qui lui sont transmises par ses fils avec ses propres contributions avant de transmettre à nouveau le résultat à son père. Dans ces méthodes, on essaie d'exprimer le maximum de parallélisme dans le découpage et le calcul des blocs denses représentés par les matrices frontales mais également dans le parallélisme de l'arbre en utilisant des méthodes de renumérotation qui l'élargissent au maximum. Une description sur la parallélisation unidimensionnelle ou bidimensionnelle de cette approche est donnée dans [10].

Dans les méthodes supernodales, on distingue deux approches qui prennent en compte différemment l'ajout des contributions. L'approche "right-looking" qui est celle présentée dans l'algorithme 1 factorise un bloc-colonne puis répercute définitivement les contributions du bloc-colonne qui viennent d'être calculées sur les blocs-colonnes *à sa droite*. L'approche "left-looking" inverse les phases d'ajout des contributions et de factorisation du bloc-colonne courant. La phase d'ajout des contributions des lignes 7 à 11 est effectuée avant la ligne 2 de l'algorithme pour ajouter les contributions des blocs-colonnes *à gauche* du bloc courant avant de le factoriser.

Dans un cadre parallèle, il y a ensuite deux manières de transmettre les données lors de la factorisation numérique. La première appelée "fan-in" consiste à compresser les volumes de données échangées. Toutes les contributions calculées sur un même processeur et destinées à un même bloc-colonne sont sommées dans un bloc (ou bloc-colonne) recouvrant ces contributions comme pour les matrices frontales des méthodes multifrontales. La seconde appelée "fan-out" consiste à envoyer au plus tôt les contributions telles qu'elles ont été calculées. Cette approche n'a de réel intérêt que dans une approche "right-looking".

Les méthodes "fan-out/right-looking" sont généralement utilisées pour la factorisation de matrices pleines où les contributions doivent être envoyées au plus tôt car les contraintes de précédence entre les calculs sont fortes. Dans le cadre de la factorisation parallèle d'une matrice creuse, l'approche "fan-in" est plus performante car elle permet de limiter considérablement la masse de communication. En effet comme on peut le voir sur la figure 1.3, dans le cas de matrices creuses les contributions sont souvent plus petites que le bloc (ou bloc-colonne) cible et le coût de ces "petites" communications est plus difficile à masquer. Par contre, le stockage local de ces agrégats génère un surcoût mémoire qui peut devenir très important, voire rédhibitoire dans la factorisation de grands problèmes [58].

1.2.5 Résolution des systèmes triangulaires

Une fois les matrices L et U calculées (ou L et D), la solution est obtenue par la résolution successive des systèmes suivants :

$$L.y = b \quad (1.1)$$

$$\text{(Uniquement en } LDL^T\text{)} \; D.z = y \quad (1.2)$$

$$U.x = z \text{ ou } L^T.x = z \quad (1.3)$$

La première phase dite de *descente* résout l'équation (1.1). La seconde phase dite *diagonale* divise le vecteur obtenu précédemment par la diagonale (équation (1.2)), elle n'est faite que dans le cas d'une factorisation de Cholesky-Crout (LDL^T). Enfin la dernière phase dite de *remontée* résout l'équation 1.3. Dans le cas d'une résolution mono-second membre (i.e. b est un vecteur), cette phase est peu coûteuse en nombre d'opérations par rapport à la factorisation et c'est pourquoi les données sont distribuées sur les nœuds en fonction de la factorisation [56, 66]. Dans le cas d'une distribution par blocs-colonnes, les éléments du vecteur b sont répartis comme la distribution des colonnes de A. Dans le cas d'une résolution multi-seconds membres où b est une matrice rectangulaire de taille $n \times m$ avec m le nombre de seconds membres si m est assez grand, il peut être intéressant de revoir la répartition des matrices A et b sur les nœuds de calcul.

1.3 Problématique des architectures récentes

Le public découvre depuis quelques années lors de l'achat d'ordinateurs personnels, les termes d'*Hyperthreading* et de *Multicore*. Derrière ces termes se cache la démocratisation des machines parallèles auparavant réservées aux calculs scientifiques. La *Loi de Moore* [74] qui annonce une puissance de calcul des ordinateurs doublée tous les deux ans ne peut plus être vérifiée depuis quelques années. En effet, la course aux fréquences d'horloge de plus en plus élevées n'est plus possible à cause des phénomènes de dissipation thermique. Cependant, les fondeurs continuent à améliorer les techniques de gravure du silicium et diminuent régulièrement la taille des transistors. La finesse de gravure actuelle permet désormais de graver plusieurs fois la structure qui constitue un seul microprocesseur : unités de décodages, de prédiction, d'exécution, cache L1, ... Ce sont les processeurs multi cœurs et c'est ce qui permet à la loi de Moore d'être encore vérifiée. Le calcul scientifique est très friand de ces nouveaux processeurs permettant de plus en plus de parallélisme. Mais ces puces sont également de plus en plus complexes et difficiles à exploiter pleinement. Nous allons voir dans cette section les types d'architectures parallèles utilisées dans le calcul hautes performances et les solutions proposées aux programmeurs pour les exploiter.

1.3.1 Machines multi-processeurs symétriques

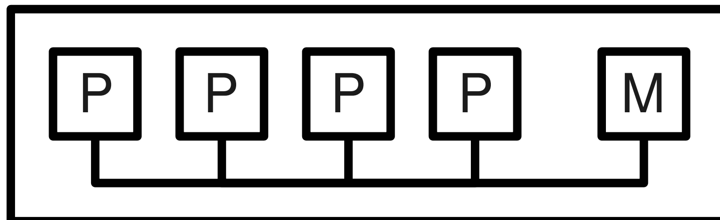


Fig. 1.4 – Architecture SMP. (P : processeur, M : mémoire).

L'architecture la plus simple de machine parallèle est l'architecture *Symmetric MultiProcessing* (**SMP**) telle qu'on peut la voir sur le schéma 1.4. Plusieurs processeurs sont connectés à une

mémoire commune via un bus de communications ou un commutateur parfait (*crossbar*). On se rend compte que cette architecture devient vite problématique avec l'augmentation du nombre de processeurs puisqu'ils partagent tous le même bus pour accéder à la mémoire disponible. IBM, par exemple, propose des architectures SMP avec les processeurs des différentes séries *Power* où 4 processeurs sont installés sur des cartes filles avec un contrôleur mémoire comme sur la figure 1.4. Puis 4 cartes filles sont connectées sur une carte mère avec des bus de données qui connectent tous les contrôleurs mémoire entre eux comme le montre la figure 1.5. On obtient alors une machine SMP avec par exemple 16 cœurs pour la machine Decrypthon du pôle M3PEC de la DRIMM [5] de l'Université de Bordeaux 1 qui contient 16 processeurs Power5 simple cœur, ou avec 32 cœurs pour la machine Vargas de l'IDRIS[6] qui contient 16 processeurs Power6 dual-core. Le nombre de cœurs disponibles sur les nouveaux processeurs permet d'augmenter le nombre d'unités de calcul mises à disposition de l'utilisateur simplement en faisant évoluer les processeurs sur l'architecture. De même, la technologie d'HyperThreading permet de multiplier *virtuellement* le nombre de cœurs disponibles en entrelaçant sur les processeurs deux flux d'instructions provenant de deux threads (ou processus) différents pour optimiser l'utilisation des unités de calcul du processeur.

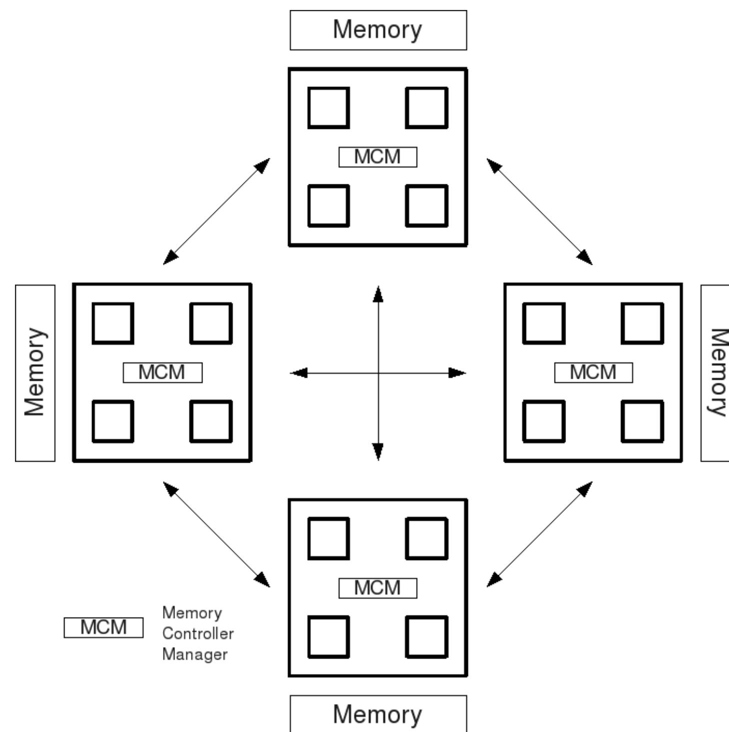


Fig. 1.5 – Architecture IBM d'un nœud du cluster *Decrypthon* (M3PEC) de la DRIMM de l'Université de Bordeaux 1 à base de Power5 et du cluster *Vargas* de l'IDRIS à base de Power6 dual-core.

La répétition du schéma de base permet d'atténuer le problème du nombre de processeurs par bus de données. Mais pour garder une architecture symétrique, il est nécessaire de rajouter des liens entre les contrôleurs mémoire de chacune des cartes filles. On comprend alors pourquoi

[5] Direction des Ressources Informatiques et Multimédia Mutualisées de l'Université de Bordeaux 1, `http://www.drimm.u-bordeaux1.fr/` et `http://www.m3pec.u-bordeaux1.fr/`.

[6] Institut du développement et des Ressources en Informatique Scientifique, `http://www.idris.fr/`.

il serait difficile de mettre plus de processeurs avec un tel schéma car le nombre de liens d'interconnexion croît très rapidement. Ces liens de communication permettent de masquer la localité de la mémoire à chacun des processeurs et de leur mettre à disposition un seul et unique ensemble. La multiplication des liens entre les cartes permet de dispatcher les communications pour ne pas effondrer la bande passante. Ce type d'architecture est très intéressant pour le calcul scientifique puisqu'il propose une augmentation du nombre d'unités de calcul avec un regroupement des ressources mémoire qui leur sont associées. Cependant cette architecture est vite limitée en nombre de processeurs qu'il est possible d'associer sur une même carte mère par le nombre de bus d'interconnexion à mettre en place entre les cartes filles.

1.3.2 Multi-cœurs et accès mémoire hiérarchiques

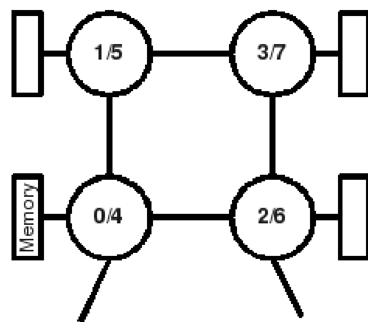
Les architectures SMP posent deux problèmes à l'augmentation du nombre de cœurs disponibles sur les machines :

1. la surcharge du bus mémoire unique ;
2. l'augmentation du nombre de cartes filles est trop coûteuse et complexe à cause des liens entre chaque carte.

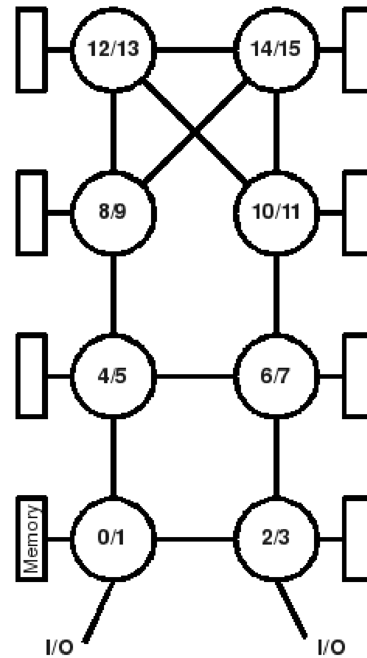
Afin de résoudre ces problèmes, les fabricants se sont orientés vers des architectures hiérarchiques. On parle d'architectures à accès mémoire non-uniformes (*Non-Uniform Memory Access, NUMA*). La mémoire est distribuée sur les différents nœuds du cluster comme sur l'architecture IBM. On peut aussi avoir une distribution de la mémoire par socket. La différence ici réside dans le nombre de bus de données disponibles. Contrairement à l'architecture IBM qui connecte tous les nœuds deux à deux, les architectures NUMA sont connectées de manière arborescente permettant ainsi d'augmenter le nombre de cartes filles pouvant être reliées sans obtenir un schéma très complexe de communication. Principalement utilisée dans les super-calculateurs, cette architecture se retrouve désormais au sein même des processeurs : le fondeur AMD intègre depuis quelques années dans ses puces Opteron un contrôleur mémoire dans le processeur pour pouvoir connecter directement la mémoire, ce qui augmente fortement la bande passante disponible pour les cœurs ainsi reliés. En revanche, cela crée un point de passage supplémentaire pour les autres cœurs souhaitant accéder à ces ressources. Pour cela, les processeurs Opteron disposent de quatre bus HyperTransport dont un est réservé aux accès mémoire comme le montrent les figures 1.6(a) et 1.6(b). Ceci augmente la disparité dans les accès mémoire, particulièrement sur les architectures comme celle dont dispose la machine Hagrid 1.6(b) de l'Université de Bordeaux 1. Le fondeur Intel intègre également ce mécanisme dans ses nouvelles puces Nehalem que l'on peut voir sur la figure 1.6(c). Cette figure représente l'architecture des nœuds du cluster Titane du CCRT[7] qui sont composés de deux Nehalem quad-cœur se partageant 24Go de mémoire. La mémoire n'est donc plus répartie sur les différents nœuds de calculs mais sur les processeurs, ajoutant ainsi un niveau de hiérarchie dans l'accès à la mémoire.

En plus de l'agencement des processeurs et de la mémoire, on découvre désormais les puces multi-cœurs qui intègrent 4 ou 8 cœurs sur une même puce avec plusieurs niveaux de cache mémoire agencés de manière hiérarchique. Enfin, Intel veut réintroduire le procédé d'Hyper-Threading dans ses puces ajoutant encore un niveau de hiérarchie. Ces nouvelles architectures NUMA apportent un gain important sur les débits mémoire et permettent d'augmenter considérablement le nombre de processeurs qui partagent la même mémoire. Certains constructeurs comme SGI Origin proposent ainsi des machines de 9728 cœurs en mémoire partagée comme celle

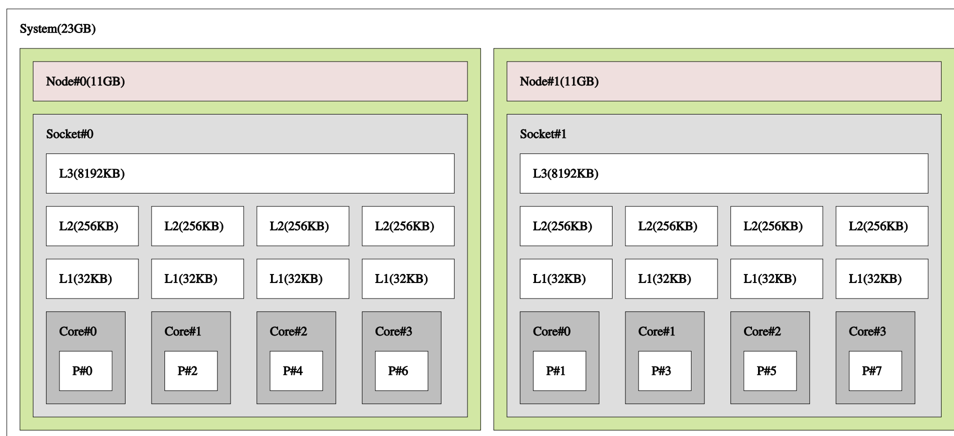
[7] Centre de Calcul Recherche et Technologie, `http://www-ccrt.cea.fr/`



(a) Nœud du cluster Borderline à base de 4 OPTERONS dual-core, Grid5000.



(b) Machine Hagrid à base de 8 OPTERONS dual-core, Université de Bordeaux 1.



(c) Machine Titane à base de 2 NEHALEM quad-core, CCRT.

FIG. 1.6 – Exemples d'architectures NUMA.

installée au LRZ de Munich depuis avril 2007 [3]. La difficulté réside alors dans la maintenance de la cohérence du cache. En réalité, pour obtenir un tel nombre de cœurs qui partagent la même mémoire, la machine est composée de 3712 *blades* de chacune 2 ou 4 cœurs. Elles sont toutes interconnectées grâce à un réseau propriétaire *NUMALink* qui est l'équivalent du bus mémoire sur une carte mère à l'échelle d'un cluster. La maintenance de la cohérence du cache est facilitée par la structure hiérarchique de ces connexions qui permettent de répercuter plus facilement et rapidement les défauts de cache trouvés par niveau, comme pour les caches de niveau 1, 2 et 3 au sein du processeur. Ces machines ouvrent la possibilité à de nouveaux algorithmes où il n'est plus nécessaire de communiquer les données, mais où il faudra prendre en compte encore plus

fortement la localité des accès mémoire.

Les constructeurs fournissent des architectures toujours plus performantes et de plus en plus parallèles au sein d'un seul ordinateur pour profiter de la facilité de programmation que propose la mémoire partagée. Cependant, comme on a pu le voir, ces architectures sont devenues très hiérarchiques pour augmenter le nombre de cœurs disponibles et améliorer les accès mémoire locaux. Mais il ne suffit pas que les fabricants le proposent pour que les programmeurs puissent en tirer pleinement parti. Les programmes doivent être adaptés à ces ressources. On peut facilement imaginer que si les données sont mal localisées sur la machine, les accès non uniformes à la mémoire peuvent faire chuter la bande passante. Nous allons désormais étudier quelles sont les solutions existantes pour exploiter ces nouvelles architectures.

1.3.3 Solutions pour l'utilisation de ces architectures

Les architectures présentées dans la section précédente sont difficiles à exploiter pleinement. On peut s'en rendre compte dans un premier temps en regardant les performances annoncées des clusters existants dans le monde dans le TOP500 [6]. Deux indices sont donnés : le premier est la performance maximale atteinte avec le benchmark LINPACK [32] sur le cluster. C'est la performance cible des codes exploitant au mieux la machine. Le second est la performance crête qui correspond à la performance théorique de tous les processeurs disponibles cumulée. On peut noter par exemple que la machine Tera10 du CEA installée en 2006 possède une puissance crête théorique de 65 TeraFlops et une puissance maximale de 53 TeraFlops obtenue avec le benchmark LINPACK. Mais elle a été conçue pour obtenir une puissance moyenne d'environ 10 TeraFlops sur les codes irréguliers plus courants, comme l'application visée dans cette thèse, que les codes très réguliers et très optimisés comme le sont les factorisations denses du benchmark LINPACK. Pour exploiter au mieux ces architectures de nombreuses solutions existent avec différents niveaux de complexité et d'apport de performance. Les scientifiques connaissent depuis longtemps la programmation par passage de message avec notamment l'interface MPI (*Message Passing Interface*) [75] pour les machines à mémoire distribuée. Mais ce n'est pas la solution optimale pour les codes en mémoire partagée puisque toutes les unités de calcul peuvent avoir accès aux données de l'autre de façon transparente. Il n'est plus nécessaire de transmettre explicitement les données, ce qui pouvait entraîner des surcoûts mémoire importants. Nous allons voir quelles sont les solutions existantes qui permettent d'exploiter ces architectures.

Bibliothèques parallèles : La solution la plus facile à mettre en œuvre est bien souvent de remplacer la bibliothèque utilisée pour les noyaux de calcul par sa version parallèle si elle existe. Dans le calcul scientifique, l'algèbre linéaire est très courante et surtout très gourmande en temps de calcul. Les scientifiques ont pour habitude d'utiliser les différentes bibliothèques d'algèbre linéaire disponibles : Basic Linear Algebra Subroutines, BLAS [67]. Plusieurs implémentations de ces fonctionnalités de base sont disponibles : refblas l'implémentation de référence de netlib [33, 34], ATLAS une implémentation open source pour tous types d'architectures [97, 98], MKL la version développée par Intel pour ses processeurs [65], Acml celle d'AMD pour exploiter au mieux les puces Opterons et Athlon [8] ou encore les GotoBLAS développés par Kazushige Goto [48]. En séquentiel, ces bibliothèques jouent sur les différents critères des processeurs pour être les plus performantes possibles et notamment sur la réduction du nombre de défauts de cache. Certaines implémentations comme les bibliothèques MKL et Acml sont tellement spécifiques à une architecture donnée que leurs performances sur les autres architectures se dégradent fortement. Il est facile pour le programmeur de remplacer ces bibliothèques par leurs versions parallèles quand elles existent. Les PBLAS [18], par exemple,

fournissent une implémentation distribuée des routines d'algèbre linéaire. Avec l'apparition des architectures SMP, on a vu apparaître des versions multi-threadées de ces bibliothèques : ACML, MKL ou GotoBLAS [31]. Cette solution simple apporte un gain sur le temps d'exécution, mais qui est loin du maximum que l'on puisse atteindre. En effet, l'utilisation de telles bibliothèques nécessite de repenser la taille des calculs demandés. La distribution des données doit se faire à *grains* plus gros puisque la taille optimale pour un seul processeur est trop petite pour fournir suffisamment de travail à un ensemble de processeurs. Cependant, il n'est pas possible de grossir le grain de calcul aussi facilement sur les codes irréguliers que sur les codes utilisant des matrices denses. Dans l'application qui nous intéresse on peut adapter la taille des blocs-colonnes du haut de l'arbre d'élimination à l'utilisation de quelques processeurs, mais il est plus difficile d'adapter la taille du grain de calcul pour utiliser des machines à plus de 4 ou 8 cœurs. On peut également avec de telles bibliothèques faire s'effondrer les performances si on ne les utilise pas correctement. En effet, si l'on crée autant de processus que de cœurs disponibles et que chacun d'entre eux fait appel à une routine d'une bibliothèque multi-threadée, les threads ainsi créés dégradent les performances les uns des autres. Avec cette technique, seul le cœur du calcul est parallélisé de cette façon, alors que tout le reste de l'application reste séquentiel. Il existe d'autres bibliothèques parallélisées pour les machines à mémoire distribuée ou partagée. Par exemple, pour les codes en C++, des versions parallèles de la STL sont développées pour gagner en performance sur les opérations coûteuses comme les opérations de tri ou de recherche. Cette solution simple n'est pas optimale pour la réalisation de codes scientifiques performants mais permet de tester le gain éventuel d'une version parallèle très facilement et rapidement.

Directives de compilation (HPF, OpenMP, UPC, . . .) : La seconde solution qui permet de paralléliser *simplement* son code est d'utiliser les techniques de parallélisation automatique. HPF [84] permet, à l'aide de PRAGMAS que l'on insère dans les codes Fortran, de paralléliser les opérations de base sur les matrices et les vecteurs. HPF découpe ainsi les données marquées par les indications du programmeur pour les distribuer en mémoire partagée et/ou distribuée. HPF est un langage qui était à la mode dans les années 90 mais qui est actuellement en voie de disparition, remplacé par des interfaces de plus haut niveau comme OpenMP [73]. Cette interface propose des ensembles de PRAGMAS qui permettent de fournir plus d'informations au compilateur sur la parallélisation du code que le simple partage de tableau. Il est possible de spécifier si les variables sont privées ou partagées lors de la parallélisation des boucles de calcul pour que le compilateur sache quelles sont les données à protéger. Cependant, au contraire d'HPF, OpenMP ne propose une parallélisation des codes que sur les machines à mémoire partagée et non plus sur les machines à mémoire distribuée. Pour les langages C et C++, les *Thread Building Blocks* d'Intel [5] vont encore plus loin en se basant sur des abstractions de C++. Il est alors possible d'exprimer la nature parallèle des objets que l'on manipule qui peuvent être plus complexes que de simples matrices. La parallélisation se fait alors à l'insu du programmeur. Le langage UPC [22] propose une parallélisation plus explicite du code sur le même schéma que la programmation avec MPI. Plusieurs flots d'exécution s'exécutent en parallèle mais cette fois au sein d'un même processus, et des éléments de syntaxe permettent de distinguer les variables privées et partagées. Ces interfaces de programmation qui proposent d'ajouter des directives par PRAGMA dans les codes Fortran et/ou C pour aider le compilateur à paralléliser le code, permettent une parallélisation rapide et relativement simple à mettre en œuvre. Cependant les codes actuels sont de plus en plus complexes et ne sont pas systématiquement de simples boucles de calculs avec des structures parallèles. Ces interfaces deviennent alors trop limitées pour la parallélisation de codes complexes, et ne permettent pas dans la plupart des cas de

tenir compte des architectures NUMA récentes, mais seulement des architectures SMP. On voit cependant apparaître de nouvelles versions des compilateurs OpenMP qui intègrent la gestion des architectures NUMA. ForestGOMP [19, 94], par exemple, se base sur la bibliothèque de threads Marcel [77] pour placer au mieux les threads de calculs et leurs données dans les boucles parallèles sur l'architecture cible. Il propose également de migrer les données sur la machine à l'aide de la bibliothèque MaMI [47, 46] si c'est avantageux par rapport au coût de la migration.

Parallélisation plus intrusive (MPC, Cilk, Thread, ...) : D'autres solutions plus complexes, mais également plus performantes peuvent être utilisées pour paralléliser son code. La solution la plus connue des scientifiques est la bibliothèque MPI qui permet d'échanger des données d'un processus à un autre et d'effectuer des communications collectives. Ceci permet aussi bien l'utilisation des machines à mémoire distribuée que des machines à mémoire partagée. Cependant les communications en mémoire partagée nécessitent l'utilisation de zones tampons intermédiaires qui entraînent un surcoût d'utilisation mémoire qui pourrait être évité. La bibliothèque MPC [81] s'appuie sur l'API de MPI pour proposer une solution qui s'adapte aussi bien aux machines multiprocesseurs qu'aux grappes de machines multiprocesseurs. Elle utilise des threads pour la partie mémoire partagée et évite si possible le surcoût des buffers de communication. D'autres approches comme Cilk [39] permettent d'illustrer le parallélisme sous forme de fonctions qui s'exécutent en parallèle et dans lesquelles on peut exprimer du parallélisme récursif. Pour obtenir les meilleures performances, il faut exprimer le plus de parallélisme possible sans que celui-ci ne devienne plus coûteux à générer qu'à exécuter. L'avantage de cette solution réside dans la facilité de parallélisation de code hétérogène dont le déroulement n'est pas suffisamment régulier pour une approche de style OpenMP.

Enfin, l'approche la plus coûteuse en terme de développement mais qui peut aboutir aux meilleures performances est celle qui utilise directement les threads avec les bibliothèques Posix, Marcel ou autres. Une telle programmation permet une programmation parallèle au "tournevis" de l'application. On peut ainsi éviter le surcoût des buffers de communications dûs aux approches par transmission de messages et facilement ordonnancer les tâches de calculs de manière statique ou dynamique. L'autre avantage de cette solution est la possibilité d'utiliser des bibliothèques comme la LibNUMA qui permettent de définir la localité des données et des threads sur la machine. Il faut toutefois faire attention à l'emploi de ces fonctions qui sont bien souvent spécifiques à un système d'exploitation. L'inconvénient de cette approche est son aspect très intrusif et contraignant vis-à-vis du code. Tout développement dans le code doit être pensé dans ce sens et n'est pas forcément aussi facilement intégré dans une version parallèle qu'avec une approche par directives de compilation.

De nombreuses approches sont disponibles pour exploiter les nouvelles architectures multiprocesseurs avec différents niveaux de difficulté d'intégration dans les applications scientifiques. Mais on peut constater qu'actuellement aucune solution de parallélisation *"automatique"* ne permet de tirer pleinement parti des architectures NUMA et de leur structure attribuant des zones mémoire privilégiées à chaque cœur. Les résultats obtenus par de tels procédés de parallélisation ne sont actuellement pas toujours satisfaisants en raison de la gestion inadaptée de la mémoire. Les développements actuels tendent à intégrer les facteurs NUMA des architectures dans ces directives de compilation comme ForestGOMP [19, 94]. Les bibliothèques MPI cherchent également à rapprocher les threads de communication des entrées/sorties et les implémentations des BLAS essaient de placer, voire déplacer, la mémoire utilisée près des threads de calculs. Nous allons maintenant voir quelles sont les solutions adoptées parmi ces proposi-

tions par les bibliothèques de résolution de systèmes linéaires et comment elles s'adaptent aux nouvelles architectures NUMA.

1.4 Comment les solveurs répondent à cette problématique

Au cours des dernières années, les solveurs directs de systèmes linéaires creux ont fait d'énormes progrès [10, 52, 61, 68]. Il est désormais possible de résoudre efficacement et dans un laps de temps raisonnable des problèmes 3D de plusieurs millions d'inconnues. La multiplication des super-calculateurs basés sur des architectures multi-cœurs SMP (Symmetric Multi-Processor) a conduit les solveurs à proposer des implémentations mieux adaptées à ces nouvelles architectures qui ont permis la résolution de ces problèmes. La plupart ont choisi dans un premier temps de proposer des solutions basées sur MPI à partir de leur version séquentielle pour l'exploitation des clusters. D'autres ont développé des versions adaptées à l'utilisation de machines multi-cœurs SMP. Enfin, une partie de ces solveurs proposent des algorithmes qui permettent de combiner ces solutions. Nous allons présenter les principaux solveurs existants actuellement qui sont : MUMPS, Pardiso, PaStiX, SuperLU et WSMP et les différentes méthodes qu'ils utilisent pour exploiter les clusters et résoudre de grands problèmes.

Le solveur MUMPS est développé par l'équipe de Patrick Amestoy à Toulouse et Jean-Yves L'Excellent à Lyon. Il est basé sur une méthode multifrontale pour la factorisation avec un ordonnancement dynamique des calculs [10, 11]. Ses avantages sont la disponibilité de différentes solutions de prétraitements numériques de la matrice pour améliorer la stabilité de la factorisation numérique et une version out-of-core performante. Il s'agit d'une version dans laquelle on peut limiter au strict minimum la consommation mémoire en stockant la majeure partie des données sur les disques durs. Cette solution offre la possibilité de résoudre des problèmes de taille plus importante sur des clusters où il y a peu de mémoire disponible. Il dispose aussi de la possibilité de faire du pivotage numérique qui le distingue des autres solveurs. La dernière version sortie est une version distribuée MPI codée en Fortran. Ce solveur ne dispose malheureusement pas encore de version exploitant les architectures multi-cœurs, mais propose l'utilisation de bibliothèques de BLAS multithreadées comme on l'a vu dans la section 1.3.3 page 18. Les auteurs travaillent actuellement sur le développement d'une version OpenMP pour étudier le gain possible avec une telle version au niveau mémoire et temps de calculs.

Le second solveur Pardiso [91] d'Olaf Schenk et Klaus Gärtner est développé uniquement pour les machines à mémoire partagée et il n'existe pas de version MPI. L'algorithme utilisé est une méthode supernodale avec un ordonnancement mixant les approches left et right-looking. Il est également développé en Fortran et la version en mémoire partagée utilise les directives de compilation OpenMP. De même que MUMPS, il propose également des techniques de permutations non-symétriques basées sur les valeurs des coefficients pour mieux conditionner la matrice.

Jim Demmel et Sherry Li proposent plusieurs versions de leur solveur SuperLU pour s'adapter aux machines séquentielles, à mémoire partagée ou à mémoire distribuée [30, 50]. Le solveur utilise un algorithme similaire à celui de Pardiso : méthode supernodale avec approche hybride left et right-looking. SuperLU_dist est la version distribuée en MPI, et SuperLU_MT la version pour machine à mémoire partagée. Cette version a été développée avant l'apparition des bibliothèques de threads et d'OpenMP et utilise la mémoire partagée entre processus. Plusieurs solutions ont donc été développées pour s'adapter aux différentes architectures et systèmes d'exploitations. Avec l'apparition de nouvelles techniques, les auteurs ont développé de nouvelles versions et SuperLU_MT existe désormais en version OpenMP et en version mul-

tithreadée (Thread POSIX) pour une meilleure exploitation des architectures récentes et pour ne plus avoir d'implémentations spécifiques à chaque système d'exploitation. Cependant aucun travail ne porte actuellement sur le couplage des versions SuperLU_Dist et SuperLU_MT pour exploiter les clusters modernes. La seule solution existante est d'intégrer une bibliothèque multithreadée dans la version distribuée comme pour le solveur MUMPS. Comme nous l'avons vu dans la section 1.3.3 page 18 l'utilisation de ces bibliothèques nécessite de pouvoir changer la taille des blocs (ou blocs-colonnes) de calculs pour adapter le grain de calcul au nombre de threads utilisés et de faire en sorte que le nombre de threads utilisés par la bibliothèque sous-jacente multiplié par le nombre de processus sur un nœud ne dépasse pas le nombre de cœurs disponibles pour ne pas surcharger la machine et perdre l'efficacité des bibliothèques BLAS.

Enfin, les solveurs PaStiX de Pierre Ramet et Pascal Hénon et WSMP d'Anshul Gupta, George Karypis et Vipin Kumar proposent des implémentations hybrides MPI/Threads Posix [53, 62]. Ils sont tous les deux codés en C et reposent respectivement sur une méthode supernodale right-looking et sur une méthode multifrontale et proposent tous les deux quatre modes de fonctionnement : séquentiel, mémoire partagée, mémoire distribuée ou méthode hybride mixant mémoire partagée et mémoire distribuée. Pour le solveur PaStiX, il s'agit d'un seul solveur que l'on compile dans la version souhaitée à l'aide d'options de compilation. Au contraire pour le solveur WSMP, il s'agit de deux versions : une en mémoire partagée, l'autre en mémoire distribuée qui peuvent toutes les deux être utilisées en séquentiel ou être couplées pour obtenir la version hybride. L'utilisation de la bibliothèque MPI pour les communications en mémoire distribuée et des threads Posix en mémoire partagée permet à ces deux solveurs de réduire significativement le surcoût mémoire dû aux buffers de communications et au nombre de blocs de contributions dans la méthode multi-frontale qui sont un frein à l'utilisation de méthodes directes sur de grands problèmes 3D. Ces deux solveurs sont donc ceux qui doivent permettre d'exploiter le plus efficacement possible les clusters de nœuds multi-cœurs. On constate finalement qu'il n'existe pas actuellement de solveurs qui prennent en compte la topologie hiérarchique des nouvelles architectures NUMA.

1.5 Choix d'un solveur de référence pour l'étude

Parmi les solveurs cités dans la section précédente, nous avons choisi pour les travaux de cette thèse de nous baser sur le solveur PaStiX. Ce solveur a été développé à l'origine à Bordeaux pour des codes d'électromagnétisme du CEA Cesta. Cette bibliothèque codée en C a été développée dans un premier temps à l'aide de MPI pour la version parallèle. Puis, pour gagner en consommation mémoire et réduire de manière importante le surcoût des buffers de communication au sein des nœuds où la version MPI était utilisée, une version hybride MPI/Thread Posix a été implémentée. Le choix de ce solveur, outre le fait qu'il était développé localement, était basé sur plusieurs critères. Nous souhaitions pouvoir interagir finement avec le solveur choisi en utilisant la bibliothèque de threads. Ceci a donc orienté notre choix vers un des solveurs développés en C, étant donné qu'il est très difficile de manipuler des threads en Fortran sans utiliser des directives de compilations comme OpenMP. Les solveurs MUMPS et Pardiso étaient exclus car écrits en Fortran, et pour MUMPS il fallait également adapter le solveur aux machines multi-cœurs. Parmi les trois solveurs présentés restants, nous avons choisi d'exclure SuperLU_MT puisqu'il n'existe pas de version MPI du solveur pour intégrer les communications dans l'ordonnancement des calculs. Effectivement, un des objectifs de cette thèse est également d'étudier les problématiques de couplage d'ordonnancement dynamique avec la distribution des données. Chaque version parallèle de SuperLU ne permettait pas d'étudier un des deux problèmes.

Les deux solveurs restants pour lesquels nous trouvions intéressant d'étudier leur comportement sur architectures NUMA étaient par conséquent WSMP et PaStiX. Le premier, WSMP, utilise une méthode multifrontale avec une distribution 2D par blocs dynamique de la matrice sur les nœuds, tandis que le second, PaStiX, utilise une méthode supernodale avec une distribution statique des données sur les nœuds. Puisque la licence IBM de WSMP ne donne pas un accès libre aux sources du solveur, nous avons choisi le solveur PaStiX sous licence Cecill-C pour une meilleure diffusion des résultats. De plus, l'approche supernodale/right-looking de PaStiX est déjà bien découpée en tâches de calculs régulières : factoriser le bloc (ou bloc-colonne) et ajouter immédiatement après les contributions à leur emplacement. L'ordonnancement statique performant de PaStiX fournit une distribution des données sur les processus MPI efficace à partir de laquelle on peut travailler pour réordonnancer les calculs au sein de chaque processus sans avoir à recalculer une distribution. Nous allons voir dans la section suivante les bases du fonctionnement de cette distribution statique et l'algorithme de factorisation numérique utilisé dans le solveur PaStiX.

1.6 Description de l'algorithme utilisé par PaStiX

Le solveur PaStiX contrairement à d'autres solveurs distribue les données statiquement une seule fois lors du prétraitement. Les données ne sont pas redistribuées dynamiquement lors de la factorisation numérique comme peut le faire MUMPS par exemple. Pour obtenir de bonnes performances, l'étape de distribution des données et la factorisation numérique doivent être fortement couplées.

1.6.1 Distribution et ordonnancement

Les bonnes performances de l'algorithme de factorisation du solveur PaStiX se basent sur un bon partitionnement et une bonne distribution des données sur les nœuds de calcul. Ceci se fait sur la structure par blocs obtenue avec la factorisation symbolique par bloc. Cette étape, issue des travaux de thèse de Pascal Hénon [59], calcule de manière statique une *régulation équilibrée* pour le solveur en se basant sur l'équilibrage de charge des processeurs et les contraintes de précédence entre les différents blocs de la matrice. Les règles de dépendance entre les calculs sont données par la structure de l'arbre d'élimination par blocs décrite dans la section 1.2.2 page 10 : le calcul d'un bloc-colonne ne peut se faire tant que les contributions que lui apportent ses descendants dans l'arbre ne sont pas calculées. Inversement le calcul d'un bloc-colonne apporte une contribution aux blocs-colonnes qui sont ses ascendants. Le parallélisme ainsi induit par le creux est d'autant plus important que l'arbre d'élimination par blocs est large. La régulation du coût de calcul et les choix de placement sur les processeurs se basent sur une modélisation fine des routines BLAS utilisées et des communications. Ces modèles de coûts ont été développés lors de la thèse de Pierre Ramet [82] et ils sont obtenus expérimentalement en calculant le temps moyen d'exécution de ces routines sur un ensemble de données de tailles différentes sur l'architecture cible. On effectue ensuite une régression linéaire sur les temps observés pour décrire le comportement des routines à l'aide de polynômes. Le coût des communications est calculé par exécution de benchmarks d'échange de messages qui déterminent la latence et le débit du réseau. L'intégration de l'implémentation hybride MPI/Thread du solveur a nécessité de différencier les coûts de communication intra et inter-nœuds puisque le coût des transferts en mémoire partagée est nul dans le cas d'une application multi-threadée.

Le but de cette étape de distribution des données est d'exploiter, grâce à ces modèles de

coûts, les différents niveaux de parallélisme existant dans les calculs, à savoir :

1. le parallélisme à gros grain, induit par l'indépendance des calculs entre les sous-arbres d'un même nœud de l'arbre d'élimination. Il s'agit du parallélisme induit par le creux.
2. le parallélisme à grain moyen, dû à la possibilité de raffiner la partition des inconnues en découpant un nœud de l'arbre d'élimination et en le distribuant sur plusieurs processeurs. C'est le parallélisme induit par une factorisation sur des blocs denses. On parlera aussi de parallélisme au niveau des nœuds.
3. le parallélisme à grain fin, ou micro-parallélisme. Celui-ci est obtenu en tirant parti du parallélisme interne d'un processeur lors du calcul sur les blocs (utilisation optimale de l'effet pipeline des processeurs superscalaires). Ce dernier niveau de parallélisme est fondamental pour obtenir de bonnes performances. Il repose, modulo *une bonne taille de blocs*, essentiellement sur l'utilisation des primitives BLAS3 [54, 55, 78, 87, 88, 89].

Cette phase se base sur l'arbre d'élimination pour distribuer de manière équilibrée les blocs-colonnes indépendants qui sont le plus bas dans l'arbre. Les sous-blocs des niveaux hauts de l'arbre sont souvent plus gros et plus denses et représentent une plus grande quantité de calculs. Il est donc nécessaire de les couper à une taille optimale pour le calcul sur un processeur pour en extraire plus de parallélisme. Ils sont ensuite distribués, par exemple de manière bloc cyclique, pour utiliser le parallélisme contenu dans les calculs denses. On obtient alors une bonne régulation des calculs de façon statique en simulant la factorisation numérique à l'aide des modèles de coûts [38, 44, 85, 88].

Cette technique peut être utilisée selon un schéma de distribution unidimensionnelle (1D) des blocs-colonnes, ce qui permet d'exploiter le parallélisme lié à l'indépendance des calculs existant entre les blocs-colonnes, ou selon un schéma bidimensionnel (2D) des blocs-colonnes dans lequel on exploite, en plus, le parallélisme lié à l'indépendance des calculs existant entre les blocs élémentaires de cette distribution comme dans la factorisation de matrice dense. Une distribution 2D permet une meilleure extensibilité du solveur parallèle [88, 92] mais génère un surcoût de communication par rapport à une distribution 1D.

1.6.2 Factorisation numérique

L'étape de distribution et d'ordonnancement statique de PaStiX donne lieu à un ordonnancement statique des calculs qui doit être suivi *à la lettre* si on ne veut pas perturber la régulation calculée. Pour cela, cette étape définit un tableau ordonné à double entrée, $Task(t, i)$, dans lequel la factorisation de chaque bloc-colonne est attribuée à un thread t. De plus, pour ne pas perturber les accès aux niveaux de cache par les routines BLAS, le solveur PaStiX fixe chaque thread de calcul sur un cœur de calcul différent grâce à différentes fonctions suivant les systèmes d'exploitation : `bindprocessor`, `bind_to_cpu_id` ou `sched_setaffinity`. Les threads, une fois associés à un cœur de calcul, parcourent lors de la factorisation numérique le tableau pour effectuer les calculs associés aux NT_t tâches qui leur sont attribuées comme le montre l'algorithme 2 :

- recevoir les données associées au bloc-colonne et les sommer,
- attendre le calcul des contributions locales,
- factoriser le bloc diagonal,
- résoudre les systèmes triangulaires sur les blocs extra-diagonaux,
- calculer et envoyer les contributions.

La première phase du calcul de chaque tâche est la réception des données (lignes 3 à 7). Lors de cette boucle de calcul, on attend toutes les contributions nécessaires à la factorisation

du bloc-colonne k, mais pour améliorer la réactivité des communications chaque thread attend en réalité une contribution qui lui est destinée mais qui n'est pas nécessairement destinée au bloc-colonne k. Les ajouts des contributions reçues peuvent ainsi être gérés avant le calcul de la tâche destinataire. Une fois que toutes les contributions distantes sont reçues, le thread attend que toutes les contributions qui sont calculées par d'autres threads internes au même processus soient également ajoutées (lignes 8 à 10).

Cette version de l'algorithme 2 hybride MPI/Threads présentée ici n'est pas tout à fait la version existante du solveur au début de ces travaux de thèse. En effet, une des premières modifications effectuées sur ce solveur a été de remplacer l'attente active au sein de la boucle de calcul par des signaux que l'on retrouve aux lignes 9 et 19. L'ancienne version bouclait en attente active sur le test de NC_i, mais comme elle utilisait un seul thread de calcul par cœur présent sur la machine, cette attente ne perturbait pas ou peu l'exécution du solveur suivant les architectures. Cependant comme nous souhaitons introduire un ordonnancement dynamique et utiliser un ou plusieurs threads supplémentaires pour la gestion des entrées/sorties, chaque thread doit pouvoir réagir au plus vite et ne pas rester bloqué dans une boucle d'attente. La solution à ce problème a été d'utiliser les routines de conditions : `pthread_cond_(time)wait` et `pthread_cond_signal` pour que les systèmes d'exploitation puissent exécuter les threads en attente. Cette modification ne perturbe pas le comportement de l'algorithme d'origine et peut même parfois l'améliorer. Elle a donc été mise en place par défaut dans toutes les versions du solveur PaStiX.

La suite du déroulement de l'algorithme de factorisation est identique à l'algorithme séquentiel pour le calcul de la tâche en elle-même : factorisation du bloc diagonal et résolution des systèmes triangulaires pour les blocs extra-diagonaux à la ligne 11. Puis, l'ajout des contributions se fait en deux étapes, le calcul (lignes 12 à 25) et l'envoi (lignes 26 à 30). La méthode utilisée est une approche "fan-in/right-looking" donc les contributions distantes sont agrégées dans un bloc-colonne temporaire avant d'être envoyées, il s'agit de la ligne 22 de l'algorithme. La version hybride MPI/Thread permet d'éviter de stocker ces blocs-colonnes temporaires en mémoire partagée [62], puisqu'on peut ajouter immédiatement les contributions dans les blocs-colonnes destinataires (lignes 15 et 16). On bascule alors vers une approche "fan-out". On mixe automatiquement les deux approches en jouant sur la distribution MPI/Threads.

Notations : Pour chaque bloc-colonne k, $1 \leq k \leq N$,
- $Proc_k$ et $Thread_k$ sont respectivement le processus et le thread de ce processus désigné par l'ordonnancement statique pour factoriser le bloc-colonne k ;
- A_k est le bloc-colonne k, par symétrie, U_k est le bloc-ligne k de U ;
- b_k est le nombre de blocs extra-diagonaux dans le bloc-colonne k ;
- C_k est le bloc-colonne de contributions calculées sur le processus p pour le bloc-colonne k ;
- NCD_k est le nombre de contributions distantes que le bloc-colonne k doit recevoir ;
- NC_k est le nombre de contributions totales que le bloc-colonne k doit assembler : locales et distantes ;
- NT_t est le nombre de tâches attribuées au thread t ;
- $Task(t, l)$ est le $l^{ème}$ bloc-colonne que doit factoriser le thread t.

Pour simplifier les notations, on considère que $A_{(i),(j)} \sim A_{i,j}$.

Algorithme 2 Factorisation parallèle par blocs-colonnes sur le thread t du processus p

```
1:  Pour l = 1 to NT_t Faire
2:      k = Task(t, l)
3:      Tant que NCD_k > 0 Faire
4:          Attendre une contribution destinée à soi-même
5:          Ajouter la contribution C_r reçue au bloc-colonne r        /* Réception */
6:          NCD_r − −; NC_r − −;
7:      Fin de Tant que
8:      Tant que NC_k > 0 Faire
9:          Attendre signal pour k
10:     Fin de Tant que
11:     Calcul de la tâche l : factorisation de A_{k,k} et résolution de L_k et U_k   /* Calcul */
12:     Pour j = 1 à b_k Faire
13:         Pour i = j à b_k Faire
14:             Si A_{i,j} est local alors
15:                 A_{i,j} = A_{i,j} − L_{i,k}.U_{k,j}
16:                 A_{j,i} = A_{j,i} − L_{j,k}.U_{k,i}
17:                 NC_j − −
18:                 Si NC_j = 0 alors
19:                     Envoyer signal au bloc-colonne j
20:                 Fin de Si
21:             Sinon
22:                 C_i = C_i − L_{i,k}.U_{k,j}
23:             Fin de Si
24:         Fin de Pour
25:     Fin de Pour
26:     Pour i = 1 à b_k Faire
27:         Si A_i est non local et C_i complet alors
28:             Envoi de la contribution C_i au processus Proc_i pour Thread_i   /* Envoi */
29:         Fin de Si
30:     Fin de Pour
31: Fin de Pour
```

1.7 Discussion

Nous avons vu au cours de ce chapitre que la résolution de systèmes linéaires est une étape fréquente et coûteuse des codes de calculs récents aussi bien du point de vue du temps que des ressources mémoire nécessaires. De nombreuses solutions ont été mises au point pour repousser ces limites comme les méthodes hybrides ou multi-grilles. Mais les méthodes directes restent la solution privilégiée pour certains problèmes pour leur précision ou comme préconditionneur de méthodes itératives. D'un autre côté, les architectures des clusters ont également beaucoup évolué permettant aussi de repousser les limites sur les tailles de problèmes calculables en un temps raisonnable. Mais de nouveaux paramètres apparaissent dans ces architectures : les accès mémoire non uniformes induits par les topologies très hiérarchiques des nouveaux clusters. Les méthodes directes ont déjà pris du retard dans l'exploitation des clusters de machines multi-cœurs SMP puisqu'il n'existe que deux solveurs capables d'exploiter pleinement ces architectures, les autres ne proposent que des solutions adaptées soit aux machines à mémoire

partagée, soit à mémoire distribuée. Pour utiliser la combinaison de ces architectures, la seule solution mise à disposition des utilisateurs est d'adapter la taille des blocs de calcul pour utiliser des bibliothèques BLAS multi-threadées.

Nous proposons dans cette thèse de nous baser sur le solveur PaStiX exploitant déjà les clusters de machines SMP pour développer une solution efficace d'ordonnancement dynamique d'arbres de tâches sur architectures NUMA. En effet les modèles de coûts utilisés dans le solveur ne permettent pas de prendre en compte les accès dissymétriques à la mémoire de ces nouvelles architectures. Un modèle de coût complet et fonctionnel sur de telles machines serait très difficile à calculer, puisqu'il dépendrait de la localité des données et des calculs mais également de la contention engendrée sur les bus de communications par la distribution des données sur la machine. Il est donc impossible de réaliser une prédiction statique réaliste des calculs avec tous les paramètres qui entrent en compte. Pour remédier à cette prédiction faussée de l'ordonnancement statique, nous souhaitons mettre en place un ordonnancement dynamique que nous évaluerons au sein du solveur PaStiX sur des clusters de machines NUMA.

La méthode de résolution utilisée dans le solveur PaStiX fait que le graphe des dépendances entre les tâches de calculs est équivalent à l'arbre d'élimination de la matrice. Le problème consiste à trouver comment ordonnancer un arbre de tâches sur une machine NUMA. Nous étudierons dans un premier temps comment adapter les structures et les méthodes d'allocations des codes de calculs à ces architectures. En effet, de simples modifications peuvent apporter des gains importants en performance si les accès mémoire sont optimisés. Puis, comme l'ordonnancement du solveur va être modifié, nous devons également revoir le schéma de communications de celui-ci. Enfin nous verrons comment nous avons choisi d'ordonnancer efficacement l'arbre de tâches du solveur sur les architectures NUMA.

Chapitre 2

Optimisation de l'utilisation mémoire pour architectures hiérarchiques

Sommaire

2.1 Accès mémoire non-uniformes **30**
2.2 Problèmes de contention **34**
2.3 Améliorations de la gestion mémoire **37**
2.4 Présentation des cas tests et des clusters utilisés pour l'étude . . . **39**
2.5 Evaluation sur le solveur PaStiX **42**

Les architectures multiprocesseurs modernes sont principalement basées sur des systèmes à mémoire partagée avec un comportement NUMA. Ces ordinateurs sont composés de plusieurs processeurs, eux-mêmes composés de plusieurs cœurs. Chacun d'eux est associé à une unité mémoire et ils sont interconnectés par un système de cohérence de cache leur donnant accès à l'intégralité de la mémoire de manière transparente. Ce type d'architecture implique une structure très hiérarchisée dont les coûts d'accès à la mémoire varient grandement d'une unité mémoire à une autre comme le montrent Antony et al. dans [12] sur diverses architectures. De plus, la bande passante est également variable à cause de l'entrelacement des accès qui peuvent se perturber entre eux. Ces architectures imposent de prendre en compte le placement des données lors de leur allocation et de leur utilisation pour réduire au maximum la distance unité de calcul / mémoire ainsi que les échanges de données entre les chipsets. Nous montrerons dans cette partie l'impact que peut avoir la localité des données sur différentes architectures NUMA et SMP (cf respectivement les sections 1.3.1 et 1.3.2 page 14 et 16). Dans un premier temps, on analysera l'impact de la localité mémoire pour des calculs isolés. Puis dans un deuxième temps, on analysera les impacts sur une machine chargée. Nous présenterons ensuite une solution simple permettant d'améliorer grandement les performances sur architecture NUMA des programmes qui utilisent intensivement la mémoire. Les résultats seront validés par un ensemble de tests réalisés sur le solveur PaStiX. L'ensemble des calculs de cette section ont été réalisés sur les machines SMP : Decryphton (figure 1.5 page 15) et NUMA : Borderline et Hagrid (figures 1.6(a) et 1.6(b) page 17).

2.1 Accès mémoire non-uniformes

La première étude que nous souhaitons faire porte sur le comportement d'applications simples avec différents placements des données et d'un seul thread de calcul. Pour placer les threads de calcul à l'endroit voulu sur l'architecture, les systèmes d'exploitation courants fournissent différentes API plus ou moins dédiées aux architectures NUMA. Ces API permettent au programmeur d'avoir un contrôle sur le placement des données et des processus de calculs. AIX fournit par exemple la routine `bindprocessor` pour fixer un thread sur un cœur de la machine. Des équivalents sont disponibles sur les autres systèmes d'exploitation ou dans d'autres bibliothèques de threads : `sched_setaffinity` sous Linux, `thread_policy_set` sous Mac OS X, `bind_to_cpu_id` sur True64 (Alpha Compaq) ou bien encore `marcel_apply_vpset` pour la bibliothèque MARCEL. Sous Linux, la bibliothèque *LibNUMA* fournit un ensemble de fonctions pour modifier les politiques d'allocation mémoire du système. Il est alors possible de forcer l'allocation des données sur un nœud spécifique ou d'utiliser des techniques de répartition comme le *round-robin* pour placer aléatoirement et uniformément les données allouées sur tous les nœuds. La stratégie par défaut étant d'allouer les données au plus proche du nœud qui aura besoin de la zone mémoire en premier. Ce principe est celui que l'on retrouve par défaut sur la majorité des systèmes d'exploitation car il est souvent celui qui apporte le plus de performance en moyenne.

Dans un premier temps, nous avons mis en avant les effets NUMA des différentes architectures auxquelles nous avions accès grâce à ces API. Les effets ont été étudiés sur deux types de fonctions BLAS couramment utilisées dans les codes de calculs scientifiques et notamment dans les solveurs directs. La fonction `dAXPY` réalise l'opération $\alpha * X + Y$ avec X et Y deux vecteurs de réels de taille n et α une constante réelle. Cette opération demande un débit mémoire important par rapport au nombre de calculs effectués puisqu'il faut transférer $O(n)$ réels double précision pour seulement $O(n)$ produits et $0(n)$ sommes. La seconde fonction utilisée est le produit de matrice `dGEMM` qui réalise l'opération $\alpha * A * B + \beta * C$ avec A, B et C trois matrices de taille respective $m \times n$, $n \times p$ et $m \times p$ et α et β deux constantes réelles. Avec $n \sim m \sim p$, on doit transférer pour cette opération $O(n^2)$ réels double précision en mémoire pour réaliser $O(n^3)$ produits et $O(n^2)$ sommes. Le rapport du nombre de calculs sur la quantité de données nécessaire est plus important et le temps de calculs devient prépondérant sur le temps du transfert mémoire grâce à une meilleure réutilisation du cache mémoire.

Les programmes réalisés mesurent le temps moyen d'exécution de ces fonctions sur une boucle de N itérations d'appels à ces routines. Pour nos expériences nous avons choisi de fixer $N = 1000$ et d'utiliser des matrices et des vecteurs de taille $n = 128$ remplis avec des réels double précision aléatoires. De plus pour ne pas fausser les résultats avec une réutilisation du cache entre chaque appel, chaque appel se fait sur des matrices et des vecteurs différents. Pour effectuer le placement des données, deux threads sont lancés et sont chacun fixés sur un cœur de calcul. Le premier alloue et initialise les données sur le banc mémoire le plus proche de son positionnement, le second fait les appels aux routines BLAS sur ces données allouées par l'autre thread. Le placement des threads se fait par les fonctions citées ci-dessus et le placement des données est laissé au système pour utiliser la politique d'allocation par défaut, c'est-à-dire au plus proche du premier thread qui fait un défaut de cache sur la zone mémoire demandée.

Les tableaux 2.1, 2.2 et 2.3 montrent les facteurs NUMA obtenus sur deux de nos architectures à 16 cœurs. La première, Decrypthon, qui est une machine SMP, donne des résultats relativement stables quelque soit le placement des données et du thread de calcul. Malgré l'effort pour uniformiser les accès mémoire de la machine du constructeur, on trouve une différence allant jusqu'à 7% dans certaines configurations sur l'utilisation des opérations BLAS vecteurs/vecteurs,

TAB. 2.1 – Influence du placement des données sur la machine Borderline. Les temps présentés sont les temps relatifs par rapport au meilleur placement théorique : les données et le thread de calcul sont associés au même cœur.

(a) Fonction dAXPY.

		Computational thread location							
		0	1	2	3	4	5	6	7
Data location	0	**1.00**	1.34	1.31	*1.57*	**1.00**	1.34	1.31	*1.57*
	1	1.33	**1.00**	*1.57*	1.31	1.33	**1.00**	*1.57*	1.31
	2	1.28	*1.57*	**1.00**	1.33	1.29	*1.57*	**1.00**	1.32
	3	*1.56*	1.32	1.34	**1.00**	*1.56*	1.31	1.33	**0.99**
	4	**1.00**	1.34	1.31	*1.57*	**1.00**	1.34	1.30	*1.58*
	5	1.33	**1.00**	*1.58*	1.30	1.33	**1.00**	*1.57*	1.30
	6	1.29	*1.57*	**1.00**	1.33	1.29	*1.57*	**1.00**	1.32
	7	*1.57*	1.32	1.33	**1.00**	*1.57*	1.32	1.35	**1.00**

(b) Fonction dGEMM.

		Computational thread location							
		0	1	2	3	4	5	6	7
Data location	0	**1.00**	1.04	1.04	*1.07*	**1.00**	1.04	1.04	*1.07*
	1	1.04	**1.00**	*1.07*	1.04	1.04	**1.00**	*1.08*	1.04
	2	1.04	*1.07*	**1.00**	1.04	1.04	*1.07*	**1.00**	1.04
	3	*1.08*	1.04	1.04	**1.00**	*1.07*	1.04	1.05	**1.00**
	4	**1.00**	1.04	1.04	*1.07*	**1.00**	1.04	1.04	*1.07*
	5	1.05	**1.00**	*1.08*	1.04	1.05	**1.00**	*1.08*	1.04
	6	1.04	*1.07*	**1.00**	1.05	1.04	*1.07*	**1.00**	1.04
	7	*1.08*	1.04	1.05	**1.00**	*1.08*	1.04	1.05	**1.00**

Tab. 2.2 – Influence du placement des données sur la machine Hagrid. Les temps présentés sont les temps relatifs par rapport au meilleur placement théorique : les données et le thread de calcul sont associés au même cœur.

(a) Fonction dAXPY.

Placement des données	Placement du thread de calcul															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	**1.00**	**1.00**	1.16	1.16	1.11	1.11	1.11	1.11	1.17	1.17	1.27	1.27	1.36	1.37	1.44	1.43
1	**1.00**	**1.00**	1.19	1.16	1.11	1.08	1.11	1.11	1.17	1.17	1.27	1.27	1.37	1.36	1.43	1.43
2	1.19	1.19	**1.00**	**1.00**	1.13	1.12	1.11	1.11	1.17	1.27	1.16	1.15	1.49	1.44	1.30	1.34
3	1.18	1.18	**1.00**	**1.00**	1.12	1.12	1.11	1.11	1.28	1.28	1.15	1.15	1.44	1.44	1.33	1.33
4	1.71	1.73	1.77	1.77	**1.00**	**1.00**	1.30	1.30	1.40	1.39	1.47	1.47	1.74	1.75	1.70	1.70
5	1.73	1.73	1.77	1.77	**1.00**	**1.00**	1.29	1.30	1.40	1.40	1.47	1.47	1.75	1.75	1.70	1.71
6	1.79	1.79	1.71	1.71	1.29	1.30	**1.00**	**1.00**	1.49	1.49	1.39	1.38	1.71	1.72	1.70	1.72
7	1.78	1.78	1.70	1.70	1.30	1.30	**1.00**	**1.00**	1.47	1.48	1.18	1.39	1.70	1.70	1.70	1.70
8	1.84	1.84	2.05	2.05	1.39	1.39	1.64	1.64	**1.00**	**1.00**	1.43	1.42	1.62	1.58	1.47	1.47
9	1.86	1.85	2.07	2.05	1.40	1.40	1.66	1.65	**1.00**	**1.00**	1.44	1.43	1.59	1.60	1.49	1.48
10	2.08	2.09	1.84	1.84	1.68	1.69	1.40	1.38	1.44	1.44	**1.00**	**0.99**	1.49	1.50	1.58	1.58
11	2.03	2.10	1.84	1.84	1.68	1.69	1.40	1.39	1.44	1.46	**1.00**	**1.00**	1.49	1.50	1.59	1.59
12	1.35	1.34	1.31	1.31	1.06	1.06	1.03	1.03	0.96	0.96	0.97	0.97	**1.00**	**0.99**	1.05	1.06
13	1.68	1.68	1.64	1.63	1.31	1.32	1.28	1.28	1.19	1.21	1.21	1.20	**1.01**	**1.00**	1.32	1.32
14	1.67	1.67	1.66	1.67	1.31	1.31	1.31	1.31	1.22	1.21	1.22	1.21	1.33	1.33	**1.00**	**1.00**
15	1.68	1.67	1.67	1.67	1.30	1.30	1.31	1.31	1.21	1.22	1.21	1.21	1.32	1.33	**1.00**	**1.00**

(b) Fonction dGEMM.

Placement des données	Placement du thread de calcul															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	**1.00**	0.90	0.90	0.90	0.88	0.89	0.89	0.89	0.89	0.91	0.91	0.92	0.92	0.93	0.93	0.94
1	1.03	**1.00**	0.97	0.97	0.96	0.97	0.96	0.96	0.97	0.98	1.00	0.99	1.00	1.00	1.02	1.02
2	0.94	0.98	**1.00**	0.87	0.86	0.88	0.86	0.86	0.89	0.90	0.87	0.87	0.92	0.92	0.90	0.90
3	1.01	1.05	**1.00**	**1.00**	1.00	1.00	0.99	1.00	1.03	1.03	1.00	1.00	1.06	1.06	1.03	1.03
4	1.10	1.04	1.04	1.02	**1.00**	0.95	0.95	0.96	0.98	0.98	0.99	0.99	1.01	1.03	1.01	1.01
5	1.10	1.17	1.07	1.08	**1.00**	**1.00**	1.01	1.02	1.03	1.03	1.04	1.04	1.08	1.08	1.07	1.07
6	1.19	1.13	1.08	1.08	1.01	1.02	**1.00**	**1.00**	1.04	1.04	1.03	1.03	1.07	1.07	1.08	1.07
7	1.15	1.09	1.07	1.08	1.01	1.01	**1.00**	**1.00**	1.03	1.04	1.03	1.03	1.07	1.07	1.06	1.07
8	1.12	1.11	1.14	1.12	1.02	1.02	1.06	1.07	**1.00**	**1.00**	1.03	1.04	1.08	1.06	1.04	1.05
9	1.23	1.21	1.13	1.13	1.02	1.04	1.07	1.07	**1.00**	**1.00**	1.03	1.04	1.07	1.06	1.04	1.05
10	1.18	1.16	1.08	1.09	1.07	1.08	1.02	1.03	1.03	1.03	**1.00**	**1.00**	1.04	1.05	1.06	1.07
11	1.26	1.25	1.25	1.08	1.15	1.07	1.11	1.05	1.04	1.04	**1.00**	**1.00**	1.05	1.05	1.07	1.07
12	0.99	0.94	0.93	0.93	0.89	0.89	0.88	0.89	0.90	0.88	0.90	0.87	**1.00**	0.87	0.89	0.89
13	1.11	1.11	1.07	1.07	1.02	1.02	1.02	1.02	1.01	1.02	1.00	1.02	**1.00**	**1.00**	1.03	1.03
14	1.18	1.11	1.07	1.07	1.01	1.03	1.02	1.02	1.00	1.01	1.01	1.02	1.03	1.04	**1.00**	**1.00**
15	1.11	1.09	1.08	1.07	1.02	1.03	1.01	1.03	1.00	1.01	1.00	1.01	1.02	1.03	**1.01**	**1.00**

Tab. 2.3 – Influence du placement des données sur la machine Decrypthon. Les temps présentés sont les temps relatifs par rapport au meilleur placement théorique : les données et le thread de calcul sont associés au même cœur.

(a) Fonction dAXPY.

		Placement du thread de calcul															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Placement des données	0	**1.00**	1.00	1.02	1.01	1.06	1.06	1.07	1.07	1.06	1.06	1.07	1.07	1.07	1.07	1.06	1.06
	1	1.00	**1.00**	1.03	1.02	1.07	1.06	1.07	1.07	1.07	1.07	1.08	1.07	1.08	1.07	1.06	1.06
	2	0.96	0.97	**1.00**	1.01	1.04	1.04	1.05	1.05	1.04	1.04	1.05	1.05	1.05	1.05	1.04	1.03
	3	0.96	0.96	1.00	**1.00**	1.03	1.03	1.04	1.04	1.04	1.04	1.05	1.04	1.04	1.04	1.03	1.03
	4	0.93	0.93	0.96	0.96	**1.00**	1.00	1.00	1.01	1.00	1.00	1.01	1.01	1.01	1.00	1.00	0.99
	5	0.92	0.93	0.96	0.96	1.00	**1.00**	1.01	1.00	1.00	0.99	1.01	1.00	1.01	1.00	0.99	0.99
	6	0.92	0.92	0.95	0.95	0.99	0.99	**1.00**	1.00	0.99	0.99	0.99	1.00	1.00	1.00	0.98	0.98
	7	0.92	0.92	0.95	0.95	0.98	0.98	1.00	**1.00**	0.99	0.99	1.00	1.00	1.00	0.99	0.98	0.98
	8	0.93	0.93	0.96	0.95	1.00	0.99	1.00	1.00	**1.00**	1.00	1.01	1.00	1.01	1.00	1.00	0.99
	9	0.93	0.93	0.96	0.96	1.00	1.00	1.01	1.00	1.00	**1.00**	1.01	1.01	1.01	1.00	0.99	1.00
	10	0.92	0.92	0.95	0.95	0.98	0.99	0.99	0.99	0.99	0.98	**1.00**	1.00	1.00	0.99	0.99	0.98
	11	0.92	0.92	0.95	0.94	0.98	0.98	0.99	0.99	0.99	0.98	1.00	**1.00**	1.00	0.99	0.99	0.98
	12	0.92	0.92	0.95	0.95	0.98	0.99	0.99	0.99	0.99	0.99	1.00	1.00	**1.00**	1.00	0.99	0.99
	13	0.92	0.93	0.95	0.95	0.99	0.99	1.00	1.00	0.99	0.99	1.00	1.00	1.00	**1.00**	0.99	0.98
	14	0.94	0.94	0.97	0.96	1.00	1.00	1.01	1.01	1.00	1.00	1.01	1.01	1.01	1.01	**1.00**	1.00
	15	0.94	0.94	0.97	0.97	1.01	1.00	1.01	1.01	1.00	1.00	1.01	1.01	1.01	1.01	1.00	**1.00**

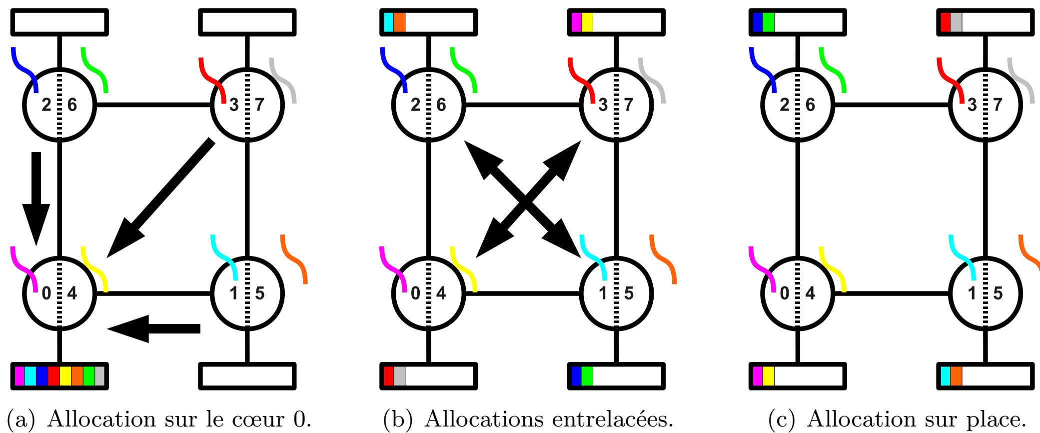
(b) Fonction dGEMM.

		Placement du thread de calcul															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Placement des données	0	**1.00**	0.99	0.98	0.98	0.98	0.99	1.00	1.00	0.98	0.99	0.98	1.00	1.00	0.99	1.00	0.98
	1	0.98	**1.00**	0.99	0.98	0.99	0.98	0.99	0.98	1.00	0.99	0.99	0.99	0.99	0.99	0.99	1.00
	2	1.00	1.00	**1.00**	1.01	1.01	1.00	0.99	1.00	1.00	1.00	1.02	1.00	1.02	1.00	1.00	1.00
	3	0.99	1.01	1.00	**1.00**	1.00	1.02	1.00	1.00	1.00	1.01	0.99	1.00	1.00	0.99	0.99	1.00
	4	1.02	1.00	1.01	0.99	**1.00**	1.02	1.02	1.01	1.01	1.00	1.01	1.03	1.00	1.01	1.01	1.02
	5	1.02	1.01	1.00	1.01	1.02	**1.00**	1.02	1.01	1.02	1.01	1.02	1.02	1.02	1.02	1.01	1.00
	6	1.00	1.00	1.00	0.99	1.00	1.01	**1.00**	1.00	1.01	0.99	0.99	1.00	1.01	1.02	1.00	1.00
	7	1.00	1.00	0.98	0.98	1.00	0.98	1.00	**1.00**	0.99	1.01	1.00	0.99	0.99	1.00	1.00	1.00
	8	0.99	0.97	1.00	1.00	0.98	0.99	0.98	1.00	**1.00**	0.98	0.98	1.00	1.00	0.98	0.99	1.01
	9	0.99	0.98	0.99	0.99	0.99	1.01	0.99	0.99	1.00	**1.00**	1.00	0.99	1.00	0.99	1.00	0.99
	10	1.00	0.99	0.99	1.00	0.99	0.99	1.01	0.99	1.00	1.00	**1.00**	1.00	0.98	1.00	1.00	1.00
	11	1.00	1.00	0.98	1.00	1.00	0.99	0.99	1.00	0.99	1.00	0.98	**1.00**	0.99	1.00	0.99	1.00
	12	1.00	1.00	1.01	1.00	1.02	1.01	1.01	1.01	1.02	1.02	1.00	1.01	**1.00**	1.00	1.00	1.01
	13	1.00	0.99	0.99	1.01	0.99	1.00	1.00	1.01	0.99	1.00	0.99	1.01	1.01	**1.00**	1.00	0.99
	14	1.00	1.00	1.00	1.00	1.01	1.02	1.00	1.01	1.01	1.01	1.00	1.00	1.00	1.01	**1.00**	1.00
	15	0.99	0.99	0.98	0.99	0.98	0.97	0.98	0.99	0.98	0.97	0.98	0.99	0.99	0.97	0.96	**1.00**

opérations dont la réutilisation du cache mémoire est faible et où la bande passante du bus mémoire est un facteur limitant. Par contre, l'utilisation de routine BLAS matrices/matrices varie au maximum de 2%, ce qui est négligeable. Les tableaux 2.1, et 2.2 montrent les résultats obtenus sur les machines NUMA : Borderline et Hagrid, à base d'OPTERON dual-core. On retrouve sur ces résultats la présence des puces dual-core qui ont un accès commun à la mémoire qui leur est proche. Les facteurs NUMA sont ici plus élevés et montrent l'importance de tenir compte du placement des données par rapport au thread de calcul qui doit les utiliser. Lorsque les données sont à l'opposé du thread de calcul, on a alors le plus mauvais placement sur l'architecture et le temps de calcul sur les opérations vecteurs/vecteurs est doublé. Les opérations matrices/matrices ont des facteurs NUMA plus faibles à cause de la quantité plus importante de calculs. Cependant les facteurs peuvent tout de même augmenter de 26% dans le pire des cas. En comparaison, on constate sur notre machine Borderline à quatre OPTERON dual-core que les facteurs NUMA sont au maximum de 58% sur la routine `dAXPY` et de 8% sur la routine `dGEMM`. Quelle que soit l'architecture NUMA utilisée, il apparaît important de tenir compte des facteurs NUMA lors de l'allocation des données dans les codes de calculs scientifiques en mémoire partagée. Cependant cette expérience utilise un seul cœur de la machine cible ce qui n'est pas caractéristique d'une utilisation courante des grappes de calculs dans les simulations scientifiques. Nous allons étudier dans la section suivante l'évolution des facteurs NUMA avec l'utilisation de tous les cœurs disponibles sur la machine.

2.2 Problèmes de contention

L'utilisation d'applications multi-threadées ou OpenMP se fait généralement dans le but d'utiliser l'ensemble des cœurs disponibles et pas seulement un cœur par nœud. C'est pourquoi nous avons étudié l'influence du facteur NUMA dans un contexte où toute la machine est utilisée et où des phénomènes de contention peuvent apparaître sur la machine.



(a) Allocation sur le cœur 0. (b) Allocations entrelacées. (c) Allocation sur place.

FIG. 2.1 – Schémas explicatifs des tests effectués avec contention sur la machine Borderline. (a) Toutes les données sont concentrées sur un seul banc mémoire. (b) Les données sont entrelacées sur la machine. (c) Les données sont allouées au plus près.

Pour cela, nous avons réalisé la même expérience que dans la section précédente en dupliquant les calculs sur tous les cœurs disponibles de la machine. Nous avons testé trois combinaisons de placement des données et des threads de calculs. Les threads de calculs sont fixés toujours avec les mêmes fonctions de manière linéaire sur tous les cœurs disponibles selon la numérotation

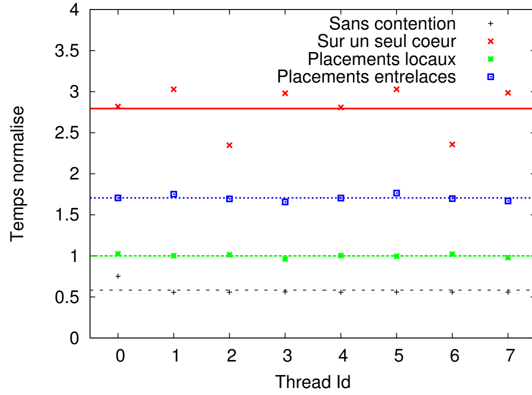
fournie par le système. Dans le premier cas (figure 2.1(a)), les données sont allouées par un seul thread sur le premier cœur de la machine. Cela correspond à la majorité des codes où la partie d'allocation et d'initialisation des données est séparée de la partie calculs. Lors de la parallélisation de ces codes, les programmeurs s'intéressent à la partie coûteuse en temps du code et oublient souvent l'initialisation des données, négligeable sur le temps total de l'exécution du code. Dans le second cas (figure 2.1(b)), nous avons placé les données à l'opposé du thread de calcul sur la machine pour saturer le bus mémoire, ce qui pourrait être obtenu dans le pire des cas avec un système allouant les pages mémoire en *round-robin*. En effet chaque thread de calcul doit systématiquement passer sur cette machine par au moins deux liens `HyperTransport` pour accéder à ses données. Enfin, dans le dernier cas (figure 2.1(c)), les données sont placées au plus près de chaque thread de façon à ne pas utiliser les liens d'interconnexion entre les processeurs.

Les différentes courbes présentées sur la figure 2.2 montrent les résultats obtenus avec ces expériences sur les machines Bordeline à huit cœurs et Hagrid et Decrypthon à seize cœurs. Pour observer également le partage de la bande passante entre les cœurs, les courbes "*Sans contention*" représentent le temps de calcul de chaque cœur dans la première expérience pour le même lot de données, c'est-à-dire sans contention. Pour chacune des machines, on voit apparaître la présence des processeurs à deux cœurs : le temps de calcul est alors divisé par 2 si il n'y a pas de contention sur les routines BLAS1.

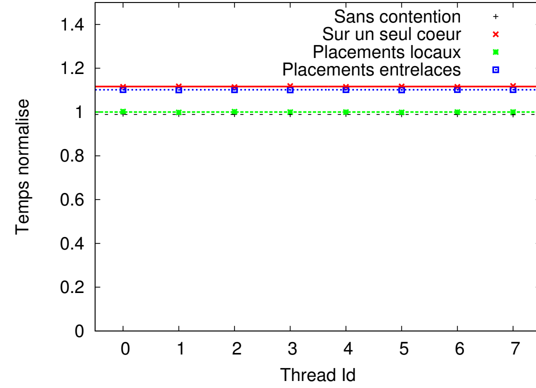
Les courbes "*Sur un seul cœur*" des figures 2.2(a) et 2.2(c) font ressortir l'architecture de la machine. Les deux cœurs d'un même processeur ont les mêmes facteurs NUMA et on peut voir l'impact du passage par un ou plusieurs liens `HyperTransport` pour accéder aux données contrôlées par un autre chipset. Globalement, on constate que la solution de l'allocation locale donne toujours les meilleurs résultats quelle que soit la machine ou la routine utilisée, excepté sur les routines BLAS1 avec la machine Decrypthon (figure 2.2(e)) où les temps sont sensiblement meilleurs dans la configuration "*Sur un seul cœur*".

Les deux machines NUMA confirment la nécessité de prendre en compte les facteurs NUMA dans les codes, puisque les temps de calcul des routines BLAS1 peuvent augmenter de 50 à 180% si l'allocation des données se fait en une seule et unique zone mémoire. Contrairement à ce qu'on aurait pu penser, le facteur NUMA n'augmente pas systématiquement avec le nombre de cœurs sur la machine, mais dépend également de l'architecture puisque les facteurs sont plus importants sur Borderline que sur Hagrid malgré son nombre de cœurs plus faible. L'utilisation des routines BLAS3 inverse cette tendance à cause des processeurs plus performants sur Borderline mais confirme le besoin de prendre en compte la localité des données. Les facteurs NUMA obtenus sont de $1,1$ sur le cluster Borderline à huit cœurs et de $1,5$ sur la machine Hagrid à 16 cœurs dans le cas où la mémoire est allouée sur le nœud 0.

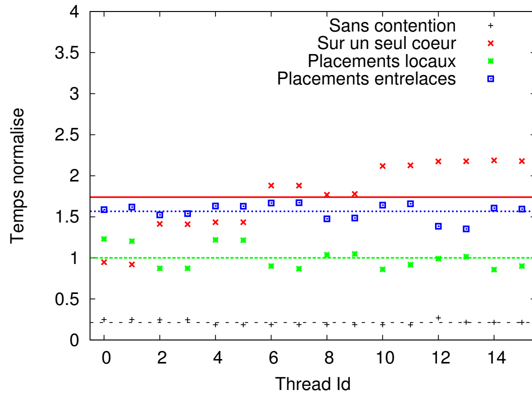
L'ensemble des résultats montre que l'utilisation d'une allocation des pages mémoire avec une répartition de type *round-robin* donne une bonne amélioration des résultats par rapport à une allocation groupée même dans le pire des cas montré ici. Malgré tout, les résultats restent quand même inférieurs à ceux obtenus avec une allocation adaptée : les données sont placées au plus proche du thread de calcul. Enfin, les résultats sur l'architecture SMP confirment une grande stabilité des accès mémoire dans les différentes configurations.



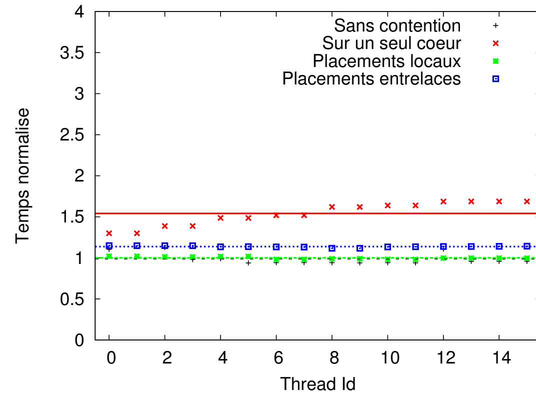
(a) dAXPY sur l'architecture Borderline.



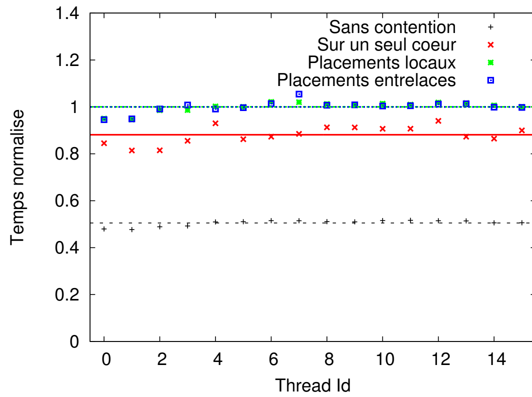
(b) dGEMM sur l'architecture Borderline.



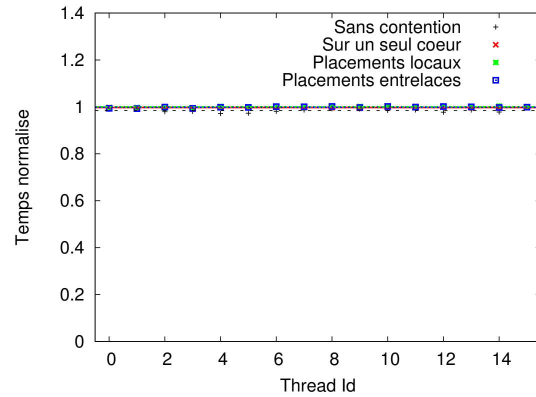
(c) dAXPY sur l'architecture Hagrid.



(d) dGEMM sur l'architecture Hagrid.



(e) dAXPY sur l'architecture Decryphton.



(f) dGEMM sur l'architecture Decryphton.

Fig. 2.2 – Influence de la contention sur nos deux clusters NUMA : Hagrid et Borderline et sur le cluster SMP : Decrypthon. Les temps sont normalisés par rapport au temps moyen du meilleur cas : *Placements locaux*.

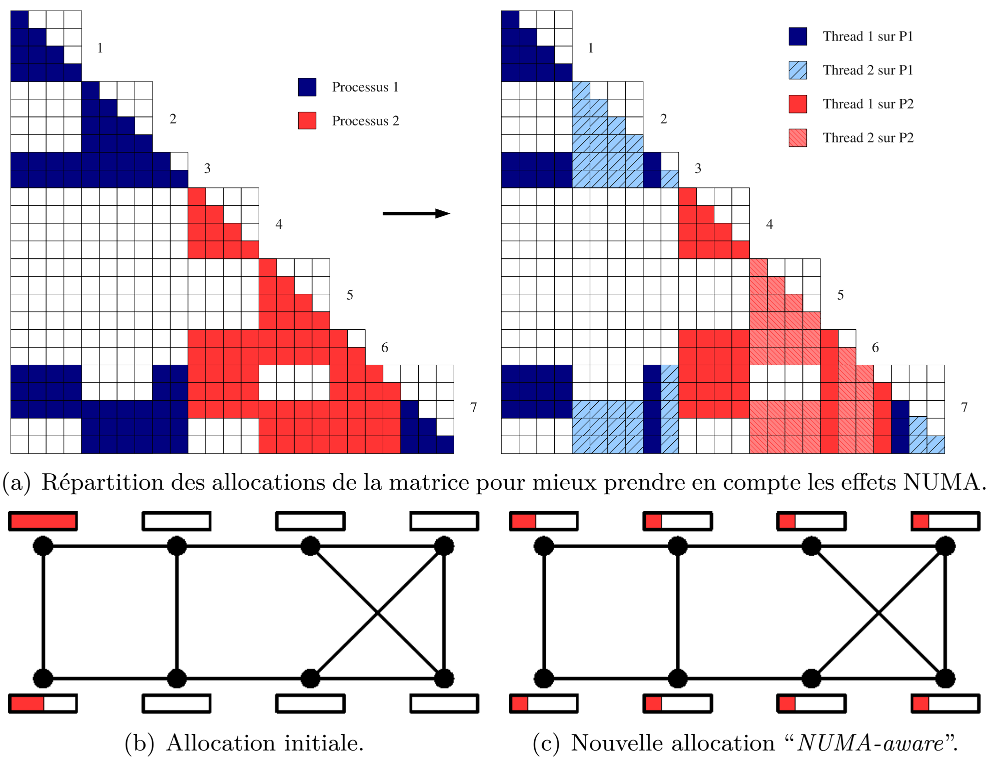
2.3 Améliorations de la gestion mémoire

Le problème souvent rencontré dans les applications multi-threadées sur les architectures NUMA est la séparation de la phase d'initialisation des données de celle des calculs. Cette solution provient souvent d'une parallélisation de la phase de calcul ultérieure. La phase d'initialisation des données en mémoire partagée reste souvent inchangée : c'est également un des avantages de la programmation multi-threadée que d'avoir accès de façon transparente aux mêmes données depuis chacun des threads. On se retrouve alors dans le cas *Allocation sur un seul cœur* présenté dans la section précédente et qui est donc loin d'être optimal. Ce problème est illustré par Henrik Löf *et al*, au sein de leurs travaux sur un solveur PDE [70]. Ils sont passés d'une machine SUN ENTREPRISE 10000 à une machine NUMA SUN FIRE 15000 toutes deux composées de 32 processeurs à respectivement 400MHz et 900MHz. Le temps d'exécution parallèle de leur code a augmenté de 18%, tandis que le temps séquentiel était presque divisé par 5. Le problème vient du processus d'allocation séquentiel des données sur le premier processeur. Le système SOLARIS groupe les pages mémoire demandées à proximité du premier processeur. Lorsque l'exécution parallèle commence, l'ensemble des processeurs accèdent à la mémoire de ce processeur qui se retrouve engorgée.

Une solution simple consiste à utiliser cette particularité du système qui place les pages mémoire à proximité du processeur qui l'initialise. Pour cela, il faut revoir l'étape d'allocation et d'initialisation des données des codes pour la rendre également parallèle. Quand cela est possible, l'initialisation des données doit être faite de manière distribuée sur la machine, c'est à dire par chacun des threads de calcul, et si possible, par ceux qui en auront besoin pour maximiser la localité des données. Il s'agit de reproduire avec la programmation en mémoire partagée, ce qui se fait naturellement dans le modèle de programmation par passage de message utilisée sur les architectures à mémoire distribuée. Dans le cas des codes OpenMP, il est intéressant de paralléliser également les boucles d'allocation et d'initialisation des données en respectant la même distribution que celle utilisée lors des calculs. Ainsi, on retrouve une distribution similaire aux résultats optimaux présentés dans la section précédente.

Dans le cas des solveurs directs, les données les plus utilisées dans les calculs sont bien évidemment les coefficients de la matrice qui sont regroupés en blocs-colonnes (ou en blocs suivant la distribution utilisée, cf section 1.6.1 page 23) de la matrice. La factorisation de la matrice s'effectue bloc-colonne par bloc-colonne. C'est la donnée principale de chacune des tâches attribuées aux threads de calculs. L'optimisation de la gestion de la mémoire dans un solveur direct multi-threadé s'obtient en distribuant la phase d'initialisation des blocs-colonnes sur les processeurs comme pour la distribution inter-nœuds.

Nous avons implémenté cette solution dans notre version hybride MPI/thread du solveur PaStiX. Dans la version initiale, une étape séquentielle réalise l'initialisation avec l'allocation d'un unique tableau de données condensées par nœud de calcul pour toute la matrice, comme représenté sur le schéma de gauche de la figure 2.3(a). On se retrouve dans le pire cas de l'expérience précédente où les données se retrouvent concentrées à proximité d'un seul processeur (Fig. 2.3(b)). La structure de données du solveur a été modifiée pour utiliser un ensemble de tableaux de coefficients, chacun correspondant à un bloc-colonne de la matrice. L'allocation de ces tableaux et leur remplissage avec les coefficients de la matrice ont ensuite été retardés pour être effectuées en parallèle par les threads de calcul. En effet, l'étape de distribution de ce solveur attribue un lot de blocs-colonnes à factoriser à chaque thread. Ainsi, chacun alloue et initialise les blocs qu'il doit factoriser de telle sorte que le système place les pages mémoire à



(a) Répartition des allocations de la matrice pour mieux prendre en compte les effets NUMA.

(b) Allocation initiale.

(c) Nouvelle allocation "*NUMA-aware*".

FIG. 2.3 – Allocation pour architecture NUMA.

proximité (Fig. 2.3(c)). Ces modifications simples peuvent être apportées à tout code de calculs et permettent un gain de performance important. Nous allons examiner dans la section suivante les résultats obtenus grâce à ces modifications sur le solveur PASTIX.

2.4 Présentation des cas tests et des clusters utilisés pour l'étude

Le tableau 2.4 recense les matrices que nous avons utilisées pour nos tests avec leurs caractéristiques. Elles sont regroupées par type de coefficients : réels ou complexes et triées par ordre décroissant du nombre d'opérations nécessaires pour les factoriser. On trouve également dans le tableau le nombre de termes non-nuls de la matrice factorisée après renumérotation avec le logiciel SCOTCH [79]. Les matrices proviennent de différentes sources universitaires ou industrielles et sont disponibles pour la plupart sur le site de GRIDTLSE (`http://gridtlse.org/`) :

- BRGM : Problèmes de sismologie du BRGM, `http://www.brgm.fr/`
- CEA-Cesta : Problème d'électromagnétisme du CEA, `http://www.cea.fr/`
- UFL : Collection de l'Université de Floride de Tim Davis [27], `http://www.cise.ufl.edu/research/sparse/matrices/`
- UMN : Université du Minnessota, `http://www.cs.umn.edu/`
- PARASOL : Collection PARASOL, `http://www.parallab.uib.no/projects/parasol/`

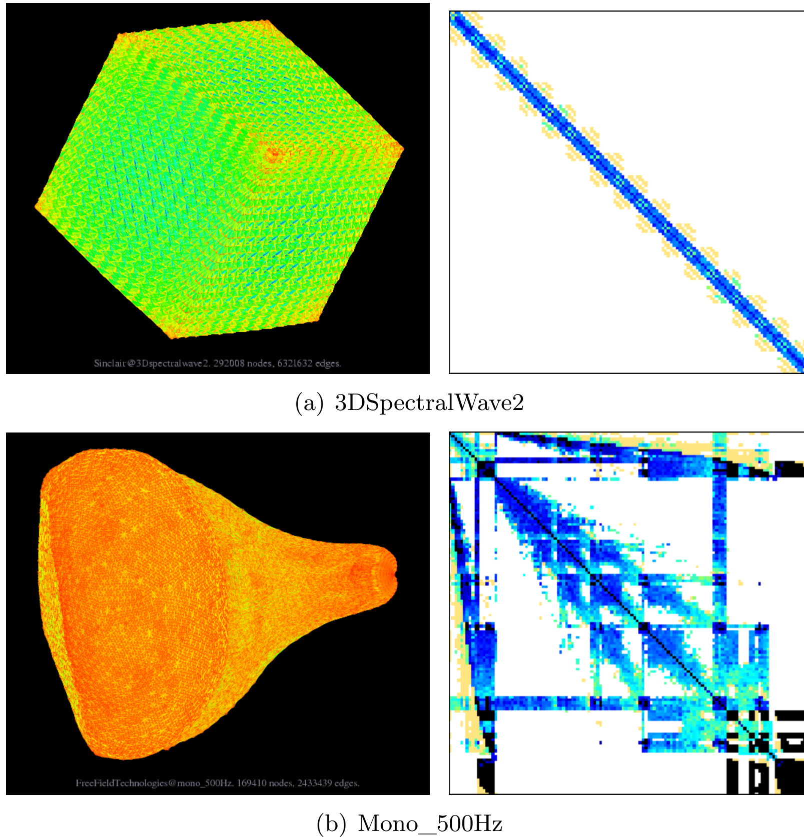
TAB. 2.4 – Caractéristiques des matrices utilisées pour l'ensemble des évaluations.

Nom	**N**	NNZ_A	NNZ_L	**OPC**	**Taille**	**T**	**S**	**Source**
Matr5	485 597	12 359 369	680 915 459	9.84e+12	10,3	D	U	UMN
Matr6	470 596	12 127 402	637 756 616	9.10e+12	9,66	D	U	UMN
Audi	943 695	39 297 771	1 141 513 029	5.21e+12	8,73	D	S	PARASOL
Nice20	715 923	28 066 527	1 050 576 453	5.19e+12	8,02	D	S	BRGM
Inline	503 712	18 660 027	158 830 261	1.41e+11	1,27	D	S	PARASOL
Nice25	140 662	2 914 634	51 133 109	5.26e+10	0,41	D	S	BRGM
Mchlnf	49 800	4 136 484	22 878 995	4.79e+10	0,37	D	U	UMN
Thread	29 736	2 249 892	25 370 568	4.45e+10	0,20	D	S	PARASOL
3DSpectralWave	680 943	17 165 766	1 340 207 093	1.00e+13	21,1	Z	H	UFL
3DSpectralWave2	292 008	7 307 376	394 131 174	1.67e+12	6,22	Z	H	UFL
Haltere	1 288 825	10 476 775	404 977 313	7.55e+11	6,46	Z	S	CEA-Cesta
FemHifreqCircuit	491 100	10 365 178	178 227 119	4.75e+11	5,6	Z	U	UFL
Mono_500hz	169 410	2 602 849	77 043 060	2.61e+11	2,42	Z	U	UFL

Notations :

- N est le nombre d'inconnues de la matrice, c'est à dire sa dimension.
- NNZ_A est le nombre de termes non nuls dans la partie triangulaire inférieure du graphe symétrisé de la matrice A.
- NNZ_L est le nombre de termes non nuls de la matrice triangulaire inférieure L.
- OPC est le nombre d'opérations double précision nécessaires à la factorisation numérique de la matrice après amalgamation.
- *Taille* est l'espace mémoire occupé par les coefficients de la matrice en Gigaoctets
- T est le type de la matrice : réel double précision (D) ou complexe double précision (Z)
- *Sym* indique la symétrie de la matrice. Les matrices réelles sont soit symétriques (S), soit non-symétriques (U) et les matrices complexes sont soit hermitiennes (H), soit symétriques (S) soit non-symétriques (U). Les matrices réelles symétriques et les matrices complexes symétriques et hermitiennes sont factorisées par la méthode de Cholesky (ou Cholesky-

Crout) et les autres par décomposition LU.



(a) 3DSpectralWave2

(b) Mono_500Hz

Fig. 2.4 – Graphes et structures des matrices.

Tab. 2.5 – Caractéristiques des clusters utilisés pour l'ensemble des évaluations.

	NUMA			SMP	
	Hagrid	Borderline	Titane	Decrypthon	Vargas
Type	AMD Opteron	AMD Opteron	Intel Nehalem	IBM Power5	IBM Power6
Nb. processeurs	8	4	2	8	16
Nb. coeurs par processeur	2	2	4	2	2
Nb. coeurs par noeud	16	8	8	16	32
Memoire par coeur en Go	4	4	3	1,75	3,2
Memoire par noeud	64	32	24	28	102,4
Nb. noeuds	1	10	1068	12	112
Réseau	-	Infiniband / Myrinet	Infiniband	IBM Federation	Infiniband
MPI	-	Mvapich2	Intel MPI	IBM MPI	IBM MPI
BLAS	ACML	ACML	MKL	ESSL	ESSL

Toutes ces matrices sont issues de divers domaines d'applications comme la sismologie, l'électromagnétisme, l'électronique ou la mécanique des fluides et ont par conséquent des structures très différentes, comme on peut le voir sur la figure 2.4. Il y a ainsi des matrices symétriques et non-symétriques, de même que des matrices à coefficients réels ou complexes. La quantité de calculs nécessaire à la factorisation de ces problèmes varie de 4,4 gigaopérations à 10 teraopérations. Nous verrons également dans le chapitre 5.3 page 86 un cas à 10 millions d'inconnues qui nécessite 50 teraopérations pour être factorisé.

Dans la suite de cette thèse, nous utiliserons ces matrices pour étudier l'évolution des performances du solveur PaStiX sur le temps de factorisation avec les apports de cette thèse. Pour chaque cas étudié, nous calculons une solution précise à 10^{-15} près, avec une étape de résolution, puis une ou plusieurs itérations de raffinement. Bien que nous ne montrions que des résultats sur le temps de factorisation, les développements réalisés concernent également l'étape de résolution. L'évolution des résultats sur cette étape est similaire à celle obtenue sur la factorisation mais les temps ne dépassant pas quelques secondes, les gains sont moins visibles.

L'ensemble des évaluations a été réalisés sur plusieurs clusters dont le tableau 2.5 récapitule les caractéristiques. Nous avons expérimenté nos travaux sur les diverses architectures disponibles dans les centres de calculs Français auxquels nous pouvions avoir accès : IDRIS, CINES et CCRT mais également sur les clusters de l'Université de Bordeaux 1.

2.5 Evaluation sur le solveur PaStiX

Tab. 2.6 – Influence de la méthode d'allocation des données sur le temps de factorisation numérique en secondes du solveur PaStiX. V0 est la version d'origine du solveur. N est la version du solveur avec la nouvelle stratégie d'allocation des blocs-colonnes.

Matrice	Borderline			Hagrid			Decrypthon	
	V0	N	Gain	V0	N	Gain	V0	N
Matr5	469	**422**	10 %	481	**386**	19,7%	162	**161**
Matr6	426	**405**	4,9 %	450	**361**	19,8%	**177**	178
Audi	274	**244**	10,9 %	255	**230**	9,8%	101	**99,9**
Nice20	248	**230**	7,2 %	248	**188**	24,2%	91,8	**90,7**
Inline	9,07	**7,95**	12,3 %	18,4	**17,3**	6 %	5,88	**5,62**
Nice25	3,28	**2,62**	20,1 %	7,23	**5,08**	29,7%	2,07	**1,9**
Mchlnf	3,13	**2,43**	22,4 %	5,86	**3,24**	44,7%	1,92	**1,89**
Thread	2,49	**2,17**	12,8 %	3,84	**2,26**	41,1%	1,15	**1,12**
3DSpectralWave	2060	**1650**	19,9%	1290	**1040**	19,4%	**575**	603
3DSpectralWave2	311	**287**	7,7 %	267	**174**	34,8%	**107**	**107**
Haltere	140	**140**	0 %	124	**92.1**	25,7%	48.2	**47,7**
Fem_Hifreq_Circuit	**101**	106	-4,9 %	82	**68**	17,1%	32,9	**32,3**
Mono_500Hz	51,8	**50,1**	3,3 %	41,6	**37,1**	10,8%	18,8	**18,7**

Le tableau 2.6 montrent les résultats obtenus sur le solveur PaStiX avec les cas tests présentés précédemment. Il s'agit des temps de factorisation numérique en secondes sur la version d'origine (V0) et sur la version avec la nouvelle stratégie d'allocation (N) sur les trois machines présentées dans le chapitre 1. Pour chacune des machines, nous avons utilisé un seul nœud avec autant de threads de calcul qu'il y a de cœurs disponibles sur la machine. Huit threads ont donc été utilisés sur Borderline et seize sur les clusters Hagrid et Decrypthon. Les résultats obtenus montrent des gains importants sur les machines NUMA allant jusqu'à 45% sur la machine Hagrid sur les cas de taille plus petite. Les performances sur les plus gros cas tests sont améliorées de 10 à 20% sur les deux machines NUMA. Ces résultats confirment la différence que l'on obtenait sur nos évaluations de l'impact de la contention sur les routines BLAS3 (cf figures 2.2(b) et 2.2(d) page 36). Les résultats sur la machine SMP Decrypthon montrent que cela ne coûte rien de prendre en compte les facteurs NUMA sur ces architectures. Au contraire, on observe également un léger gain sur le temps de calcul mais qui reste inférieur à 5%.

Les résultats sur ce solveur hautes performances confirment les premières expériences sur le placement et la contention sur machines NUMA. Il est important de bien prendre en compte les facteurs NUMA et la localité des données dans les codes de calculs qui ont des besoins mémoire importants. Une meilleure stratégie d'allocation peut améliorer les temps d'exécution de ces algorithmes par des facteurs non négligeables (ici 1,5 fois plus rapide sur les meilleurs cas). De plus, on constate que ce phénomène devient plus important avec l'augmentation de la taille des machines qui impliquent de nouveaux niveaux hiérarchiques dans la topologie. Les gains sont en moyenne deux fois plus importants sur Hagrid que sur Borderline. Ce changement de stratégie d'allocation est une étape importante dans l'adaptation du solveur PaStiX aux architectures NUMA, surtout que les clusters évoluant en tête du Top500 contiennent désormais 16 cœurs ou plus et la tendance est à l'augmentation du nombre de cœurs par nœud de calcul. Cette nouvelle méthode d'allocation est présente et activée par défaut depuis les deux dernières versions du solveur PaStiX.

Chapitre 3

Gestion des communications plus adaptées à un ordonnancement dynamique

Sommaire

3.1 Les schémas de communication dans le HPC **44**
3.2 Utilisation d'un thread de progression **45**
3.3 Une solution plus portable **48**
3.4 Évaluation des solutions proposées sur le solveur PaStiX **51**

Dans les applications distribuées, les communications sont souvent une limite critique à la scalabilité du code. Les programmeurs essaient autant que possible de recouvrir le transfert des données par des calculs. Une partie du coût de la communication est ainsi masquée dans le temps d'exécution de la simulation, puisque communications et calculs se font simultanément. Les bons résultats du solveur PaStiX sont basés sur l'utilisation de cette technique de recouvrement, mais également sur une simulation des calculs qui permet de distribuer les données sur les nœuds de calcul en minimisant le nombre de communications. De plus, cette simulation permet de prévoir en amont de la factorisation numérique la priorité des données à envoyer et de grouper les envois de même destination. L'ensemble des communications est ainsi ordonné de la même façon que les tâches de calculs.

Cette solution n'est malheureusement plus adaptée à un ordonnancement dynamique des calculs. En effet, l'ordre des calculs n'est plus déterminé à l'avance. Et il devient nécessaire d'envoyer les données, mais également de les recevoir le plus tôt possible pour s'adapter à la réorganisation des calculs et éviter l'inactivité d'une partie des ressources de calcul. Nous allons dans cette section présenter les travaux réalisés sur le schéma de communication du solveur PaStiX, dans le but de répondre à l'utilisation d'un ordonnancement dynamique. Nous présenterons ensuite une solution que nous avons développée pour étendre le parc de clusters pouvant utiliser le solveur PaStiX. Ces développements seront ensuite évalués sur un ensemble de cas tests sur différentes machines.

3.1 Les schémas de communication dans le HPC

La technique de recouvrement des communications par les calculs est une technique classique du parallélisme pour masquer les temps de transfert dans les applications largement distribuées. Pour cela, les programmeurs utilisent les différents types de communications mis à leur disposition. La première différence sur les communications concerne la taille des données échangées. Les petits échanges de données se font par copie des données à envoyer dans un buffer spécifique à la bibliothèque ou directement par écriture dans la mémoire du nœud destination grâce aux techniques de *RDMA (Remote Direct Memory Access)* [4] disponibles sur les cartes réseaux. Au delà d'une taille seuil, les paquets ne peuvent plus être transférés par anticipation à cause de l'espace mémoire nécessaire. Pour résoudre ce problème, les bibliothèques de communication utilisent le protocole de *Rendez-vous*. L'émetteur envoie un premier message au destinataire : *Request to Send (RTS)* pour lui signaler l'arrivée du message. Celui-ci, dès qu'il est prêt à recevoir les données, c'est à dire dès que l'espace mémoire nécessaire est alloué, répond à l'émetteur par un : *Clear To Send (CTS)* qui peut alors, commencer à lui envoyer les données. Ce protocole est illustré sur les figures 3.1.

En plus de ces deux protocoles de communication dépendants de la taille des messages, on trouve deux types d'appels de fonction. Les premières sont les communications synchrones dont les appels de fonctions ne rendent la main qu'une fois l'échange de données terminé comme `MPI_Send` et `MPI_Recv`. Ce type de communication ne permet pas de faire du recouvrement calculs/communications dans le cas du protocole de *Rendez-vous* (figure 3.1(a)), mais au contraire assure la possibilité de réutiliser les buffers de communications immédiatement après l'appel à la fonction. Elles peuvent également être utilisées pour obtenir une synchronisation légère des deux processus communiquant entre eux. Les deuxièmes types sont les communications asynchrones qui permettent le recouvrement des communications par les calculs comme `MPI_Isend` et `MPI_Irecv`. Elles enregistrent la demande d'échange et rendent la main aussitôt. Elles font ensuite progresser la copie ou la demande de *Rendez-vous* en arrière-plan. On peut ainsi faire d'autres calculs pendant l'échange mais on ne peut pas utiliser les zones mémoire échangées avant l'appel à `MPI_Wait` qui assure la fin du transfert.

Cependant toutes les implémentations de cette API ne fournissent pas toujours une bonne *progression asynchrone* des communications et particulièrement avec le protocole de *Rendez-vous* utilisé pour les échanges de taille importante. Les figures 3.2(a) et 3.2(b) illustrent ce problème. Une mauvaise progression des communications peut ainsi donner des résultats comparables à l'utilisation de communications synchrones : l'échange des données et les calculs sont sérialisés et ne donnent aucun ou peu de recouvrement.

Pour faire progresser les communications de manière efficace, les différentes implémentations de `MPI` utilisent les capacités des cartes réseaux comme par exemple la possibilité d'utiliser la technologie RDMA. Ainsi pour les petits paquets, la carte réseau gère directement l'échange sans perturber le processeur. Par contre, pour les messages de taille plus importante, le protocole de *rendez-vous* est plus difficile à faire progresser efficacement. Certaines des bibliothèques `MPI` ou des drivers utilisés proposent désormais des threads de progression pour faire avancer l'échange plus régulièrement et rapidement. `Mpich2` [7] n'utilise pas cette solution mais tire parti du thread de progression disponible dans le driver `MX` [76]. OpenMPI [49] intègre directement un thread de progression qui lui permet d'améliorer la latence sur les communications comme les auteurs le montrent dans [99]. Enfin d'autres bibliothèques comme PIOMan proposent une solution efficace pour améliorer la réactivité des entrées/sorties sur tous les réseaux [96]. Cette solution se base sur la bibliothèque de communication hautes performances NewMadeleine [17] et sur la bibliothèque de thread Marcel [77]. L'intégration de l'ordonnancement des communications

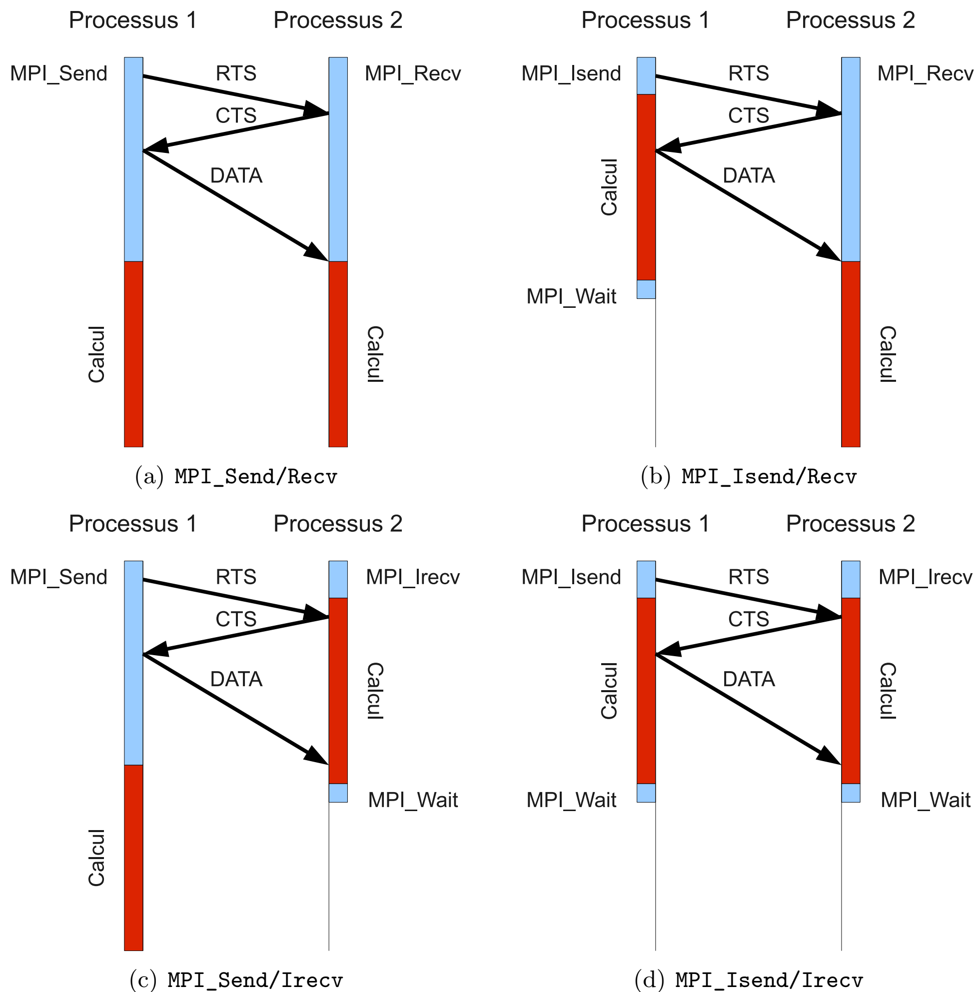


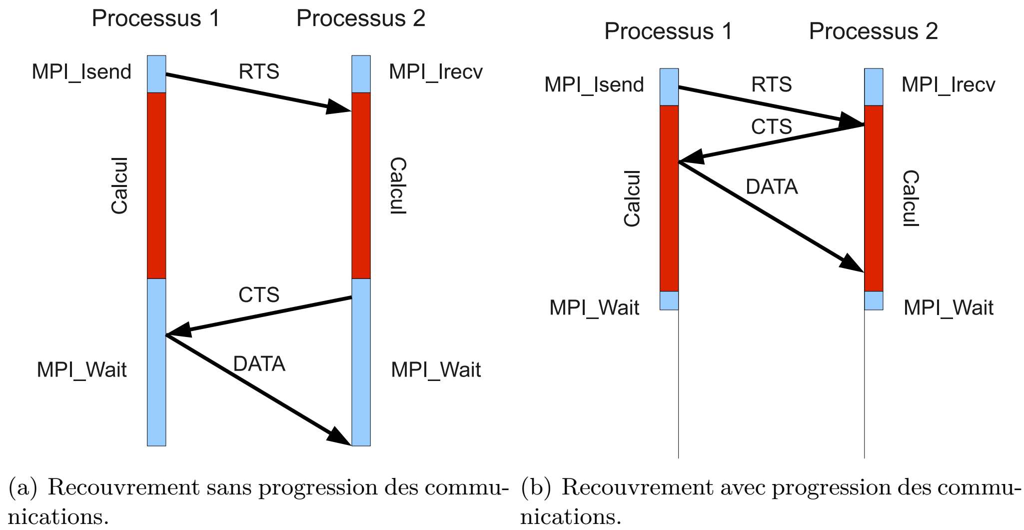
FIG. 3.1 – Comparaison de l'efficacité théorique des différentes combinaisons de communications synchrones et asynchrones avec le protocole de *rendez-vous*.

au même niveau que celui des threads permet de scruter périodiquement l'état du réseau sans entraîner de surcoût prohibitif à son utilisation.

Suivant ce principe, nous avons souhaité faire progresser nos réceptions plus régulièrement quelque soit l'implémentation `MPI` utilisée. Pour cela, et pour répondre au besoin d'un ordonnancement dynamique dans le solveur PaStiX, nous voulons modifier l'algorithme de factorisation numérique pour déléguer la réception des données à un thread spécifique. Nous allons voir maintenant les changements apportés à l'algorithme pour répondre à cette requête.

3.2 Utilisation d'un thread de progression

L'algorithme de PaStiX dans sa version d'origine prédit l'ordonnancement statique des calculs et des communications en se basant sur un modèle de coût prédéfini, mais également sur la supposition que les communications sont correctement recouvertes. L'algorithme 2 page 26 montre le déroulement de la boucle principale de la factorisation numérique dans la version parallèle multi-threadée du solveur. Chaque thread de calcul effectue une boucle de calcul sur



(a) Recouvrement sans progression des communications.

(b) Recouvrement avec progression des communications.

FIG. 3.2 – Comparaison du recouvrement calculs/communications possible avec ou sans une progression des communications efficace.

l'ensemble des tâches (blocs-colonnes) qui lui sont affectées lors du calcul de l'ordonnancement. À chaque itération, l'algorithme réalise trois opérations : la réception des données distantes et locales nécessaires au calcul de la tâche, la factorisation du bloc-colonne ainsi que le calcul des contributions résultantes et enfin l'envoi des contributions. En réalité, seule une partie des contributions qui viennent d'être calculées sont envoyées en les agrégeant ou non à d'autres qui ont été calculées précédemment. Certaines sont stockées en local avant d'être envoyées si la prédiction statique a prédit qu'il était préférable d'attendre d'autres contributions pour les agréger et gagner en temps de communication.

Plusieurs problèmes se posent sur cet algorithme vis-à-vis de notre volonté de faire un ordonnancement dynamique et par rapport au problème d'efficacité du recouvrement qu'on a mentionné dans la section précédente.

L'ordre des calculs n'est plus déterminé, ce qui impose de revoir également l'ordre des communications pour ne pas créer des périodes d'inactivité, voire des blocages dans l'exécution du code. De plus, les contributions issues du calcul de la $i^{ème}$ tâche ne sont pas nécessairement envoyées immédiatement si la prédiction a estimé qu'il était préférable de retarder l'émission pour la grouper avec d'autres données de même destination. Il est nécessaire de revoir l'ordre des communications tout en essayant de préserver le maximum d'agrégation des paquets pour limiter le nombre d'échanges. Pour cela, nous avons changé la structure de données utilisée. La version d'origine utilise un ensemble de tableaux des données à envoyer par destination triés selon la prédiction statique. Ces tableaux ont été remplacés par des files triées selon le même critère, mais qui ne contiennent que les communications prêtes à l'envoi. Dans la version d'origine, à la fin du calcul de chaque tâche, on parcourt pour chaque nœud destination le tableau des échanges à partir de l'indice courant jusqu'à la dernière tâche prête à être envoyée. Il est possible alors pour plusieurs destinations d'avoir des contributions non prêtes qui empêchent l'envoi des suivantes. Pour éviter cela, dans la version dynamique, on ajoute dynamiquement les contributions à envoyer dans la file associée au processus destination. De cette façon, à la fin de la factorisation d'un bloc colonne, on envoie immédiatement tout ce qui est disponible dans les files, en groupant les données par destination. Le nombre d'échanges effectifs est ainsi plus

important que dans la version d'origine. Le nombre de communications pour un bloc-colonne est borné par le nombre de fils de ce bloc-colonne dans l'arbre d'élimination contre auparavant le nombre de processeurs possédant ces fils. En réalité, l'ordre de traitement des blocs-colonnes choisi dans le solveur fait que le nombre d'échanges augmente très faiblement par rapport à la version d'origine.

En plus de l'ordre des calculs, la prédiction impose également le placement des calculs et des communications sur chacun des threads. De cette façon, dans la version d'origine, les contributions sont envoyées directement au thread qui sera en charge de factoriser le bloc-colonne destinataire (ligne 27 de l'algorithme 2 page 26). Pour cela les communications sont envoyées à ce processus qui détient le thread destinataire et le message est marqué par un *tag* correspondant à l'identifiant du thread. De cette façon les réceptions sont en attente sur un marqueur différent pour chaque thread et il n'est pas possible que l'un réceptionne les données destinées à un autre.

Dans l'objectif d'aller vers un ordonnancement dynamique, on veut supprimer cette association qui existe entre les tâches et les threads de façon à ce que chaque thread puisse exécuter n'importe quelle tâche. Il n'est donc plus nécessaire de connaître le thread à qui envoyer les données mais seulement le processus MPI qui en a besoin. Ceci semble à première vue plus facile à mettre en œuvre puisqu'il suffit de supprimer les marqueurs sur les communications qui permettaient de distinguer les destinations. Or si on conserve l'algorithme décrit précédemment, il devient complexe de laisser l'ensemble des threads recevoir les données sans risquer de bloquer l'application dans des attentes de communications. En effet, on sait toujours quel est le nombre de contributions que l'on doit recevoir de la part de chaque processus puisque cela reste déterminé par la prédiction statique du solveur. Cependant, avec les changements apportés sur l'algorithme d'agrégation des paquets, il est désormais impossible de savoir combien d'échanges vont se faire entre les processus et ainsi éviter les attentes de communications inutiles et bloquantes ou les tests de réceptions trop nombreux. Pour résoudre ce problème, nous nous sommes basés sur le principe décrit dans la section précédente 3.1 qui consiste à utiliser un thread supplémentaire pour faire progresser la réception des communications. Il permet de simplifier l'algorithme dynamique aussi bien que celui de la version statique. L'algorithme 3 montre le déplacement de la boucle de réception des données. Elle est faite désormais par le thread dédié aux communications et non plus par tous les threads de calcul. De cette façon, le calcul d'une tâche n'attend plus que sur un seul compteur local de contributions.

Ce nouvel algorithme permet non seulement de mieux s'adapter à l'ordonnancement dynamique que l'on veut mettre en place, mais également d'améliorer le recouvrement des échanges de données dans la version statique. Le thread supplémentaire dédié aux réceptions des données permet de faire progresser plus régulièrement les communications quelque soit l'implémentation `MPI` utilisée. La progression se fait simplement par une boucle d'attente sur la fonction `MPI_Wait` qui teste l'arrivée de données. En le dédiant à un thread supplémentaire, le test est effectué à chaque changement de contexte où ce thread prend la main, contrairement à la version initiale où l'attente des communications se fait à la fin de la factorisation d'un bloc-colonne. Cette solution permet d'avoir des performances plus uniformes sur les communications mais nous allons constater qu'elle est encore problématique dans certains cas. En effet, certaines bibliothèques MPI n'autorisent pas tous les threads à communiquer. Pour remédier à ce problème, nous avons utilisé cette version pour développer un schéma de communication moins contraignant vis-à-vis des implémentations MPI pour permettre une meilleure portabilité et une diffusion plus importante du solveur PaStiX.

Algorithme 3 Utilisation d'un thread de progression en réception.

```
1:  /* Thread de calcul */
2:  Pour l = 1 à NT_t Faire
3:      k = Task(t, l)
4:      Tant que NC_k > 0 Faire
5:          Attendre signal
6:      Fin de Tant que
7:      Calcul de la tâche l : factorisation de A_{k,k} et résolution de L_k et U_k      /* Calcul */
8:      Calcul des contributions des blocs extra-diagonaux A_{i,j} avec (i, j) ∈ [[1, b_k]]^2, avec mise à
    jour de NC_i et envoi du signal (correspond aux lignes 12 à 24 de l'algorithme 2 page 26)
9:      Pour i = 1 à b_k Faire
10:         Si A_i est non local et C_i complet alors
11:             Envoi de la contribution C_i au nœud Proc_i                              /* Envoi */
12:         Fin de Si
13:     Fin de Pour
14: Fin de Pour
15:
16: /* Thread de communication */
17: Tant que Il reste des données à recevoir Faire
18:     Attendre une contribution
19:     Ajouter la contribution C_i reçue
20:     NC_i − −;                                                                        /* Réception */
21:     Si NC_i = 0 alors
22:         Envoyer signal pour i
23:     Fin de Si
24: Fin de Tant que
```

3.3 Une solution plus portable

Le choix du développement du solveur PaStiX dans une version hybride MPI/Threads est une solution efficace et performante mais qui comporte des inconvénients. La section précédente a mis en avant le fait que tous les threads peuvent communiquer entre eux d'un processus à un autre. De cette façon, le solveur délègue le plus possible la gestion des communications et de leurs données associées à la bibliothèque `MPI`. L'implémentation de `MPI` choisie peut alors gérer directement les accès concurrents sur les données spécifiques à l'`API`. Les zones protégées par exclusions mutuelles peuvent être plus fines que si leur gestion était laissée au niveau applicatif.

Cependant toutes les implémentations `MPI` ne proposent pas un support complet des threads dans les applications comme il est requis dans le solveur PaStiX. Il existe quatre niveaux de support des threads dans la norme de `MPI` :

- `MPI_THREAD_SINGLE` : ce niveau n'offre aucun support de thread, l'application ne doit tout simplement pas en utiliser.
- `MPI_THREAD_FUNNELED` : ce niveau est le plus courant. Il ne pose pas de problèmes majeurs et offre la possibilité à l'application d'utiliser des threads. Mais seul le thread initial de l'application peut faire appel aux routines `MPI`. Dans un code `OpenMP`, cela revient à placer les appels MPI dans les sections de codes non parallélisées.
- `MPI_THREAD_SERIALIZED` : ce niveau permet à l'application d'être multi-threadée et à tous les threads de disposer des routines `MPI`. Le programmeur est chargé de protéger les

appels aux fonctions `MPI` dans son code pour garantir que deux appels ne peuvent avoir lieu simultanément.
- `MPI_THREAD_MULTIPLE` : C'est le support maximal des threads dans l'application. Ils ont tous accès aux routines `MPI` et c'est la bibliothèque qui s'occupe de protéger les données partagées et d'éviter l'apparition de blocages lors des appels aux fonctions de communication.

Les algorithmes 2 et 3 nécessitent tous les deux le niveaux `MPI_THREAD_MULTIPLE` puisque tous les threads doivent pouvoir faire des envois et/ou des réceptions. Ce support des threads est bien entendu le moins évident à mettre à disposition dans les bibliothèques `MPI`. R. Thakur and W. Gropp recensent dans [13, 51] les problèmes liés à l'implémentation des différents niveaux de supports de threads ainsi que des comparatifs sur le surcoût apporté aux appels de fonctions relativement au support choisi. Plus le niveau de support est élevé, plus la protection des ressources partagées par exclusion mutuelle est intégrée dans la bibliothèque et entraîne un surcoût sur les appels de fonctions. Au contraire, plus le support des threads est faible, plus la protection des ressources partagées est remontée au niveau de l'application. Le surcoût sur les fonctions `MPI` est très faible mais le coût de la protection dans l'application peut être plus important. Le programmeur doit choisir le niveau de support le plus adapté à son application pour obtenir les meilleures performances. Cependant, au vu de tous les prérequis nécessaires au support maximal des threads dans les bibliothèques `MPI`, toutes les implémentations ne le supportent pas encore complètement ou seulement sur certains réseaux. Celles qui acceptent `MPI_THREAD_MULTIPLE` sont par exemple : OpenMPI et Mpich2 sur TCP, Mvapich2 [64] qui est une extension de Mpich2 sur `Infiniband`, IntelMPI [2], l'implémentation IBM pour leur réseau propriétaire `Fédération` ou encore MPI-Bull [21] sur `Quadrics` pour le cluster TERA10 du CEA. Quelques implémentations proposent ce niveau de support de threads mais il existe encore quelques réseaux comme Myrinet sur lesquels les bibliothèques `MPI` ne proposent pas un tel support des threads.

Certains clusters de calcul ne disposent pas de ces bibliothèques. Par exemple, le cluster Platine du CCRT dispose d'un réseau Infiniband et ne propose que le niveau de support `MPI_THREAD_FUNNELED`. De même, plusieurs clusters de la grille de calcul Grid5000 [1] utilisent des réseaux Myrinet, Infiniband ou GigaEthernet et proposent uniquement le niveau `MPI_THREAD_FUNNELED` de support des threads excepté sur les réseaux Ethernet et quelques exceptions. Pour remédier à ce problème qui nous empêchait d'utiliser le solveur PaStiX sur ces clusters, nous avons modifié l'algorithme présenté précédemment pour développer une version du solveur moins exigeante qui ne requiert que le niveau `MPI_THREAD_FUNNELED`. Le thread de communication qui réceptionne les données doit désormais également envoyer les contributions, de telle sorte qu'il fait progresser émissions et réceptions. L'algorithme 4 présente cette solution : les threads de calcul ne font plus les communications mais postent les contributions prêtes à l'envoi dans une file comme dans la version précédente, à l'exception qu'on n'utilise plus qu'une seule file. Les données sont triées par destination et par ordre de priorité. Le thread de communication alterne dans cette version entre envois et réceptions des données. Les envois sont envoyés cycliquement aux différentes destinations. Pour ne pas rester bloqué sur une des deux communications, on ne fait que des appels asynchrones sur les envois et des tests sur des communications persistantes en réception. Ces réceptions sont ensuite relancées dès qu'un paquet est arrivé.

Le problème d'un tel algorithme est la sérialisation des communications car elles sont toutes effectuées par un seul thread. On réduit grandement la proportion de calculs à réaliser en parallèle

Algorithme 4 Version *Funneled*

```
1:  /* Thread de calcul */
2:  Pour l = 1 à NT_t Faire
3:      k = Task(t, l)
4:      Tant que NC_k > 0 Faire
5:          Attendre signal
6:      Fin de Tant que
7:      Calcul de la tâche l : factorisation de A_{k,k} et résolution de L_k et U_k      /* Calcul */
8:      Calcul des contributions des blocs extra-diagonaux A_{i,j} avec (i, j) ∈ [[1, b_k]]^2, avec mise à
    jour de NC_i et envoi du signal (correspond aux lignes 12 à 24 de l'algorithme 2 page 26)
9:      Pour i = 1 à b_k Faire
10:         Si A_i est non local et C_i complet alors
11:             Ajouter la contribution C_i dans la file des envois      /* Envoi */
12:         Fin de Si
13:     Fin de Pour
14: Fin de Pour
15:
16: /* Thread de communications */
17: Démarrer une réception
18: Tant que Il reste des données à recevoir ou à envoyer Faire
19:     Si Il reste des données à recevoir alors
20:         Tester si une contribution est arrivée
21:         Si Ok alors
22:             Ajouter la contribution C_i reçue
23:             NC_i − −;      /* Réception */
24:             Si NC_i = 0 alors
25:                 Envoyer signal pour i
26:             Fin de Si
27:         Fin de Si
28:     Fin de Si
29:     Si La file des données à envoyer est non-vide alors
30:         Envoyer les données en attente      /* Envoi */
31:     Fin de Si
32: Fin de Tant que
```

lors des communications et on ne peut pas tirer parti de certaines optimisations permises au niveau de l'implémentation MPI. Comme dans les différents niveaux de supports des threads, on remonte la protection des données partagées à un niveau supérieur (ici la file des données à envoyer). Les zones protégées sont plus grandes et les attentes sur ces données deviennent ainsi plus fréquentes avec l'augmentation du nombre de threads, contrairement à une utilisation en MPI_THREAD_MULTIPLE qui va pouvoir limiter les zones critiques à leur plus simple expression. Cette solution a été développée pour rendre le solveur plus portable et le rendre disponible sur les clusters qui n'ont pas d'implémentation MPI supportant pleinement les threads. Mais elle reste une version "dégradée" du solveur, puisque les performances sont moins bonnes que pour les autres versions comme on va le voir dans la section suivante.

3.4 Évaluation des solutions proposées sur le solveur PaStiX

Les trois versions de schémas de communication disponibles dans le solveur PaStiX sont résumées sur la figure 3.3. Le premier *Multiple* est le schéma d'origine du solveur : tous les threads s'échangent directement entre eux les données dont ils ont besoin. Le second *ThComm* est celui qui a été développé pour la version dynamique. Un thread supplémentaire est chargé de recevoir l'ensemble des communications venant de n'importe quel thread de calcul des autres nœuds. Il doit permettre de faire progresser plus régulièrement les communications. Enfin le dernier, *Funneled*, est la version *dégradée* du schéma précédent pour rendre le solveur plus portable. Un thread de communication par nœud se charge de toutes les communications, envois et réceptions, pour permettre l'utilisation des bibliothèques `MPI` qui ne fournissent pas le support complet des threads. Nous allons voir dans cette section l'évolution des performances du solveur PaStiX sur le temps de factorisation avec ces différents modèles de communication.

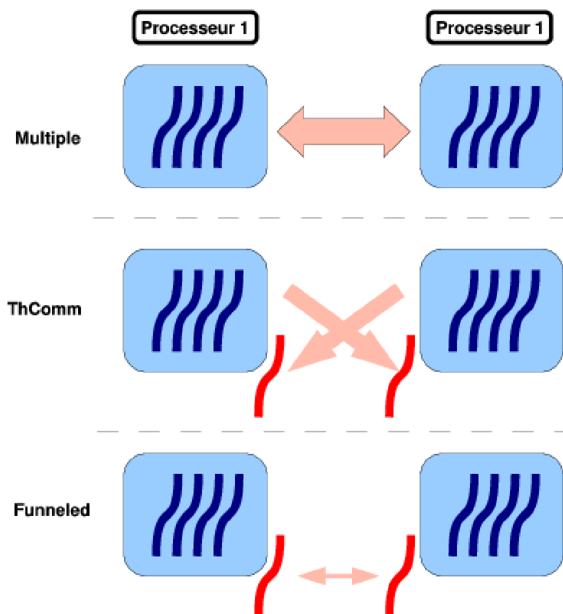


Fig. 3.3 – Schémas explicatifs des différents modes de communication mis en place dans le solveur PaStiX. *Multiple* est la version d'origine : tous les threads communiquent vers tout le monde. *ThComm* est la version avec un thread de progression en réception. *Funneled* est la version dégradée avec un thread gérant l'ensemble des communications.

Pour étudier cette évolution, nous présentons les temps de factorisation sur quatre de nos cas tests les plus importants en temps de calcul : Matr6, 3DSpectralWave, Matr5 et Audi avec deux clusters différents. Nous avons étudié la scalabilité des différents modèles de communication avec un nombre de threads variant de 1 à 16 et un nombre de processus MPI variant de 2 à 8. Dans chaque cas, nous avons placé un seul processus MPI par nœud. Les versions avec un thread supplémentaire sont favorisées lorsque le nombre de threads de calcul est inférieur au nombre de cœurs présents sur la machine car le thread de communication est exécuté en parallèle du ou des threads de factorisation sur les cœurs de calcul disponibles. Lorsque la machine est chargée avec autant de threads de calcul que de cœurs, le thread de communication n'est exécuté que lors des changements de contexte du système sur n'importe quel cœur de la machine. Cette solution permet de conserver une bonne réactivité et de ne pas dégrader les performances des routines

BLAS qui sont généralement dans des sections protégées de ces changements de contexte.

Les deux premiers tableaux 3.1 sont les résultats obtenus sur le cluster Decrypthon. Ce cluster utilise un réseau Fédération qui est un réseau propriétaire IBM, avec une implémentation MPI spécifique à l'utilisation de ce réseau.

TAB. 3.1 – Étude de la scalabilité du solveur PaStiX sur le temps de factorisation en secondes en fonction du schéma de communication utilisé sur le cluster Decrypthon. La librairie MPI utilisée est la version propriétaire d'IBM sur réseau Fédération.

Nb. Threads			Matr6					3DSpectralWave				
			1	2	4	8	16	1	2	4	8	16
Nb. Processus	1	Sequentiel	2110	1100	553	292	178	8260	4260	2104	1150	573
	2	Multiple	1140	601	307	191	102	4400	2320	1290	724	367
		ThComm	**1090**	**552**	**284**	**161**	**101**	**4130**	2110	**1090**	**569**	**315**
		Funneled	1100	583	330	234	179	4140	**2090**	1130	621	446
	4	Multiple	672	368	195	108	61,4	2560	1300	654	352	190
		ThComm	**557**	**292**	**152**	**87,4**	**59,8**	2220	**1110**	**575**	**300**	**189**
		Funneled	593	346	279	236	226	**2090**	1180	684	576	481
	8	Multiple	483	213	130	68,9	56	1510	715	**359**	**195**	**131**
		ThComm	**353**	**179**	**101**	**64,4**	**44,5**	1270	**650**	474	197	144
		Funneled	389	313	291	288	272	**1160**	752	632	487	445

Nb. Threads			Matr5					Audi				
			1	2	4	8	16	1	2	4	8	16
Nb. Processus	1	Séquentiel	2100	1160	592	297	161	1320	705	369	188	100
	2	Multiple	1160	608	339	213	124	714	409	205	118	68,7
		ThComm	**1100**	**562**	**282**	**151**	**99,6**	679	378	197	**99**	**67,3**
		Funneled	1110	594	335	243	219	**672**	**359**	**195**	123	113
	4	Multiple	686	352	210	117	**60,3**	449	209	121	**62,1**	35,6
		ThComm	**565**	**297**	**186**	**107**	65	**355**	**184**	**107**	68	**33,6**
		Funneled	602	383	307	302	268	364	203	136	127	131
	8	Multiple	470	221	**113**	63,9	51,6	237	124	**64,7**	**31,9**	**21,8**
		ThComm	**300**	**193**	116	**55,7**	**45,9**	**195**	**114**	67,5	42,5	23,2
		Funneled	377	316	350	362	385	204	153	131	138	150

La première chose que l'on remarque grâce à ces résultats est que, lorsqu'il n'y a qu'un seul thread de calcul, le surcoût de l'utilisation de la version *Multiple* peut être très important. Quelque soit le cas test étudié et le nombre de processus lancés, on observe un surcoût allant de 5 à 50% sur certains cas particuliers. Cependant, il disparaît rapidement avec l'augmentation du nombre de threads de calcul par processus. Une partie de cette augmentation du temps total dans la version *Multiple* correspond au temps d'ajout des contributions reçues des autres processeurs. Lorsque les nouveaux schémas de communication sont utilisés, ces calculs sont réalisés en parallèle de la factorisation. Le surcoût de la version *Multiple* est ici encore plus présent car on ne place qu'un processus par nœud. Le thread de progression dispose alors de la totalité du temps de calcul des autres cœurs disponibles. Cet écart se réduit avec l'augmentation du nombre de threads de calcul car les ajouts de contribution se retrouvent également parallélisés dans la version *Multiple*. Dans certains cas, la version *ThComm* devient moins performante que la version *Multiple* comme pour le calcul de la matrice 3DSpectralWave car la gestion

des communications est trop sérialisée et devient coûteuse par rapport au temps moyen du calcul d'une tâche élémentaire. Le second point important qui ressort de ces tableaux est la perte de performance importante due à l'utilisation de la version *Funneled*. Il semble que la bibliothèque MPI d'IBM ne permet pas de faire progresser les communications correctement dans cette configuration. Globalement, les versions *Multiple* et *ThComm* donnent les meilleurs résultats et passent à l'échelle de manière similaire. La version *ThComm* est même globalement meilleure que la version *Multiple*.

Les deux tableaux 3.2 suivants présentent les résultats sur la machine Titane du CCRT. La bibliothèque MPI utilisée est la version 3.2.1.009 d'Intel sur réseau Infiniband.

Tab. 3.2 – Étude de la scalabilité du solveur PaStiX sur le temps de factorisation en secondes en fonction du schéma de communication utilisé sur le cluster Titane. La librairie MPI utilisée est la version Intel sur réseau Infiniband.

Nb. Threads			Matr6				3DSpectralWave			
			1	2	4	8	1	2	4	8
Nb. Processus	1	Séquentiel	1040	532	295	144	4190	2130	1110	563
	2	Multiple	547	276	144	91,3	3290	1570	630	304
		ThComm	**518**	**275**	**143**	**75,3**	**2090**	**1060**	**560**	**285**
		Funneled	520	276	145	77,2	**2090**	**1060**	563	286
	4	Multiple	351	146	82,1	51,9	1510	667	385	179
		ThComm	**271**	**135**	**76**	**43**	**1030**	**539**	**284**	**151**
		Funneled	273	137	82,2	50,9	1040	540	288	157
	8	Multiple	217	97,8	49,5	29,4	935	371	200	101
		ThComm	**134**	**71**	**51,9**	**25,3**	**538**	**275**	**151**	**84,3**
		Funneled	137	80,8	54,5	35,1	540	282	159	94,8

Nb. Threads			Matr5				Audi			
			1	2	4	8	1	2	4	8
Nb. Processus	1	Séquentiel	1110	572	307	159	610	320	169	86,9
	2	Multiple	583	290	**158**	80,5	581	258	120	50,3
		ThComm	**552**	**288**	**158**	**80,1**	**305**	**157**	**87,5**	**43,8**
		Funneled	553	289	159	81,6	306	158	88,4	46,6
	4	Multiple	366	162	85,1	**55,2**	299	114	50,1	25,9
		ThComm	286	**148**	**80,1**	56,7	**154**	**82,9**	**44,3**	**24,3**
		Funneled	**285**	151	87,8	56,8	156	84,7	48,5	31,7
	8	Multiple	214	96	**55**	31,7	134	57	27,4	16,6
		ThComm	**151**	**75,5**	57,1	**27,8**	**80,3**	**41,7**	**24,1**	**14,6**
		Funneled	159	86,9	56,9	38,4	91,5	49,1	39,1	25,5

Ces tableaux confirment le surcoût déjà observé sur la version *Multiple* lorsqu'il n'y a qu'un seul thread de calcul, même si la différence est moins importante que sur la machine Decrypthon. On constate cependant ici que les trois versions du solveur passent correctement à l'échelle même avec la version *Funneled* contrairement aux résultats obtenus sur la machine IBM avec la bibliothèque MPI propriétaire. Sur cette architecture, comme sur Decrypthon, la version *ThComm* donne les meilleurs résultats sur l'ensemble des configurations et des matrices utilisées. Cependant, ces résultats ont été obtenus en modifiant les paramètres par défaut de la bibliothèque IntelMPI et notamment la fréquence de scrutation sur la carte Infiniband. La fréquence par

défaut limitait la réception des données surtout quand on ne disposait que d'un seul thread de progression. Les temps de factorisation étaient alors doublés sur certaines matrices avec les configurations à huit processus de huit threads chacun pour les versions *ThComm* et *Funneled*.

Pour remédier au problème éventuel de la sérialisation des communications ou pour multiplier le nombre de threads scrutant la carte réseau, la version *ThComm* permet d'utiliser plusieurs threads de communications pour faire progresser en parallèle plusieurs réceptions. Des tests ont montré qu'il était trop coûteux d'utiliser autant de threads de progression que de threads de calcul en les dédiant à la progression des communications associées à chacun d'eux. Il est préférable de mettre les threads de progression en réception sur des communications "*ANY_SOURCE*".

Une autre approche des versions *Funneled* et *ThComm* consiste à dédier un cœur de calcul à ces threads de communication pour qu'ils puissent être les plus réactifs possible. Il ne reste alors que $n-1$ cœurs de calcul utiles. Cette solution s'est révélée inefficace. Par exemple, sur le cluster Decrypthon qui possède 16 cœurs par nœud, les temps de factorisation étaient en moyenne supérieurs de $16/15^{ème}$ aux temps de calcul présentés dans les tableaux 3.1. Cette différence obtenue correspond aux ressources de calcul retirées à la factorisation. Le résultat est logique car le coût de calcul de l'ajout des contributions est faible contrairement au coût de factorisation. De plus, les méthodes directes concentrent de façon intrinsèque les communications sur le haut de l'arbre d'élimination, elles arrivent donc à la fin de la factorisation. Dans cette version, le cœur dédié au thread de communication reste inactif environ 80% du temps. Cette inactivité se retrouve sur le temps total d'exécution.

La version *ThComm*, introduite pour permettre le développement d'un ordonnancement dynamique dans le solveur PaStiX et pour faire progresser plus régulièrement les communications, donne de très bons résultats. L'objectif qui était d'obtenir une version dont l'efficacité était au moins similaire à celle de la version *Multiple* est atteint et même dépassé, puisque cette version est la plus performante dans la majorité des cas. Les résultats montrés dans la suite de nos évaluations seront basés sur cette version *ThComm*.

Comme les développements présentés dans le chapitre précédent sur la méthode d'allocation, ces deux nouveaux schémas de communications, également utilisables dans la version avec ordonnancement statique, peuvent se généraliser à d'autres codes utilisant une programmation hybride MPI/Thread. Ces développements sont intégrés dans la dernière version du solveur PaStiX (PaStiX 5.1.2 - Sleeper), mais la version utilisée par défaut reste actuellement la méthode *Multiple*. La méthode utilisant un thread de progression des communications sera désormais activée par défaut dans les versions à venir. On peut basculer d'un modèle de communication à un autre par des options de compilation que nous présentons dans l'annexe B.

Chapitre 4

Algorithme d'ordonnancement dynamique pour architectures NUMA

Sommaire

4.1 Description de la solution statique 56
4.1.1 Repartitionnement et affectation des processeurs candidats 56
4.1.2 Algorithme de distribution et d'ordonnancement 59
4.2 Description de la solution dynamique 61
4.2.1 Nouvel algorithme de distribution des données 62
4.2.2 Ordonnancement dynamique des calculs 64
4.3 Validation 69
4.3.1 Version en mémoire partagée 69
4.3.2 Version hybride MPI/Threads 70

Nous avons vu dans le chapitre 1 que peu de bibliothèques de résolution de systèmes linéaires par méthodes directes sont adaptées à l'utilisation d'architectures NUMA. Le problème de ces architectures comme nous l'avons montré est la dissymétrie des accès mémoire selon la localité des données. Ces accès mémoire non-uniformes sont très difficiles à modéliser pour les intégrer dans les modèles de coûts et notamment dans ceux du solveur PaStiX. La simulation réalisée pour calculer l'ordonnancement statique du solveur est alors faussée par ces effets et de nouveaux temps d'inactivité apparaissent lors de la factorisation numérique.

Pour remédier aux imperfections du modèle de coût utilisé, nous proposons de le corriger par un ré-ordonnancement dynamique des calculs lors de la factorisation. Pour cela, on utilise la prédiction statique pour distribuer efficacement les données sur les nœuds de calcul et obtenir un premier ordonnancement des calculs sur les cœurs que l'on corrigera par la suite dynamiquement.

Les deux chapitres précédents ont amené les bases à ces modifications que l'on souhaite apporter au solveur PaStiX. Le chapitre 2 a introduit la nouvelle méthode d'allocation des données qui permet de mieux répondre au problème de placement des données sur la machine. Le chapitre 3, lui, a présenté les développements effectués sur les schémas de communication du solveur. Ces développement ont été réalisés afin d'être plus réactifs dans la gestion des

communications dans notre version dynamique, mais également pour permettre une meilleure diffusion du solveur.

Dans ce chapitre nous allons présenter dans la première section 4.1, l'algorithme de distribution des données et de simulation de la factorisation aboutissant à l'ordonnancement statique des calculs. Nous verrons ensuite dans la deuxième section 4.2 comment nous nous sommes appuyés sur cet algorithme pour repenser la distribution des données de façon à ce qu'elle s'adapte mieux aux architectures NUMA. Ceci nous amènera à l'algorithme de vol de travail implémenté dans le solveur PaStiX. Nous verrons pourquoi et comment nous avons exploité l'arbre d'élimination du problème plutôt qu'un graphe direct acyclique (DAG) utilisé dans d'autres bibliothèques comme PLASMA [40] ou HSL_MA87 [63].

Les évolutions apportées seront ensuite évaluées sur l'ensemble de nos cas tests et de nos machines. Dans un premier temps, nous analyserons les résultats en mémoire partagée pour voir l'apport et/ou le surcoût de l'ordonnancement dynamique. Puis nous validerons les algorithmes en version distribuée avec l'utilisation du thread de progression des communications présenté dans le chapitre précédent.

4.1 Description de la solution statique

La difficulté pour obtenir une bibliothèque de résolution linéaire qui soit performante et qui passe à l'échelle efficacement lorsque la taille du problème croît est d'avoir une régulation équilibrée des calculs sur les ressources disponibles. Pour avoir une vision plus détaillée de ce problème on se reportera à la thèse de Pascal Hénon [59]. Le solveur PaStiX résout ce problème statiquement lors du pré-traitement pour ne pas avoir à le gérer dynamiquement lors des calculs. Cela permet d'avoir plus d'informations sur les données et leur dépendances que lorsqu'elles sont distribuées. On gagne ainsi du temps sur la prise de décision d'ordonnancement des calculs lors de la factorisation numérique. L'ordonnancement statique du solveur PaStiX repose sur une phase d'analyse en deux étapes. Celle-ci vient après les deux premières phases de prétraitement que sont la renumérotation de la matrice et la factorisation symbolique. Elle consiste à répartir les données sur les nœuds et calculer l'ordonnancement associé à cette distribution pour tirer le plus de parallélisme et le plus d'efficacité possible lors de la phase de factorisation numérique. La première étape que nous allons présenter est celle dite de répartition proportionnelle. Elle affecte à chaque bloc (ou bloc-colonne) un ensemble de threads candidats pour effectuer les calculs associés à ce bloc (respectivement bloc-colonne). La deuxième est celle de distribution et d'ordonnancement. Elle place définitivement les blocs de données sur chacun des processus et détermine l'ordre dans lequel vont se faire les calculs.

4.1.1 Repartitionnement et affectation des processeurs candidats

La première étape de la phase d'analyse ne cherche pas à calculer la distribution finale des données mais seulement à fournir une première idée de ce que sera cette distribution. On calcule en réalité pour chaque bloc-colonne un ensemble des meilleurs candidats possibles pour le factoriser de façon à minimiser les communications. Cette phase de prétraitement repose sur deux éléments : une bonne utilisation de l'arbre d'élimination (plus il est large, plus il fournit des branches indépendantes sans intercommunications), et une bonne modélisation temporelle des opérations les plus importantes dans la factorisation. Ces opérations sont les suivantes :

- la factorisation LL^T, LDL^T ou LU d'un bloc dense pour les blocs diagonaux ;
- la résolution d'un système triangulaire multi seconds membres pour la résolution des équations des lignes 4 et 5 de l'algorithme séquentiel 1 page 12 ;

– la multiplication de deux matrices pour le calcul des contributions ;
– l'ajout de deux matrices pour la mise en place des contributions.

Ce modèle de coût temporel permet de pondérer chaque nœud de l'arbre d'élimination par le temps nécessaire à sa factorisation. À partir de cet arbre d'élimination pondéré, on réalise récursivement une répartition proportionnelle de l'ensemble des threads candidats sur les super-nœuds. L'initialisation de l'algorithme commence par attribuer tous les threads de calculs comme candidats à la factorisation du super-nœud racine de l'arbre d'élimination. Puis, chaque thread se voit attribuer une charge de calcul identique dont la somme correspond au coût de l'arbre complet. Une fois cette initialisation effectuée, récursivement en descendant dans l'arbre, le coût de calcul du nœud courant est déduit de façon équilibrée entre les processeurs candidats pour générer plus de parallélisme. Enfin, chaque sous-arbre du nœud courant se voit attribuer un sous-ensemble de candidats proportionnellement au coût de calcul de tout le sous-arbre et à la charge de calcul qu'il reste à pourvoir pour chacun des candidats disponibles à ce niveau. La figure 4.1 illustre cette répartition des processeurs candidats sur l'arbre d'élimination.

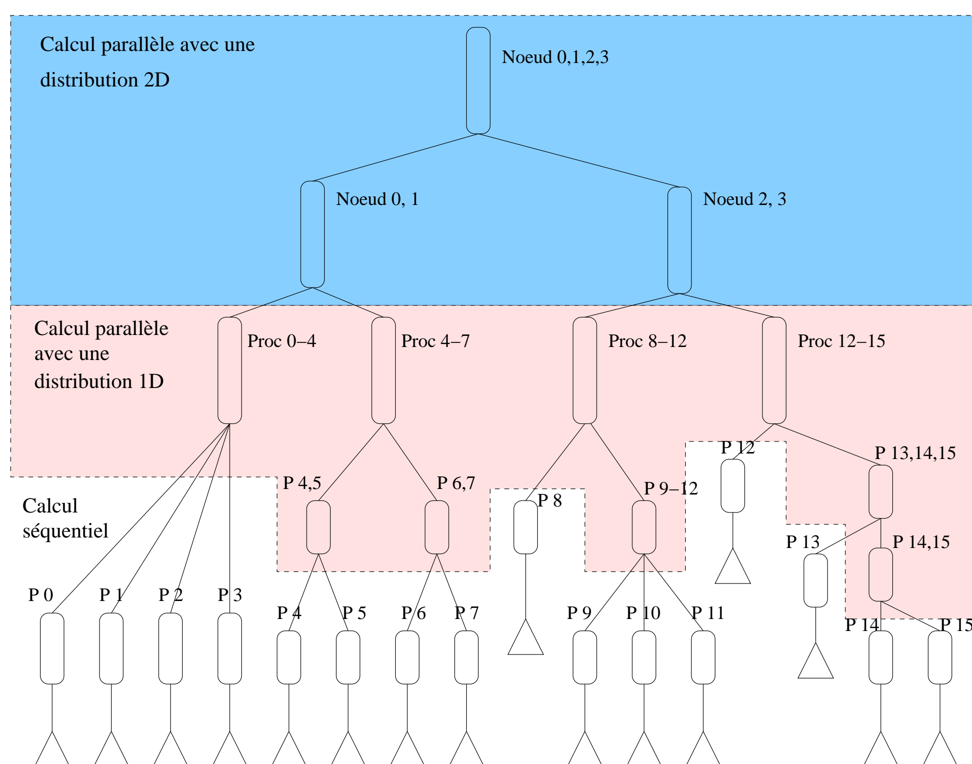


Fig. 4.1 – Répartition et processeurs candidats.

Lors du parcours de l'arbre par cet algorithme de répartition, les blocs-colonnes sont découpés à une taille optimale pour l'utilisation des routines BLAS sur l'architecture cible. Cela permet d'augmenter le parallélisme à grain fin décrit dans la section 1.6.1 page 23. Ce découpage est fait principalement sur les nœuds auxquels on a attribué plusieurs processeurs candidats. Ce découpage permet également d'ajuster le grain de calcul en fonction du nombre de processeurs candidats. Deux types de tâches de calcul sont utilisés : des tâches de calcul correspondant à un repartitionnement en blocs-colonnes du super-nœud (distribution 1D), et les tâches de calcul associées aux calculs élémentaires sur les blocs du super-nœud (distribution 2D). Une distribution 1D privilégie les effets BLAS sur les calculs grâce aux données organisées par blocs-colonnes. En

effet, que ce soit avec la méthode d'allocation initiale ou avec la nouvelle méthode d'allocation présentée dans le chapitre 2, les données sont allouées de manière continue par bloc-colonne. On peut ainsi faire un seul appel aux routines BLAS directement sur tout le bloc-colonne et non une multitude d'appels bloc par bloc sur des données. Au contraire, une distribution 2D est plus extensible à un nombre élevé de processeurs grâce à son grain de calculs plus fin et à la possibilité de mieux recouvrir les communications par les calculs. La figure 4.1 montre le repartitionnement des super-nœuds effectué lors de la distribution des processeurs candidats. Plus les super-nœuds sont hauts dans l'arbre d'élimination, plus ils sont potentiellement coûteux à calculer et donc plus on va utiliser une distribution à grain fin. On observe généralement que le nombre de processeurs candidats et la répartition de ces candidats sur plusieurs nœuds est un bon critère pour passer d'une distribution à une autre.

L'étape d'attribution des processeurs candidats est caractérisée par deux particularités. La première est de permettre à un processeur de participer à la factorisation de plusieurs sous-arbres adjacents pour éviter les problèmes d'arithmétique entière que l'on rencontre traditionnellement dans les méthodes de *proportionnal mapping*. Le processeur dédoublé (comme les processeurs 4 et 12 sur la figure 4.1) est candidat dans chacun des sous-arbres dans des proportions qui permettent d'équilibrer le temps de calcul de tous les candidats. Il peut arriver alors que ce processeur dédoublé ne soit candidat que pour une partie des super-nœuds et ne soit pas candidat jusqu'en bas de ce sous-arbre. En effet, la charge de travail qu'on lui a attribuée dans cette partie de l'arbre peut s'avérer trop faible pour qu'il participe au calcul de l'intégralité du sous-arbre. Cette solution permet d'éviter les problèmes des méthodes de *proportionnal mapping* mais c'est une très mauvaise solution pour les architectures NUMA car on ne conserve pas l'affinité mémoire entre les tâches définie par l'appartenance à une même branche de l'arbre.

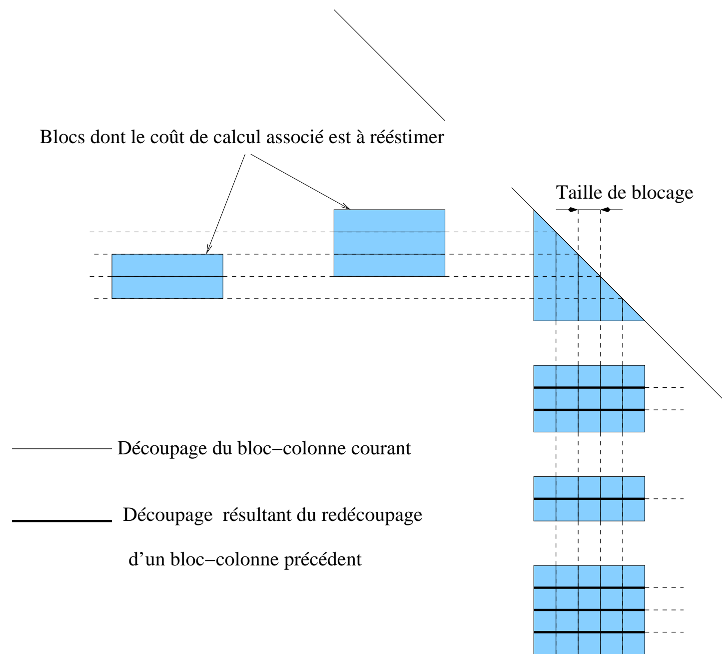


Fig. 4.2 – Découpage d'un bloc-colonne.

L'autre particularité de cet algorithme est la correction des coûts de calcul des super-nœuds. En effet, lors du découpage des super-nœuds avec une distribution 1D ou 2D, le coût de factori-

sation de l'ensemble des tâches est changé. Le découpage des blocs-colonnes implique également le découpage des blocs extra-diagonaux en regard du bloc diagonal du bloc-colonne découpé (voir figure 4.2). Ces blocs extra-diagonaux appartiennent à des blocs-colonnes associés à des super-nœuds qui se situent dans les sous-arbres du nœud courant et ils seront rencontrés en descendant dans l'arbre d'élimination par blocs. Le coût des opérations élémentaires BLAS n'est pas linéaire par rapport à la taille des blocs. Afin de préserver une bonne répartition des ensembles de processeurs candidats, il est nécessaire de conserver une bonne modélisation des coûts des sous-arbres et donc de corriger le coût de ceux-ci lorsqu'ils sont repartitionnés. La charge de calcul à attribuer restante doit également être ajustée linéairement pour coller au nouveau coût du sous-arbre.

Á la fin de la répartition, le graphe des tâches de calculs élémentaires avec leurs dépendances est construit. Un ensemble de processeurs candidats est attribué à chaque super-nœud de l'arbre d'élimination par blocs en respectant un bon équilibrage de la charge de travail sur chaque processeur. Cette charge de travail est approximativement la même pour chaque processeur, c'est-à-dire le rapport du coût total sur le nombre de processeurs.

4.1.2 Algorithme de distribution et d'ordonnancement

Nous allons maintenant décrire la deuxième étape de la phase d'analyse qui permet d'aboutir à l'ordonnancement statique utilisé dans le solveur PaStiX. Cette phase a pour but de distribuer et d'ordonnancer les tâches de calcul. Par conséquent, les blocs ou blocs-colonnes associés sont également distribués en même temps que la tâche de calcul. La distribution des données est finalement guidée par la simulation en temps de la factorisation numérique. On conserve tout de même une bonne scalabilité mémoire du code grâce à l'étape de répartition proportionnelle précédente. En effet, en distribuant de manière équilibrée la charge de calcul, la quantité de données est également efficacement répartie. Pour obtenir un bon ordonnancement, deux problèmes sont à résoudre. On doit pouvoir tenir compte du fait qu'une tâche de calcul peut être plus critique qu'une autre si elle permet de débloquer d'autres tâches plus rapidement. Dans le cas de notre arbre de dépendances, la relation d'ordre entre les tâches est donnée par la profondeur du nœud dans l'arbre d'élimination. Ainsi, plus la profondeur d'une tâche est importante, plus elle sera critique pour la factorisation. Le deuxième problème à résoudre dans l'ordonnancement de tâches sur une machine distribuée est la gestion des communications. C'est ce qui constitue l'avantage de la méthode statique utilisée dans le solveur PaStiX. L'ordonnancement des communications est guidé par la distribution des calculs et non l'inverse. Le calcul statique de l'ordonnancement des tâches induit naturellement l'ordonnancement des communications associées.

Le solveur PaStiX calcule l'ordonnancement des tâches et par conséquent la distribution des données à l'aide d'une simulation temporelle de l'étape de factorisation numérique. Ceci permet, lorsqu'une tâche de calcul devient prête au sens de la simulation, de choisir *lequel de ses processeurs candidats permettra de la traiter au plus vite*. Pour cela, on distingue quatre types de tâches :

- pour les données distribuées en 1D par bloc-colonne :
  - ***Tâche1D*** qui englobe tous les calculs nécessaires à la factorisation d'un bloc-colonne : factorisation du bloc diagonal, résolution du système triangulaire sur le bloc-colonne (et sur le bloc-ligne en LU) et calcul et report des contributions générées.
- pour les super-nœuds distribués en 2D, on utilise trois tâches différentes :
  - **Diag** est la factorisation du bloc diagonal (ligne 2 de l'algorithme séquentiel 1 page 12).
  - **TRSM** est la résolution du système triangulaire associé à chaque bloc extra-diagonal

(lignes 4 et 5). Les résolutions des blocs symétriques de L et U sont associées dans la même tâche.

– **GEMM** est celle associée au calcul des contributions générées par chaque couple de blocs extra-diagonaux (ligne 9).

Chacune de ces tâches a un coût de calcul et est associée à un bloc-colonne ou un bloc de la matrice. Son ordonnancement sur un de ses processeurs candidats va également définir la distribution des données associées.

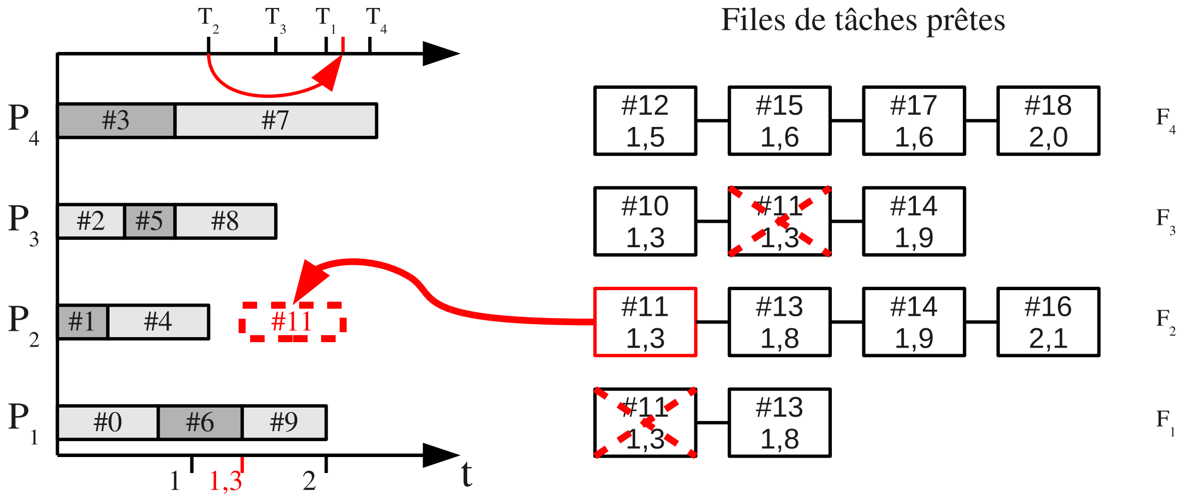


Fig. 4.3 – Ordonnancement des tâches prêtes sur les processeurs candidats.

La simulation en temps de la factorisation numérique se base sur plusieurs informations comme le type des tâches, l'arbre d'élimination et les processeurs candidats calculés lors des étapes précédentes du prétraitement. L'algorithme utilisé ensuite sur ces données est une heuristique de liste. On attribue à chaque processeur P_i deux données. Un *timer* T_i pour connaître le temps de calcul qu'on lui a déjà attribué et une file de tâches F_i qui contient toutes les tâches dont il est candidat à la factorisation et qui sont prêtes à être factorisées. Ceci est représenté sur la figure 4.3. Les tâches présentes dans les files F_i sont triées grâce à un compteur de temps qui détermine l'instant où elles ont reçu toutes leurs contributions. Le deuxième critère de tri des ces tâches est leur chemin critique qui est donné par leur niveau dans l'arbre d'élimination. Plus une tâche est loin du sommet de l'arbre, plus elle est critique. L'algorithme consiste à prendre à chaque itération l'association processeur/tâche qui peut s'exécuter au plus tôt et dont la tâche est la plus critique. Sur la figure 4.3, la tâche #11 sur le processeur P_2 est l'association choisie.

Le choix de ces associations à chaque itération permet de distribuer les données associées à chaque tâche sur le processeur choisi pour son calcul. Cette distribution permet par la suite de déterminer l'instant où les tâches suivantes sont disponibles en fonction de la localité des tâches qui contribuent à son calcul. Le calcul de la disponibilité des nouvelles tâches se fait suivant le schéma de la figure 4.4. Les contributions locales mettent à jour le compteur directement car elles sont ajoutées immédiatement. Mais les contributions distantes sont préalablement agrégées avant d'être envoyées au processeur qui sera en charge de calculer la tâche. Ceci nécessite de calculer avec la simulation à quel moment les contributions de chaque processeur sont prêtes, ce qui est calculé à l'aide du modèle de coût des communications qui détermine le moment où la tâche est prête. Les tâches pour lesquelles plusieurs processeurs sont candidats se retrouvent ainsi dans les files de ces candidats avec des timers indiquant leur disponibilité qui ont des

valeurs différentes suivant les processus.

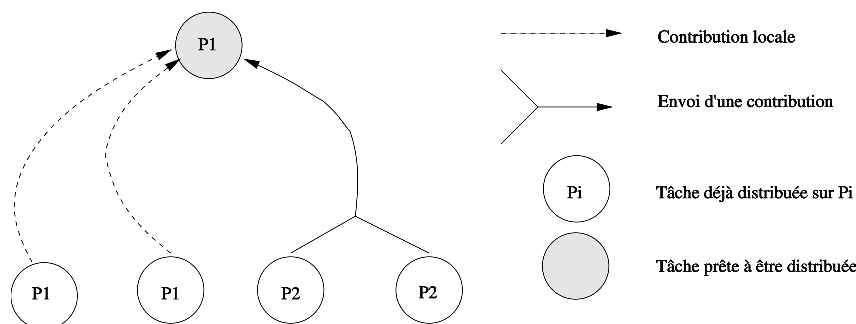


FIG. 4.4 – Schéma de communication si la tâche est distribuée sur P_1.

Le procédé est ainsi répété tant qu'il reste des tâches dans les files F_i. Quand l'algorithme est terminé, toutes les tâches sont distribuées sur les processeurs ou threads disponibles. L'ordre dans lequel les tâches ont été affectées à chacun des processeurs constitue l'ordonnancement statique qui va être exploité lors de la factorisation numérique. Cet ordonnancement et cette distribution induisent également l'ordonnancement des communications qui est fortement couplé avec celui des calculs. Cet ordonnancement calculé avec les modèles de coût des routines BLAS n'est pas adapté à l'utilisation des architectures NUMA. Nous souhaitons le corriger en réordonnançant les calculs dynamiquement au sein des nœuds pour pallier les défauts qui peuvent apparaître dans ces modèles.

4.2 Description de la solution dynamique

L'ordonnancement statique calculé dans le solveur PaStiX et décrit dans la section précédente est très efficace sur de nombreux problèmes et architectures. Son efficacité repose sur l'utilisation des modèles de coûts utilisés dans les thèses de Pascal Hénon et Pierre Ramet [59, 82] qui ont été mis au point sur une architecture IBM Power4. Les calculs nécessaires pour obtenir une modélisation fine des routines BLAS sur une plate-forme d'exécution est un travail long et fastidieux. Pour éviter aux utilisateurs du solveur de devoir faire ces modélisations, le solveur PaStiX utilise sur toutes les architectures le modèle de coûts plus récent calculé sur Power5. Malgré cette erreur introduite dans la modélisation des calculs effectués, on constate que le solveur reste efficace sur toutes les architectures utilisées jusqu'à maintenant comme les Opterons, les Power5 ou 6, les Xeons ou les Itaniums. Ceci est obtenu grâce aux deux étapes décrites précédemment et au couplage qui existe entre ces étapes de prétraitement et la factorisation numérique. Cependant, on peut penser que ces erreurs introduites dans la simulation conduisent à un ordonnancement qui n'est pas optimal. L'ordonnancement dynamique que nous souhaitons mettre au point est destiné à corriger les facteurs NUMA qui ne sont pas intégrés dans les modèles de coût et les accès distants dûs à la distribution statique initiale qui permet à une partie des threads candidats de travailler dans plusieurs branches parallèles de l'arbre. En effet, ceux-ci induisent des différences de temps d'accès et donc des différences d'exécution importantes comme on a pu le voir dans le chapitre 2 page 29. Le but de ce nouvel ordonnancement est de corriger dynamiquement lors de la factorisation les erreurs qui ont été introduites par le modèle de coûts aussi bien que les erreurs dues à une activité extérieure qui peut ralentir un ou plusieurs cœurs de calcul. Pour cela, nous nous basons sur deux étapes. La première est faite lors du prétraitement : elle consiste à redistribuer les tâches de calcul au sein des nœuds pour

prendre en compte le mieux possible l'affinité mémoire entre les tâches de calcul pour éviter le surcoût des effets NUMA. La seconde étape que nous présenterons est la modification de la boucle principale de calcul pour s'adapter à l'ordonnancement dynamique. Dans la suite de ce chapitre, nous nous baserons uniquement sur une distribution statique en 1D des données calculée à partir du modèle de coût Power5 du solveur PaStiX.

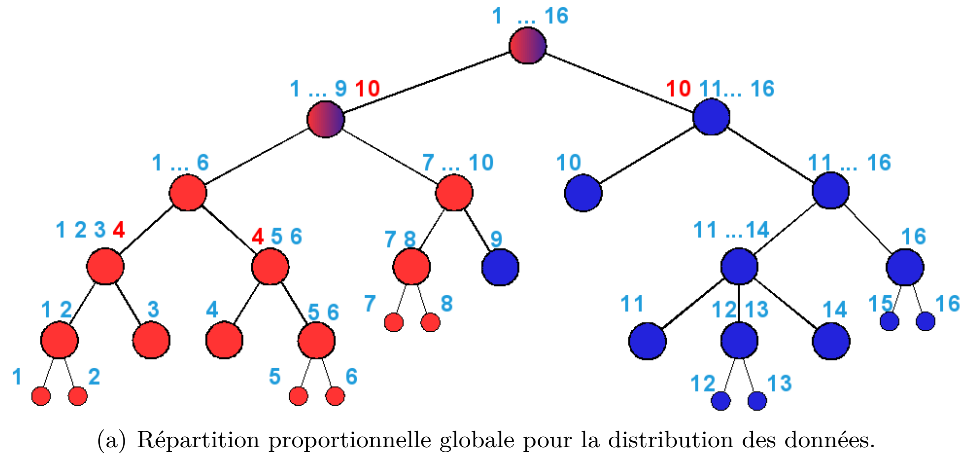
4.2.1 Nouvel algorithme de distribution des données

On souhaite ordonnancer dynamiquement les calculs internes à un nœud, c'est-à-dire entre les threads, et non entre les processus MPI, pour ne pas avoir à gérer dynamiquement la distribution des données de la matrice. Pour cela, on conserve les deux étapes de l'ordonnancement statique décrites dans la section 4.1 page 56. Ces deux étapes vont nous permettre de distribuer efficacement les blocs de la matrice sur les nœuds du cluster grâce aux modèles de coûts utilisés dans l'ordonnancement statique.

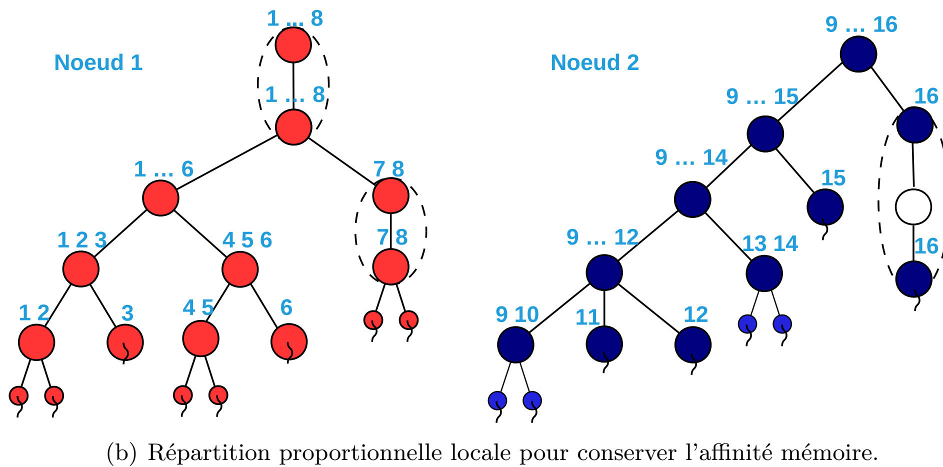
Une fois la distribution calculée, la solution la plus simple pour ordonnancer dynamiquement les calculs est celle utilisée dans la plupart des codes à ordonnancement dynamique. Toutes les tâches sont ajoutées dans une file de tâches prêtes dès qu'elles ont reçu toutes leurs contributions, puis les threads de calcul viennent chercher du travail dans cette file quand ils ont fini la tâche qu'ils avaient en cours. Cette solution est efficace sur des nœuds de calcul SMP, donc à accès mémoire uniformes, avec un faible nombre de cœurs. Cependant, elle devient assez rapidement non scalable avec un nombre de cœurs important. En effet quand le nombre de cœurs devient trop important, la protection nécessaire de la file est alors trop coûteuse car elle sérialise les accès à cette file.

Pour résoudre ce problème, une solution consiste à créer une file par thread de calcul et à avoir un thread supplémentaire qui répartit les tâches de façon équilibrée entre ces files. Cette solution passe mieux à l'échelle mais ne prend pas en compte les accès mémoire non uniformes des architectures NUMA. Pour cela, il faut guider l'orientation des tâches dans la bonne file en fonction de la localité des données qui lui sont associées par rapport au thread à qui on fournit la tâche. Dans notre cas, nous voulons résoudre le problème inverse : on veut orienter le choix de la file dans laquelle les threads vont trouver du travail, et ne pas avoir à déterminer dynamiquement dans quelle file doit aller chaque tâche. Pour cela, nous nous basons sur la structure de l'arbre d'élimination et sur le fait que chaque branche de l'arbre est indépendante. Il n'existe pas de communications d'une branche à l'autre mais seulement vers des nœuds plus haut dans l'arbre. Pour conserver l'affinité mémoire entre les tâches, il faut alors conserver le plus possible les branches de l'arbre sur un même thread de calcul.

L'algorithme de répartition proportionnelle utilisé dans le calcul de l'ordonnancement statique permet de répartir équitablement les branches de l'arbre sur les processeurs disponibles. Pour résoudre notre problème, nous avons choisi d'appliquer deux fois cet algorithme sur l'arbre d'élimination. La première passe de l'algorithme va nous permettre, avec la simulation temporelle de la factorisation, d'obtenir la distribution des données par processus. Pour cela, on applique la répartition proportionnelle sur l'arbre complet avec le nombre total de threads disponibles pour le calcul du problème. Cependant, pour avoir plus de parallélisme et alimenter plus régulièrement les files de tâches, nous avons ajusté le découpage des blocs-colonnes effectué lors de cette phase. Nous avons vu dans la section 4.1.1 page 56 que les blocs-colonnes sont découpés lors du parcours de l'arbre pour avoir le meilleur grain de calcul qui soit adapté au nombre de processeurs candidats. Nous avons choisi dans le cas de l'ordonnancement dynamique de continuer ce découpage pour les tâches associées à un seul candidat pour obtenir une meilleure réactivité



(a) Répartition proportionnelle globale pour la distribution des données.



(b) Répartition proportionnelle locale pour conserver l'affinité mémoire.

Fig. 4.5 – Schéma de la répartition proportionnelle des calculs en deux étapes.

sur l'algorithme d'ordonnancement dynamique.

Une seconde passe de l'algorithme de répartition proportionnelle modifié va permettre ensuite de répartir équitablement les calculs sur les processeurs tout en conservant l'affinité mémoire entre les données. Pour cela, on l'applique uniquement aux données qui ont été distribuées localement. Cette seconde passe sera moins coûteuse que la première car il ne s'agit plus d'un algorithme séquentiel sur l'ensemble des données mais chaque processus réalise en parallèle le calcul sur ses données locales. Dans un premier temps, il faut obtenir le sous-arbre *local* sur lequel on va exécuter la deuxième passe de l'algorithme de répartition proportionnelle. Chaque processus cherche donc à extraire de l'arbre d'élimination uniquement les nœuds dont un de ses threads doit factoriser les données associées ou une partie de ces données dans le cas des super-nœuds qui sont découpés sur plusieurs processus. Pour cela, on parcourt récursivement l'arbre d'élimination en recalculant le coût de chaque sous-arbre dans lequel on met à zéro le coût de calcul des tâches externes au processus. On obtient alors les arbres bleu et rouge de la figure 4.5(b) issus de l'arbre initial de la figure 4.5(a). Le fait que les processus puissent être candidats dans plusieurs branches différentes peut amener à avoir une forêt sur certains processus car les nœuds sont dupliqués dans tous les arbres locaux où le coût de calcul est non nul. On considérera par la suite qu'on est toujours dans le cas d'un arbre puisqu'ici les forêts peuvent être rassemblées avec des nœuds fictifs qui sont en réalité des tâches distribuées sur d'autres

processus. Il peut également arriver, comme sur l'arbre local du nœud 2 de la figure 4.5(b), que certains nœuds se voient attribuer des candidats provenant de différents processus lors de la première passe de l'algorithme de répartition proportionnelle qui attribuent les ensembles de candidats. Puis, lors de l'étape de simulation qui distribue les données sur les processus, c'est-à-dire qui affecte les couleurs aux nœuds, les nœuds de l'arbre d'élimination peuvent être partagés sur plusieurs processus (nœuds bicolores) ou n'être distribués que sur un seul processus (nœud vide). Dans ce cas, les nœuds sont conservés dans l'arbre local pour préserver les dépendances entre les calculs mais ils seront ignorés dans la deuxième passe de l'algorithme que nous allons décrire.

Une fois cet arbre calculé, on va effectuer une affectation des threads candidats sur les nœuds de calculs. On cherche ici à maximiser la localité des données en regroupant les tâches qui appartiennent à une même branche. Il n'est plus autorisé à un thread de participer au calcul de deux sous-branches dans lesquelles il y a plusieurs candidats. Il peut en revanche participer au calcul d'une branche avec d'autres threads et être également le seul candidat d'une branche voisine dont la charge de travail est suffisamment faible pour être attribuée complètement à un seul thread. Pour éviter les problèmes d'arrondis sur cette répartition, on trie désormais les fils d'un nœud par ordre décroissant du coût de calcul de leur branche. On commence ensuite par répartir les branches les plus importantes sur des groupes de threads disjoints, ce sont les branches dont la charge de calcul ne peut pas être assurée par un seul thread. Puis, on donne comme candidat à chaque branche restante qui peut être calculée par un seul thread, le thread dont la charge de travail déjà attribuée est la plus faible. On met ensuite à jour la charge de travail du thread choisi, et on recommence jusqu'à ce qu'il ne reste plus de fils dans le nœud courant à distribuer. Puis récursivement, on descend dans l'arbre pour attribuer les candidats pour toutes les tâches locales.

Quand tout l'arbre a été parcouru, on obtient une distribution des tâches de calcul qui répartit équitablement la charge et les données sur les cœurs de la machine. La méthode de construction conserve l'affinité mémoire entre les tâches. Comme pour la partie séquentielle de la figure 4.1 page 57, les sous-arbres du bas de l'arbre sont attribués à un seul candidat. Les données qui lui sont associées vont ainsi pouvoir être allouées par l'unique thread candidat à leur factorisation pour optimiser la localité des données comme on l'a vu dans le chapitre 2. Les super-nœuds qui sont plus haut dans l'arbre d'élimination par blocs représentent généralement plusieurs tâches et possèdent plusieurs candidats pour les exécuter. Nous avons choisi de répartir cycliquement les blocs-colonnes ou blocs associés à ces tâches sur les cœurs candidats lors de l'exécution. De cette façon, comme la localité topologique des données est définie par la numérotation des threads, on conserve les données sur une zone restreinte de la machine pour éviter les problèmes de contention et les surcoûts des effets NUMA. Nous allons voir désormais comment, lors de la factorisation numérique, nous exploitons ces informations et comment nous orientons le vol de travail de notre ordonnancement dynamique pour conserver l'affinité mémoire des données.

4.2.2 Ordonnancement dynamique des calculs

La répartition proportionnelle décrite précédemment fournit une distribution équilibrée des données sur les nœuds de calcul et définit les threads qui seront candidats à la factorisation de chaque tâche locale. L'ordonnancement statique utilisé auparavant donnait l'ordre dans lequel les calculs devaient être exécutés pour obtenir les meilleures performances selon la simulation temporelle de la factorisation numérique. Comme nous l'avons dit précédemment, cet ordon-

nancement peut être amené à avoir quelques défauts. Effectivement, les accès mémoire non-uniformes des nouvelles architectures faussent les modèles de coût utilisés pour la simulation mais les phénomènes de contention peuvent également jouer un rôle important dans cette mauvaise prédiction de l'ordonnancement. On a pu voir dans le chapitre 2 page 29 que les temps de calcul des routines BLAS peuvent être doublés si le placement des données utilisées est trop entrelacé. Ces perturbations non proportionnelles dans les coûts des tâches de calcul peuvent entraîner l'apparition de temps d'inactivité sur un ou plusieurs cœurs de la machine. Il est donc utile de pouvoir permuter l'ordonnancement d'une partie des tâches de calcul pour réduire les temps d'inactivité dans l'ordonnancement et améliorer l'efficacité du solveur, d'autant plus que nous avons introduit de nouvelles contraintes dans la distribution statiques des données dans la section précédente. Cependant, pour optimiser le placement mémoire et s'approcher du cas optimal de l'étude du chapitre 2, le choix des tâches à permuter doit être guidé par la localité des données qui lui sont associées. De cette façon, on veut limiter le plus possible l'apparition de contention sur la machine et le surcoût des accès mémoire.

Pour conserver l'affinité mémoire des tâches de calcul et guider le vol de travail qui permettra le réordonnancement les calculs, on utilise l'arbre d'élimination local précédemment calculé avec les ensembles de threads candidats attribués à chaque tâche. L'arbre d'élimination local est condensé sous la forme d'un arbre de files de tâches présenté dans la figure 4.6. Toutes les tâches dont l'ensemble des threads candidats est identique sont regroupées dans une même file. Ce sont les regroupements en pointillés sur la figure 4.5(b) page 63. Au sein de cette file, nous avons choisi de trier les tâches selon l'ordre dans lequel elles auraient été calculées par l'ordonnancement statique. L'arbre est ensuite construit suivant le principe suivant : chaque nœud est le fils de celui dont l'ensemble de threads candidats est le plus petit ensemble incluant le sien. Les feuilles de l'arbre sont les files de tâches associées à un seul thread candidat. Il y en a une par thread, soit une par cœur sur l'architecture cible. Ce sont par construction celles qui contiennent le plus de tâches de calcul. La racine de l'arbre est la file des tâches dont tous les threads du processus sont candidats à leur factorisation.

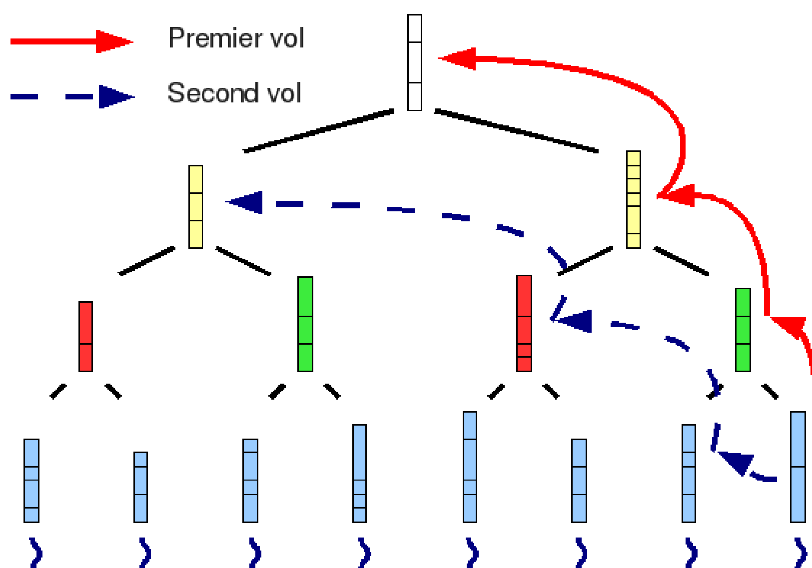


Fig. 4.6 – Arbre de vol de travail.

Dans la version avec ordonnancement statique du solveur, le tableau de tâches attribué à

chaque thread déterminait la localité des données associées, puisque dans la nouvelle méthode d'allocation c'est ce thread qui alloue et initialise ces données (cf section 2.3 page 37). Dans la version dynamique, nous fixons toujours chaque thread sur un cœur de calcul et nous allons nous baser sur cette structure pour déterminer le placement des données. Les tâches présentes dans les files des feuilles de l'arbre ont un seul thread *privilégié* pour leur factorisation. Les données de ces tâches sont logiquement allouées et initialisées par ce thread. Les données resteront ainsi au plus proche du thread de calcul. C'est pourquoi, lors de la construction de l'arbre, on cherche à remplir au maximum les feuilles de l'arbre. Ceci est fait grâce au tri par ordre décroissant du coût de calcul des branches de chaque nœud qui permet de placer celles dont le coût de calcul peut être pris en charge par un seul thread dans les feuilles de l'arbre de vol. On augmente ainsi le nombre de tâches où la localité mémoire est optimale. De plus ces files regroupent la majorité des tâches de calcul en nombre et surtout les tâches associées aux plus petits blocs-colonnes concentrés en bas de l'arbre d'élimination. Il est primordial que ces tâches soient correctement placées sur la machine pour éviter le surcoût des effets NUMA qui sont plus importants sur les matrices de petite taille où le rapport du coût de calcul sur le coût de transfert est très faible. En remontant dans l'arbre de vol de travail, il devient difficile de choisir quel thread doit initialiser les données car plusieurs threads sont désormais candidats à la factorisation, et le choix n'est plus déterminé à l'avance. Nous avons choisi d'attribuer l'initialisation des données cycliquement aux threads candidats. Ainsi, si les threads sont correctement placés, on limite la zone de contention à une faible sous-partie de la machine. Dans le cas de machines à processeurs dual-cœurs ou quad-cœurs, on conserve même les placements optimaux pour les premiers niveaux de l'arbre.

On considère que les threads de calcul sont correctement placés sur l'architecture si ceux-ci sont placés de proche en proche sur les cœurs de calcul de l'architecture. On peut voir sur les figures 1.6(a) et 1.6(c) page 17 que la numérotation des cœurs des clusters Borderline et Titane n'est pas linéaire. Il est donc important sur de telles architectures de prendre en compte cette numérotation pour placer correctement les données et limiter dans l'espace les communications entre les threads. On considère que les threads sont placés linéairement sur la machine si la numérotation des threads de calcul suit la structure de l'architecture et non la numérotation du système. Pour obtenir ce placement optimal des threads de calcul nous utilisons la bibliothèque HwLoc (Portable Hardware Locality) [8] développée par l'équipe Runtime et le projet OpenMPI. Elle permet de lire les informations fournies par le système et de récupérer simplement les informations voulues dans son code. Lorsque cette bibliothèque est disponible, on peut directement au sein du solveur PaStiX placer correctement les threads de calcul sur l'architecture. Dans le cas contraire, le placement des threads est défini soit linéairement selon la numérotation, soit grâce à un tableau passé en paramètre permettant de placer les threads sur l'unité de calcul voulue.

Le but lors de la factorisation numérique va être de limiter le vol de travail aux tâches qui ont été placées au plus proche de chacun des threads pour optimiser l'affinité mémoire et limiter au maximum les phénomènes de contention très coûteux sur les routines BLAS matrice/vecteur. On utilise pour cela un vol de travail à deux niveaux. Dans un premier temps, chaque thread prend du travail dans la file de tâches qui lui est attribuée. Puis, quand il n'a plus de travail, on parcourt les nœuds entre lui et la racine de l'arbre de vol de travail à la recherche de tâches disponibles. Ce sont les vols représentés par les flèches rouges continues sur la figure 4.6 et qui correspondent à la boucle des lignes 10 à 13 de l'algorithme 5. Enfin, si aucune tâche n'est

[8] Cette bibliothèque fait partie du projet OpenMPI et est la fusion des projets libtopology de l'équipe Runtime et Portable Linux Processor Affinity (PLPA) d'OpenMPI. `http://www.open-mpi.org/projects/hwloc`

disponible sur ce chemin, on parcourt les nœuds qui sont à une distance 1 de ce chemin, c'est-à-dire les fils des nœuds appartenant au chemin vers la racine. Ce sont les flèches bleues pointillées de la figure 4.6 et les lignes 20 à 31 de l'algorithme 5. De cette façon, on garantit que chaque thread exécute les tâches qui sont toujours au plus proche de lui ou dans un rayon limité sur la machine. Cette solution privilégie les accès mémoire sur le chemin critique.

Notations :

- F_f est la file numéro f de l'arbre de vol de travail. Les feuilles sont les premières numérotées, elles correspondent chacune à un thread de calcul ;
- La racine de l'arbre est la file f dont $Pere(f) = -1$;
- $T_{tot}(f)$ est le nombre total de tâches associées à la file F_f ;
- $T_{calc}(f)$ est le nombre de tâches associées à la file F_f qui ont déjà été calculées.

Algorithme 5 Algorithme d'ordonnancement dynamique des calculs sur le thread t

```
1: f = t
2: Tant que Vrai Faire
3:     Tant que T_calc(f) = T_tot(f) et f ≠ −1 Faire        /* On met à jour la file initiale */
4:         f = Pere(f)
5:     Fin de Tant que
6:     Si f = −1 alors                                      /* Tous les calculs sont finis */
7:         Sortir
8:     Fin de Si
9:     f' = f
10:    Tant que F_f' est vide et f' ≠ −1 Faire             /* Premier parcours de vol de travail */
11:        f' = Pere(f')
12:    Fin de Tant que
13:    /* Test si la file choisie par le premier vol contient des travaux */
14:    Si F_f' est non vide alors
15:        T_calc(f') + +
16:        task = Premier(F_f')
17:        Calculer(task)
18:        Continuer
19:    Sinon
20:        f' = Pere(f)                                     /* Second parcours de vol de travail */
21:        Tant que f' ≠ −1 Faire
22:            Pour chaque i fils de f' Faire
23:                Si F_i est non vide alors
24:                    T_calc(i) + +
25:                    task = Premier(F_i)
26:                    Calculer(task)
27:                    Continuer
28:                Fin de Si
29:            Fin de Pour
30:            f' = Pere(f')
31:        Fin de Tant que
32:    Fin de Si
33:    /* Aucune tâche proche disponible */
34:    Attendre signal d'ajout dans F_f ou recommencer le vol n ms plus tard si pas de signal
35: Fin de Tant que
```

Il est également possible d'inverser les priorités : voler du travail à ses *frères* puis ensuite remonter dans l'arbre. Lorsque qu'on privilégie le père en premier, on ne sait pas où se trouve la tâche qui va être ordonnancée, mais il y a une probabilité qu'elle soit locale. Au contraire, en allant chercher chez les frères en premier, on sait qu'on ordonnancera systématiquement une tâche dont les données sont distantes. Nous avons observé que la solution retenue était plus performante sur le temps de factorisation. Nous verrons dans la section suivante les résultats de l'évaluation de ces deux parcours. On peut expliquer le phénomène par le fait que les vols qui se font sur le bas de l'arbre concernent des tâches avec des données de petites tailles. Le coût du transfert mémoire est alors plus coûteux et le surcoût est prépondérant sur le temps de calculs.

Nous allons maintenant étudier l'apport de cet ordonnancement dynamique sur le solveur PaStiX dans sa version en mémoire partagée, puis dans sa version distribuée.

4.3 Validation

Nous allons voir dans cette section l'apport de l'ordonnancement dynamique au sein du solveur PaStiX. Dans un premier temps nous présenterons les résultats obtenus sur nos trois architectures en mémoire partagée. Puis, nous présenterons les résultats obtenus sur les clusters Titane et Decrypthon en mémoire distribuée pour vérifier que les modifications apportées au schéma de communication sont bénéfiques à l'ordonnancement dynamique. Dans les deux parties, nous présentons des résultats sur les architectures NUMA qui sont les architectures cibles de ce travail, mais également les résultats sur architectures SMP pour montrer que ces apports sont génériques et ne perturbent pas l'utilisation du solveur sur d'autres architectures. Tous les temps présentés dans cette section intègrent la nouvelle méthode d'allocation.

4.3.1 Version en mémoire partagée

Nous présentons dans cette section les résultats obtenus sur le temps de factorisation en secondes du solveur PaStiX avec le nouvel ordonnancement dynamique présenté dans les sections précédentes en mémoire partagée. De même que dans les chapitres précédents nous ne montrons pas les résultats sur la phase de résolution du système. Les gains sont cependant du même ordre de grandeur que sur la factorisation mais avec des temps de calculs moins significatifs. Le premier tableau 4.1 montre l'évolution des temps de calcul sur l'ensemble de nos cas tests sur les trois architectures cibles : Borderline, Hagrid et Decrypthon. Chaque exécution utilise un thread de calcul par nœud et fixe les threads de calcul linéairement sur l'architecture (cf section 4.2.2 page 64). Ainsi, les threads de numéros successifs partagent des zones mémoire proches.

Le tableau 4.1 récapitule les résultats présentés dans le chapitre 2 page 42 de l'ordonnancement statique et les compare avec ceux de l'ordonnancement dynamique. Les gains affichés montrent que la version avec ordonnancement dynamique est meilleure que la version avec ordonnancement statique de 10% en moyenne et de 15% dans les meilleurs cas. On remarque que les gains varient d'une matrice à une autre mais également en fonction de l'architecture utilisée. En effet, le modèle de coût utilisé pour l'ordonnancement statique est mieux adapté à certaines architectures et notamment à celle de Decrypthon qu'aux architectures des deux autres clusters. Les réordonnancements de tâches sont plus nombreux sur les architectures NUMA en raison des erreurs des modèles de coût mais on constate que l'heuristique utilisée dans l'ordonnancement statique n'est pas non plus toujours optimale sur l'architecture d'origine des modèles de coût.

Les résultats sur les deux clusters NUMA : Borderline et Hagrid, nous montrent qu'on réussit à corriger correctement les imperfections des modèles de coûts sur ces architectures où les accès

TAB. 4.1 – Influence de l'ordonnancement dynamique sur le temps de factorisation numérique en secondes sur le solveur PaStiX. *Stat* et *Dyn* sont respectivement les versions avec ordonnancement statique et dynamique.

Matrice	Borderline			Hagrid			Decrypthon		
	Stat.	Dyn.	Gain	Stat.	Dyn.	Gain	Stat.	Dyn.	Gain
Matr5	422	**390**	7,58 %	386	**352**	8,81 %	161	**151**	6,21 %
Matr6	405	**363**	10,37%	361	**315**	12,74%	178	**150**	15,73%
Audi	244	**208**	14,75%	230	**196**	14,78%	99,9	**99,9**	0,00 %
Nice20	230	**211**	8,26 %	188	**173**	7,98 %	90,7	**89,3**	1,54 %
Inline	7,95	**7,41**	6,79 %	17,3	**14,7**	15,03%	**5,62**	6,12	-8,90%
Nice25	**2,62**	2,81	-7,25%	**5,08**	5,22	-2,76%	1,9	**1,9**	0,00 %
Mchlnf	2,43	**2,42**	0,41 %	**3,24**	3,47	-7,10%	1,89	**1,72**	8,99 %
Thread	2,17	**2,04**	5,99 %	2,26	**2,18**	3,54 %	1,12	**1,02**	8,93 %
3DSpectralWave	1650	**1590**	3,64 %	1040	**960**	7,69 %	603	**544**	9,78 %
3DSpectralWave2	287	**279**	2,79 %	174	**168**	3,45 %	107	**100**	6,54 %
Haltere	140	**131**	6,43 %	**92.1**	94	-2,06%	47,7	**47,3**	0,83 %
Fem_Hifreq_Circuit	106	**96,2**	9,25 %	**68**	73,1	-7,50%	32,3	**31,6**	2,17 %
Mono_500Hz	50,1	**49,1**	2,00 %	37,1	**33,4**	9,97 %	18,7	**17,5**	6,42 %

mémoire sont hiérarchiques. On constate également que l'ordonnancement dynamique conserve correctement l'affinité mémoire des données sinon les résultats seraient dégradés par le coût de la contention sur la machine comme on l'a vu dans le chapitre 2.

Dans la section 4.2.2, nous avons vu qu'il existe plusieurs solutions pour parcourir l'arbre de vol de travail. La première consiste à donner la priorité à la localité mémoire des données en parcourant les parents du nœud courant avant les frères. C'est la solution retenue dans notre ordonnancement dynamique. La seconde donne la priorité au chemin critique de l'arbre en parcourant les frères du nœud courant avant ses parents. Le tableau 4.2 montre l'impact de ses deux parcours sur l'ordonnancement dynamique avec l'architecture Borderline en utilisant huit threads de calcul, un par cœur.

Ce tableau montre que le parcours 2 (priorité du chemin critique) est moins efficace que le parcours choisi. En effet, cet ordre de parcours de l'arbre de vol de travail implique que lorsque un thread n'a plus de travail dans sa file, il va chercher en priorité dans les files voisines dont les données sont distantes, voire très éloignées si l'arbre de vol est large. Le coût d'accès à ces données est plus important que le coût d'attente du calcul de la tâche par le thread initialement prédit. De plus, ce choix augmente la contention sur la machine avec les voisins. C'est pourquoi les performances de ce deuxième parcours sont inférieures à celles du parcours retenu. Ces résultats confirment également que l'ordonnancement dynamique implémenté dans le solveur PaStiX améliore les performances du solveur car il conserve correctement l'affinité mémoire des données.

4.3.2 Version hybride MPI/Threads

Nous allons étudier dans cette section le comportement de notre algorithme d'ordonnancement dynamique lorsqu'on distribue les calculs sur plusieurs nœuds. De même que précédemment les deux versions du solveur comparées intègrent la nouvelle méthode d'allocation des données présentée dans le chapitre 2. Nous comparons ici les versions avec ordonnancement statique et dynamique avec le même schéma de communication, *ThComm*, présenté dans le chapitre 3.

Le tableau 4.3 présente les résultats obtenus sur le temps de factorisation pour les quatre cas

Tab. 4.2 – Influence de l'ordre de parcours dans le vol de travail sur le temps de factorisation numérique en secondes sur le solveur PaStiX sur la machine Borderline. Le parcours 1 privilégie en priorité les parents, c'est-à-dire la localité mémoire et le parcours 2 privilégie les frères, c'est-à-dire le chemin critique.

Matrice	Parcours 1	Parcours 2
Matr5	**390**	414
Matr6	**363**	400
Audi	**208**	284
Nice20	**211**	251
Inline	**7,41**	8,6
Nice25	**2,81**	3,19
Mchlnf	**2,42**	2,88
Thread	**2,04**	3,74
3DSpectralWave	**1590**	2090
3DSpectralWave2	**279**	320
Haltere	**284**	316
Fem_Hifreq_Circuit	**96,2**	104
Mono_500Hz	**49,1**	53,8

Tab. 4.3 – Comparaison du temps de factorisation numérique en secondes entre l'ordonnancement statique et l'ordonnancement dynamique. Les résultats présentés sont ceux des machines Titane et Decrypthon avec respectivement huit et seize threads par nœud.

Matrices	Nb.	Titane			Decrypthon		
	Nœuds	Stat.	Dyn.	Gain	Stat.	Dyn.	Gain
Matr5	1	159	**151**	5,03%	161	**151**	6,21%
	2	80,1	**78,7**	1,75%	99,6	**90,9**	8,73%
	4	56,7	**45,3**	20,11%	65	**55,7**	14,31%
	8	**27,8**	**27,8**	0%	45,9	**40,9**	10,89%
Matr6	1	144	**141**	2,08%	178	**150**	15,73%
	2	**75,3**	77,3	-2,66%	101	**92,3**	8,61%
	4	43	**42,1**	2,09%	59,8	**58,2**	2,67%
	8	**25,3**	26,1	-3,16%	44,5	**43**	3,37%
Audi	1	86,9	**82,6**	4,95%	**100**	**100**	0%
	2	**43,8**	45,9	-4,79%	67,3	**66,8**	0,74%
	4	**24,3**	24,4	-0,41%	33,6	**32,9**	2,08%
	8	**14,6**	15,9	-8,90%	23,2	**22,1**	4,74%
3DSpectralWave	1	563	**545**	3,20%	603	**544**	9,78%
	2	285	**279**	2,11%	315	**315**	0%
	4	**151**	154	-1,99%	189	**173**	8,47%
	8	**84,3**	85,7	-1,66%	144	**110**	23,61%

tests les plus importants en terme de mémoire consommée sur les clusters Titane et Decrypthon. Dans tous les tests, nous avons utilisé un thread de calcul par cœur fixé linéairement sur la machine, c'est-à-dire respectivement huit et seize threads de calcul sur les deux clusters. On constate que l'ordonnancement dynamique donne des résultats tout à fait différents sur les deux clusters.

L'évaluation sur le cluster SMP, Decrypthon, montre des gains très intéressants de 10% à 20% sur le temps de factorisation avec trois des cas tests utilisés et d'environ 5% en moyenne sur les autres cas. Au contraire, sur le cluster de nœuds NUMA, Titane, l'ordonnancement dynamique n'apporte pas de gains significatifs sur la version hybride MPI/Threads. Les résultats sont globalement équivalents entre les deux versions du solveur et la différence que l'on observe correspond plus exactement à la différence de coût du parcours de la boucle principale sur les tâches de calcul. Cependant, le cas de la matrice *Matr5* sur quatre processus montre un gain important sur le temps de factorisation comme pour les résultats homologues sur la machine Decrypthon. On remarque, en observant les diagrammes de Gantt des exécutions de l'ensemble des cas tests sur cette machine, qu'il existe très peu de temps d'inactivité sur la version avec ordonnancement statique. Il est donc difficile de gagner du temps à l'aide de l'ordonnancement dynamique sur ces cas tests avec cette architecture cible. En revanche, nous avions observé sur des résultats plus anciens des gains plus réguliers de 4% en moyenne sur le cluster Borderline qui est également une architecture NUMA à huit cœurs, avec les matrices Audi et Matr5. Ces résultats sont exposés dans le tableau 4.4. La différence de gains entre ces deux machines s'explique par la différence de puissance de calcul des processeurs utilisés sur les deux clusters. Les processeurs OPTERON dual-cœur de la machine Borderline ont surtout une puissance de calcul moins importante que celle de ceux utilisés pour générer nos modèles de coût. On crée ainsi plus de temps d'inactivité dans l'ordonnancement statique des calculs que sur la machine Titane.

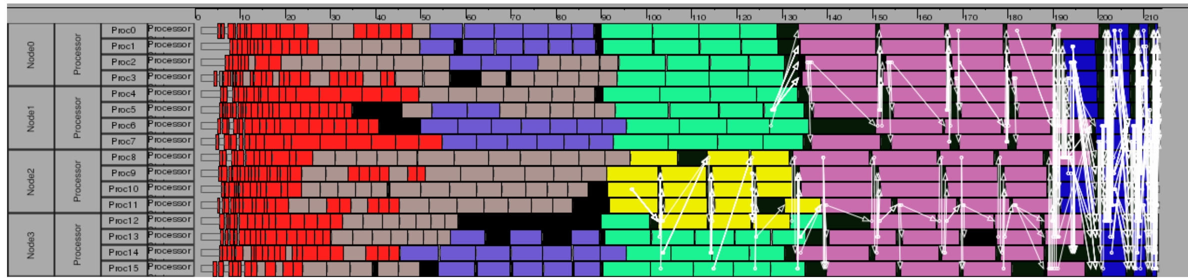
TAB. 4.4 – Comparaison du temps de factorisation numérique en secondes entre l'ordonnancement statique et l'ordonnancement dynamique sur le cluster Borderline.

Nb.	Audi			Matr5		
Nœud	Stat.	Dyn.	Gain	Stat.	Dyn.	Gain
1	217	**210**	3,2%	410	**389**	5,1%
2	111	**111**	0%	208	**200**	3,8%
4	60.5	**57.7**	4,6%	121	**114**	5,7%
8	37.2	**35.6**	4,3%	82.7	**78.8**	4,7%

Globalement les résultats obtenus sont satisfaisants et correspondent aux objectifs que nous nous étions fixés : obtenir des performances au moins équivalentes à celles de l'ordonnancement statique quand celui-ci est optimal et récupérer les défauts de la modélisation sur les cas où des temps d'inactivité sont introduits dans l'ordonnancement des calculs.

Les figures 4.7(a) et 4.7(b) sont les diagrammes de Gantt de la factorisation numérique de la matrice *Matr5* sur quatre processus de quatre threads. Les blocs noirs représentent les temps d'inactivité des threads de calcul et les flèches blanches, les communications. On constate que les gains obtenus avec l'ordonnancement dynamique sont de deux natures. Premièrement, le thread de communication permet de réceptionner les données et de débloquer les calculs plus rapidement. Deuxièmement, l'ordonnancement dynamique permet de compacter les calculs des tâches du bas de l'arbre d'élimination au début de l'exécution. Cependant, il reste encore quelques moments d'inactivité sur la fin de la factorisation numérique dûs au schéma de distribution des calculs 1D. Nous allons voir dans le chapitre suivant comment nous souhaitons combler ces temps d'inactivité, principalement sur les matrices de taille importante dont les derniers super-nœuds sont des matrices denses de grande taille.

L'ordonnancement dynamique développé dans le solveur PASTIX donne au final de bons résultats. Il améliore les performances du solveur en mémoire partagée sur toutes les architectures testées et on constate même une augmentation du gain obtenu avec l'augmentation du nombre



(a) Ordonnancement statique.

(b) Ordonnancement dynamique.

FIG. 4.7 – Diagrammes de Gantt de la factorisation de la matrice *Matr5* à l'aide de quatre processus de quatre threads.

de cœurs présents sur l'architecture. Les résultats sur la version hybride MPI/Thread sont également satisfaisants puisqu'on obtient des performances similaires à l'ordonnancement statique sur une architecture NUMA avec un faible nombre de cœurs de calcul. De même les résultats sur cette version sur l'architecture SMP montrent des gains similaires à ceux obtenus en mémoire partagée seule. Nous souhaitons au cours d'une prochaine collaboration avec le CEA/Cesta évaluer cet ordonnancement dynamique sur le cluster TERA 10 du CEA comportant 16 cœurs par nœud NUMA.

L'implémentation de ces travaux s'est faite de manière intrusive dans le code du solveur PaStiX en raison de la programmation hybride MPI/Threads utilisée et de la volonté d'avoir de meilleures performances. Il est toutefois possible d'adapter l'algorithmique utilisée à d'autres logiciels qui reposent sur une structure de dépendance entre les tâches de calcul proche d'une structure d'arbre. Cet algorithme d'ordonnancement dynamique des calculs est disponible dans la dernière version diffusée du solveur PaStiX. Dans l'annexe B, nous présentons comment activer cette option dans le solveur.

Chapitre 5

Algorithme d'ordonnancement dynamique à grains plus fins pour le solveur PaStiX

Sommaire

5.1 Description des méthodes de distributions 1D et 2D 76
5.1.1 Distribution 1D 76
5.1.2 Distribution 2D 78
5.1.3 Distibution 2D dynamique 80
5.2 Validation 83
5.2.1 Version en mémoire partagée 83
5.2.2 Version hybride MPI/Threads 84
5.3 Validation sur un cas challenge 86

Nous avons vu au cours des chapitres précédents la mise en place des prérequis au développement de l'ordonnancement dynamique d'un arbre de tâches sur machine NUMA et les solutions choisies pour la distribution des données et l'ordonnancement dynamique des calculs. Ces travaux nous ont permis d'améliorer les performances du solveur PaStiX sur les clusters de machines NUMA et SMP, mais nous ne nous sommes concentrés actuellement que sur une distribution 1D (par blocs-colonnes) des données. R. Schreiber a montré dans [92] qu'il est plus intéressant de basculer sur un schéma de distribution des données en 2D (par blocs) lorsqu'on utilise des matrices denses et un grand nombre de processeurs si l'on veut conserver une bonne scalabilité du solveur. Le solveur PaStiX propose une distribution 2D des données pour la factorisation [60] qui a confirmé la nécessité de ce schéma pour obtenir une meilleure scalabilité du code sur des problèmes de taille importante. Cependant le solveur ne dispose pas actuellement d'une étape de résolution adaptée à cette distribution. En effet, l'utilisation d'une distribution 2D des données implique des algorithmes de descentes/remontées plus complexes et plus coûteux en communications que les algorithmes utilisés pour les distributions 1D et ce problème n'a pas encore été étudié par les auteurs du solveur.

L'utilisation de machines multi-cœurs SMP ou NUMA avec un nombre de cœurs importants oblige également à revoir la distribution des données. En effet, grâce à la mémoire partagée sur ces architectures, il est plus intéressant d'avoir un grain de calcul plus fin au sein des nœuds

pour mieux répartir les coûts sur l'ensemble des ressources disponibles. De plus, sur certains problèmes qui ont un remplissage très important, le calcul des derniers super-nœuds de l'arbre d'élimination correspond à la factorisation d'une matrice dense de grande taille.

Pour améliorer la scalabilité du solveur, nous souhaitons permettre la création de tâches dynamiquement dans le solveur grâce à l'ordonnancement dynamique mis en place. De cette façon nous pourrons passer d'un schéma de distribution 1D des données à une distribution 2D des calculs au sein des nœuds.

Nous allons présenter dans la première section de ce chapitre les modèles de distribution 1D et 2D du solveur PaStiX. Puis, nous verrons comment nous créons dynamiquement des tâches de calcul 2D et comment elles s'intègrent dans l'ordonnancement dynamique mis en place dans le solveur. Ces modifications seront validées en mémoire partagée et dans la version hybride MPI/Threads sur l'ensemble de nos cas tests.

5.1 Description des méthodes de distributions 1D et 2D

Il existe deux méthodes de distribution des données dans les bibliothèques de résolution de systèmes linéaires. La première consiste à distribuer les données par blocs-colonnes. Elle est très utilisée dans les solveurs creux car elle permet de condenser les blocs extra-diagonaux en mémoire et de mieux bénéficier généralement des routines BLAS. La seconde consiste à distribuer les données par blocs. Elle est utilisée dans les solveurs denses car elle permet de débloquer plus rapidement les calculs et d'obtenir plus de parallélisme malgré l'augmentation des communications. Elle est également utilisée dans certains solveurs creux pour distribuer les données des super-nœuds du haut de l'arbre d'élimination par blocs qui peuvent être assimilés à des matrices denses de taille importante. Il est alors préférable de les distribuer par blocs pour obtenir un algorithme de factorisation qui passe correctement à l'échelle lorsque le nombre de processeurs augmente. On utilise dans la suite de cette section trois types de tâches pour $k \in [\![1, n]\!]$:

- $Diag(k)$: factorisation du bloc diagonal k : $A_{k,k} = L_{k,k}U_{k,k}$.
- $TRSM_k(i)$: résolution du système triangulaire sur le bloc extra-diagonal i du bloc-colonne k : $L_{(i),k}.U_{k,k} = A_{(i),k}$ (et $L_{k,k}.U_{k,(i)} = A_{k,(i)}$) pour $i \in [\![1, b_k]\!]$. b_k est le nombre de blocs extra-diagonaux du bloc-colonne k. On note $TRSM_k$, la résolution groupée du système sur l'ensemble des blocs extra-diagonaux du bloc-colonne k.
- $GEMM_k(i, j)$: calcule et ajoute la contribution du couple de blocs extra-diagonaux (i, j) du bloc-colonne k sur le bloc-colonne i : $A_{i,j} = A_{i,j} - L_{i,k}.U_{k,j}$ avec $(i, j) \in [\![1, b_k]\!]^2 \backslash \{j \leq i\}$ (et $A_{j,i} = A_{j,i} - L_{j,k}.U_{k,i}$ dans le cas non symétrique).

Nous présentons dans la suite de cette section comment les différents modèles de distribution du solveur PaStiX exploitent ces tâches. Puis, nous verrons comment nous exploitons la distribution par blocs-colonnes et l'ordonnancement dynamique pour modifier la distribution des calculs au sein des nœuds et créer dynamiquement les tâches.

5.1.1 Distribution 1D

La distribution utilisée par défaut dans le solveur PaStiX est la distribution 1D par blocs-colonnes. C'est sur cette distribution que nous nous sommes basés pour l'ensemble de nos travaux présentés dans les chapitres précédents, même si comme nous allons le voir les modifications apportées sont également disponibles avec une distribution par blocs. Cette distribution est la plus utilisée dans les solveurs de systèmes linéaires creux, principalement pour le bas de l'arbre d'élimination. Les blocs qui le constituent sont souvent de taille réduite et ne permettent pas

d'exploiter pleinement les avantages des bibliothèques BLAS comme la gestion du cache et des effets pipeline des processeurs. Pour améliorer ces résultats, on stocke les blocs extra-diagonaux de façon contiguë et on applique les calculs au bloc-colonne complet. Cela permet d'obtenir une plus grande surface de bloc, d'où plus de données et plus de calculs à exploiter dans les opérations matrices/matrices.

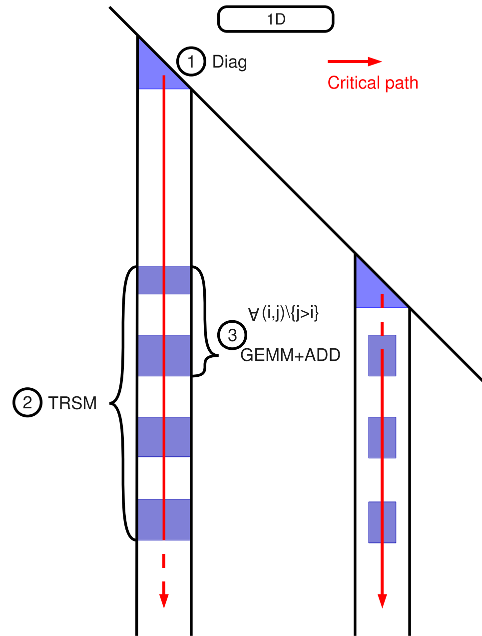


FIG. 5.1 – Étapes du calcul d'une tâche 1D et chemin critique du solveur.

Algorithme 6 Fonction Tâche1D(k)

1: $Diag(k)$
2: $TRSM_k$
3: **Pour** $j = 1$ à b_k **Faire**
4: **Pour** $i = j$ à b_k **Faire**
5: $GEMM_k(i, j)$
6: $NC_j - -$
7: **Fin de Pour**
8: **Si** $NC_j = 0$ **alors**
9: Débloquer *Tâche1D*(j)
10: **Fin de Si**
11: **Fin de Pour**

Le schéma de distribution des données est réalisé de telle sorte que l'on puisse regrouper un ensemble de calculs au sein d'une seule tâche. La figure 5.1 représente les différentes étapes des tâches associées aux distributions 1D. Ces étapes correspondent à celles décrites dans l'algorithme séquentiel 1 page 12 dans lequel les lignes 2 à 11 sont groupées en une seule tâche. Une tâche de la distribution 1D correspond au groupement pour tout $k \in [\![1, n]\!]$ des tâches : $Diag(k)$, $TRSM_k$ et de tous les $GEMM_k$. La seconde étape du calcul, $TRSM_k$, est celle qui bénéficie le plus de la distribution par blocs-colonnes et de l'allocation groupée des blocs extra-diagonaux car elle peut être réalisée avec un seul appel BLAS. La dernière étape du calcul des tâches 1D,

les $GEMM_k(i,j)$, ne bénéficie pas totalement de l'allocation par blocs-colonnes des données car les blocs cibles sont souvent plus grands que les contributions à ajouter et il est impossible de calculer les ajouts en une seule opération. On peut cependant bénéficier des routines BLAS matrices/matrices sur la multiplication du bloc extra-diagonal j avec le groupement des blocs extra-diagonaux pour $i \in [\![j, b_k]\!]$. Il faut alors faire un compromis entre le surcoût mémoire nécessaire au stockage du produit matriciel et le temps gagné à regrouper les appels BLAS. Le regroupement de toutes ces étapes en une seule tâche dans le solveur PaStiX donne le chemin critique en rouge sur la figure 5.1. Une tâche n'est prête que lorsque les tâches précédentes ont fini les étapes 1 et 2 et le calcul de toutes les contributions destinées au bloc-colonne qui lui est associé.

5.1.2 Distribution 2D

La deuxième solution pour distribuer les données est la distribution 2D par blocs. Elle fournit une meilleure scalabilité aux solveurs grâce à son grain de calcul plus fin qui permet d'alimenter plus rapidement tous les processeurs disponibles. Cependant, elle entraîne un surcoût de communication important que l'on peut recouvrir par la factorisation d'autres blocs lorsque la matrice est suffisamment dense. Pour obtenir ce recouvrement, on ne doit plus agréger les contributions comme dans la distribution 1D. Cependant le solveur PaStiX conserve une approche *fan-in* pour la gestion de ces contributions et se base sur la simulation pour savoir quand effectuer les échanges de données contrairement aux méthodes classiques de distribution 2D qui basculent sur approche *fan-out*. Plusieurs solveurs comme SuperLU_Dist, PaStiX ou WSMP proposent de mixer les deux distributions : les éléments du bas de l'arbre d'élimination sont distribués avec un schéma 1D et ceux du haut de l'arbre sont distribués en 2D car ils nécessitent plus de calculs et ils sont répartis sur plusieurs processus MPI. Le solveur MUMPS propose également de mixer les deux approches en appelant la bibliothèque de résolution de systèmes denses ScaLAPACK [18] sur le dernier séparateur.

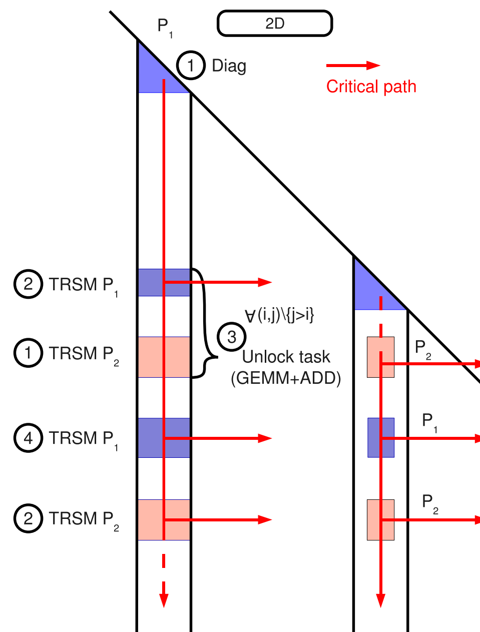


Fig. 5.2 – Étapes du calcul des tâches 2D et chemin critique du solveur.

Algorithme 7 Tâches 2D

1: **Fonction** DIAG(k)
2: $A_{k,k} = L_{k,k}U_{k,k}$
3: **Pour** $j = 1$ à b_k **Faire**
4: Débloquer $TRSM_k(j)$
5: **Fin de Pour**
6: **Fin de Fonction**
7:
8: **Fonction** $TRSM_k(j)$
9: Résoudre $L_{j,k}.U_{k,k} = A_{j,k}$
10: Résoudre $L_{k,k}.U_{k,j} = A_{k,j}$
11: **Pour** $i = 1$ à b_k **Faire**
12: **Si** $TRSM_k(i)$ calculé **alors**
13: Débloquer $GEMM_k(i, j)$
14: **Fin de Si**
15: **Fin de Pour**
16: **Fin de Fonction**
17:
18: **Fonction** $GEMM_k(i, j)$
19: $A_{i,j} = A_{i,j} - L_{i,k}.U_{k,j}$
20: $A_{j,i} = A_{j,i} - L_{j,k}.U_{k,i}$
21: $NC_j - -$
22: **Si** $NC_j = 0$ **alors**
23: Débloquer $Diag(j)$
24: **Fin de Si**
25: **Fin de Fonction**

La distribution 2D des données permet de découper les tâches de calcul plus finement. La figure 5.2 illustre le fonctionnement de la distribution 2D dans laquelle on utilise unitairement les tâches décrites précédemment : $Diag(k)$, $TRSM(j)$ et $GEMM(i, j)$. Dans cette version, le calcul de la tâche $Diag(k)$ débloque toutes les tâches $TRSM_k(j)$ pour $j \in [\![1, b_k]\!]$ (ligne 4 de l'algorithme 7). Elles sont ensuite calculées en parallèle sur les différents threads et processus qui possèdent les données associées. Ces tâches débloquent à leur tour chacune un ensemble de tâches $GEMM_k(i, j)$ avec $(i, j) \in [\![1, b_k]\!]^2 \backslash \{j \leq i\}$ dès que les tâches $TRSM(i)$ et $TRSM(j)$ ont été calculées. Les numéros de la figure 5.2 indiquent l'ordre d'exécution des tâches sur les processus P_1 et P_2. Ces tâches peuvent en plus être exécutées en parallèle sur plusieurs threads. On constate sur la figure que cette distribution permet de dégager le chemin critique plus rapidement en offrant de multiples possibilités de calcul après chaque tâche, c'est pourquoi cette distribution est préférée dans le calcul de matrices denses. De plus, elle offre une meilleure scalabilité.

Le solveur PASTIX gère la distribution 2D de la même façon que la distribution 1D. La distribution des blocs de la matrice est faite lors de la phase de simulation temporelle de la factorisation. Les tâches *Diag* et *TRSM* sont ajoutées dans les files de tâches prêtes comme les tâches 1D et le bloc qui leur est associé est placé sur le processeur qui a été choisi pour exécuter la tâche. Leur distribution dépend du processeur qui sera disponible au plus tôt pour les calculer.

Par contre, les tâches *GEMM* sont plus difficiles à gérer car les données qui leur sont associées sont déjà distribuées sur un ou plusieurs processeurs (par distribution des autres tâches). Pour simplifier la complexité de l'algorithme de simulation temporelle, le solveur PaStiX choisit de distribuer les tâches $GEMM(i, j)$ systématiquement sur le processeur qui est en charge de $TRSM(i)$. Le passage à une distribution 2D accroît fortement la complexité de l'algorithme dans le choix des tâches à exécuter. En effet, le nombre de tâches passe de $O(n)$ à $O(n^2)$ sur la partie de la matrice distribuée en 2D. Le temps de calcul du prétraitement peut alors ne plus être négligeable devant le temps de factorisation lorsque l'on distribue un nombre trop important de niveaux en 2D.

Les modifications apportées au solveur dans les chapitres précédents ont été généralisées pour la distribution 2D statique des données. Pour conserver la continuité des données en mémoire qui permet d'utiliser efficacement les routines BLAS sur des blocs de données plus importants, nous avons choisi dans le cas d'une distribution 2D de conserver l'allocation des données par blocs-colonnes. Cependant, dans cette distribution, les blocs-colonnes ne contiennent plus que les blocs de données locaux. Ainsi, un bloc-colonne peut-être alloué sur plusieurs processus MPI. Nous avons décidé d'attribuer l'allocation et l'initialisation des données au thread qui possède le bloc diagonal ou le premier bloc extra-diagonal. Dans le cas de l'ordonnancement dynamique, ces calculs se retrouvent dans le haut de l'arbre d'élimination locale et ont plusieurs threads candidats à leur factorisation. L'allocation des données est alors toujours distribuée par round-robin sur les threads. L'ordre de criticité des tâches fait qu'on obtient le "même" résultat que dans la version statique. Le thread qui sera en charge d'allouer les données d'un bloc-colonne sera le premier à recevoir une tâche du bloc-colonne. Il s'agit du thread qui est en charge du bloc diagonal ou du premier bloc extra-diagonal. L'ordonnancement dynamique, tel qu'il a été conçu, permet également de gérer cette distribution en ajoutant simplement les tâches de calcul 2D dans les files d'exécution. L'avantage de la version avec ordonnancement dynamique est que les tâches $GEMM(i, j)$ ne sont plus seulement dédiées à un seul thread, mais à tous ceux qui sont candidats à l'exécution de la tâche $TRSM(i)$.

Nous souhaitons proposer une solution à l'aide de l'ordonnancement dynamique qui puisse exploiter l'avantage que représente l'implémentation hybride MPI/Thread du solveur PaStiX sur les architectures multi-cœurs. En effet, l'exploitation de machines à mémoire partagée incite à créer des tâches de calcul à grain plus fin que celui imposé par les modèles de distribution en mémoire distribuée pour alimenter toutes les ressources. L'ordonnancement dynamique peut nous permettre de créer ces tâches de calcul dynamiquement tout en conservant l'efficacité de l'algorithme de distribution par bloc-colonnes.

5.1.3 Distibution 2D dynamique

Nous venons d'étudier les deux schémas de distribution des données disponibles dans les solveurs linéaires creux et plus particulièrement leur fonctionnement dans le solveur PaStiX. La distribution des données par blocs permet d'avoir un grain de calcul plus fin qui s'adapte mieux à l'utilisation d'un nombre de processeurs très important. L'ordonnancement dynamique au sein du solveur PaStiX nous a permis de développer une nouvelle méthode, appelée *ESP (Enhanced Sparse Parallelism)*, pour obtenir l'avantage d'une distribution par blocs (grain de calcul plus fin) et conserver l'algorithme de résolution 1D présent dans le solveur PaStiX. Les distributions des données par blocs-colonnes sont plus adaptées à des clusters de machines à mémoire distribuée, tandis que les schémas de distribution des données par blocs s'adaptent mieux aux machines à mémoire partagée.

En effet, le coût des échanges engendrés par une distribution 2D est faible en mémoire partagée. La distribution dynamique des calculs que nous avons développée dans le solveur PaStiX se base sur cette propriété. Nous préservons la distribution des données par blocs-colonnes pour répartir équitablement les données entre les nœuds et conserver un nombre plus faible de communications de taille plus importante pour exploiter plus efficacement les réseaux rapides. Cette solution conserve par conséquent le même nombre de tâches que pour une distribution 1D et le prétraitement reste peu coûteux. Puis nous créons dynamiquement lors de la factorisation numérique les tâches correspondantes à une distribution par blocs vue dans la section précédente.

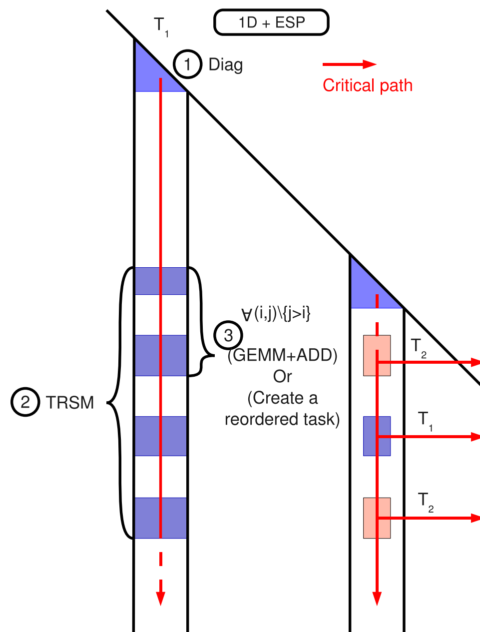


Fig. 5.3 – Étapes du calcul des tâches 1D avec l'option ESP qui permet de créer dynamiquement des tâches 2D.

La figure 5.3 illustre la nouvelle répartition des tâches de calcul dans cette version *ESP* du solveur. L'ordonnancement dynamique contient initialement les tâches de calcul 1D comme pour la distribution par blocs-colonnes qui regroupent les trois types de tâches de base. Dans cette version, les deux premières étapes restent identiques. De cette façon, on conserve la résolution des systèmes triangulaires sur le bloc-colonne complet pour accroître l'efficacité des routines BLAS utilisées. Lorsque les blocs-colonnes deviennent importants, les tâches de calcul 1D utilisées dans le solveur ne permettent pas de débloquer le calcul des autres blocs-colonnes suffisamment rapidement. En effet, toutes les sous-tâches $GEMM(i,j)$ avec $i \in [\![j, b_k]\!]$ sont calculées séquentiellement, alors qu'elles pourraient être calculées en parallèle pour débloquer plus rapidement la factorisation du bloc-colonne j comme ce serait le cas avec une distribution 2D. La méthode *ESP* permet de retrouver en partie le chemin critique de la version 2D.

Au cours de la double boucle de calcul des contributions, à la ligne 5 de l'algorithme 8, on choisit de faire le calcul immédiatement ou de créer une tâche responsable de ce calcul. Elle pourra ainsi être exécutée en parallèle par un autre thread. Ce dernier point est illustré par l'étape 3 de la figure 5.3. La création de ces tâches de calcul est dépendante de la surface du bloc résultat de la multiplication des blocs extra-diagonaux i et j. Celui-ci doit être suffisamment

Algorithme 8 Fonction Tâche1D_ESP(k)

1: $Diag(k)$
2: $TRSM_k$
3: **Pour** $j = 1$ à b_k **Faire**
4: **Pour** $i = j$ à b_k **Faire**
5: **Si** $Dimension(A_{k,i}) > seuil$ **alors**
6: Créer une tâche $GEMM_k(i, j)$
7: **Sinon**
8: $GEMM_k(i, j)$
9: $NC_j - -$
10: **Fin de Si**
11: **Fin de Pour**
12: **Si** $NC_j = 0$ **alors**
13: Débloquer *Tâche1D_ESP*(j)
14: **Fin de Si**
15: **Fin de Pour**

grand pour que le coût de calcul des contributions qui lui sont liées soit supérieur au coût de création de la tâche et à la perte d'efficacité sur les routines BLAS. Ceci permet de limiter le nombre de tâches créées. De plus, on ne crée pas de tâches 2D sur le bas de l'arbre d'élimination. On limite également la création de ces tâches aux blocs-colonnes dont plusieurs threads sont candidats pour les factoriser. En effet, il est inutile de les créer dans les files de tâches réservées à un seul thread car c'est ce même thread qui exécutera ces tâches.

Les tâches $GEMM(i, j)$ créées dynamiquement sont insérées dans la même file de tâches que celle dont est originaire la tâche initiale. Le problème lors de cet ajout est de déterminer la priorité de ces nouvelles tâches dans la file. Dans un premier temps, nous avons choisi de les insérer avec une priorité juste inférieure à celle de la tâche chargée de la factorisation du bloc-colonne destinataire de ces contributions. Cette solution n'était pas performante car toutes les contributions de tous les nœuds appartenant aux fils du nœud destinataire dans l'arbre d'élimination sont calculées simultanément par tous les threads. Il y avait alors un goulot d'étranglement sur l'accès au bloc-colonne cible pour ajouter les contributions et toutes ces tâches se retrouvaient sérialisées. Pour remédier à ce problème nous avons choisi de leur attribuer comme priorité la moyenne des priorités des tâches source et destination. Ainsi, dans l'agorithme 8 la tâche créée à la ligne 13 est insérée dans une file de tâches avec comme priorité : $(\textit{Priorité}(k) + \textit{Priorité}(j))/2$. Les contributions sont ainsi mieux réparties dans la file des tâches prêtes et on observe moins de contention sur l'accès au bloc-colonne cible.

Cette solution de création dynamique de tâches de calcul 2D permet de conserver un temps d'analyse réduit puisqu'on conserve un schéma de distribution 1D et la même complexité qu'avec une distribution 1D. Seules les factorisations LL^T et LU du solveur PaStiX profitent de ce nouveau schéma de distribution des calculs. En effet, la factorisation LDL^T fait apparaître une séquentialité dans l'ordre de traitement des tâches 2D qui n'est pas adaptée à ce schéma dynamique. C'est pourquoi cette solution n'est pas implémentée dans la factorisation de Crout dans le solveur PaStiX. L'implémentation réalisée permet de mixer les approches de distributions 2D et ESP. Nous verrons dans la section 5.3 page 86 que cette solution peut s'avérer avantageuse sur certains problèmes car un compromis est possible entre le coût de l'analyse et la distribution obtenue. La section suivante présente les résultats sur le solveur PaStiX de cette

nouvelle option du solveur avec nos cas tests.

5.2 Validation

Cette section expose les résultats obtenus avec l'utilisation de la méthode ESP sur l'ensemble de nos cas tests. Nous présenterons dans un premier temps les résultats en mémoire partagée sur les trois architectures Borderline, Hagrid et Decrypthon afin d'examiner l'impact de cette solution sur les architectures NUMA comme sur les architectures SMP avec un nombre de cœurs différent. Puis nous verrons si cette solution permet de débloquer plus rapidement les communications en version distribuée sur les clusters Titane et Decrypthon. Tous les résultats présentés dans cette section prennent comme référence la version du solveur obtenue dans le chapitre précédent, c'est-à-dire avec la nouvelle méthode d'allocation, le thread de progression en réception et l'ordonnancement dynamique.

5.2.1 Version en mémoire partagée

Dans cette première section, nous étudions l'apport de la méthode ESP en mémoire partagée. Celle-ci permet, lorsque le nombre de cœurs de la machine est important, d'avoir un grain de calcul plus adapté au nombre de ressources disponibles. Cette section ne présente pas les résultats sur l'étape de résolution car la méthode ESP n'est disponible que pour la factorisation numérique et elle ne se justifie pas pour l'étape de résolution en raison du faible volume de calcul par rapport aux communications. On pourrait cependant étudier cette possibilité pour l'utilisation du solveur avec un nombre important de seconds membres. Le passage à une résolution multi seconds membres permet en effet d'utiliser des routines BLAS de niveau supérieur qu'il pourrait être intéressant de découper en tâches parallèles.

Tab. 5.1 – Comparaison des versions avec l'ordonnancement dynamique et avec ou sans la méthode ESP de distribution des calculs sur le temps de factorisation numérique en secondes. N_t est le nombre de tâches de calcul 2D ajoutées dynamiquement.

Matrice	Borderline				Hagrid				Decrypthon			
	Dyn.	ESP	N_t	Gain	Dyn.	ESP	N_t	Gain	Dyn.	ESP	N_t	Gain
Matr5	390	**384**	8091	1,54%	352	**328**	11755	6,82%	151	**149**	10305	1,32%
Matr6	363	**357**	7623	1,65%	315	**309**	10771	1,90%	150	**149**	11012	0,67%
Audi	208	**205**	1547	1,44%	196	**174**	5579	11,22%	99,9	**97,5**	6454	2,40%
Nice20	**211**	**211**	4201	0%	173	**165**	5414	4,62%	89,3	**88,4**	6981	1,01%
Inline	7,41	**7,09**	6	4,32%	14,7	**14**	33	4,76%	6,12	**5,57**	161	8,99%
Nice25	2,81	**2,74**	136	2,49%	5,22	**5,15**	331	1,34%	1,9	**1,89**	251	0,53%
Mchlnf	**2,42**	**2,42**	57	0%	3,47	**3,36**	173	3,17%	**1,72**	1,81	154	-5,23%
Thread	2,04	**1,92**	147	5,88%	2,18	**2,09**	201	4,13%	**1,02**	1,04	201	-1,96%
3DSpectralWave	**1590**	**1590**	8385	0%	960	**946**	15450	1,46%	554	**538**	16371	2,89%
3DSpectralWave2	279	**268**	2415	3,94%	**168**	**168**	191	0%	100	**99.4**	121	0,60%
Haltere	**131**	132	947	-0,76%	94	**91,6**	290	2,55%	47,3	**47**	136	0,63%
Fem_Hifreq_Circuit	**96,2**	96,8	0	-0,62%	73,1	**70,3**	21	3,83%	**31,6**	**31,6**	126	0%
Mono_500Hz	49,1	**49**	14	0,20%	33,4	**32**	233	4,19%	**17,5**	17,7	151	-1,14%

Le tableau 5.1 présente les résultats obtenus sur l'ensemble de nos cas tests avec la méthode ESP. Nous avons choisi pour ces cas tests un seuil de création des tâches de calcul 2D correspondant à la surface d'un bloc de taille 128×128. On constate sur la colonne N_t du tableau que ce seuil permet de créer un nombre important de tâches de calcul 2D sur nos plus gros cas tests : de 2000 à 8000 ou de 6000 à 16000 suivant le nombre de cœurs présents sur l'architecture cible. La création de ce parallélisme permet d'apporter un gain sur le temps de factorisation

de ces cas tests. Au contraire, les cas tests plus petits génèrent très peu de nouvelles tâches et par conséquent le gain affiché n'est pas significatif de l'apport de la méthode ESP qui n'est pas nécessaire sur des matrices si petites. Le nombre de tâches créées dépend également du nombre de cœurs présents sur la machine puisque la création de tâches n'est autorisée que dans les files de l'arbre de vol de travail possédant au moins N threads candidats pour exécuter son contenu et nous avons choisi de fixer N à 4. La machine Borderline est ainsi moins préposée à la création dynamique de tâches de calcul. De même, le nombre de tâches créées dynamiquement varie entre les deux machines à seize cœurs car la renumérotation, et donc la factorisation symbolique, ne sont pas identiques. On remarque également la même tendance que sur les résultats obtenus pour la méthode d'allocation dans le chapitre 2. Les gains restent en moyenne plus grands sur les machines NUMA que sur le cluster SMP et ils augmentent avec l'augmentation du nombre de cœurs.

La méthode ESP implémentée dans le solveur PaStiX apporte donc un gain supplémentaire sur le temps de factorisation des matrices creuses qui s'ajoute à ceux déjà obtenus avec les développements présentés dans les chapitres précédents.

5.2.2 Version hybride MPI/Threads

Nous nous intéressons désormais aux résultats de cette méthode en utilisant la version hybride MPI/Thread du solveur PaStiX. Les résultats sont présentés comme dans le chapitre précédent sur les clusters Titane et Decrypthon avec la version du solveur utilisant la nouvelle méthode d'allocation et le schéma de communication *ThComm*. Les gains affichés dans le tableau 5.2 sont ceux de la version finale (ESP) intégrant l'ordonnancement dynamique et la méthode ESP par rapport à la version initiale avec ordonnancement statique. Nous avons utilisé le même seuil que dans la section précédente pour choisir de créer dynamiquement les tâches de calcul, c'est-à-dire une surface de bloc supérieure à 128^2.

Tab. 5.2 – Comparaison des versions de l'ordonnancement dynamique avec ou sans la méthode ESP par rapport à l'ordonnancement statique initial sur le temps de factorisation en secondes.

Matrices	Nb.	Titane				Decrypthon			
	Nœuds	Stat.	Dyn.	ESP	Gain	Stat.	Dyn.	ESP	Gain
Matr5	1	159	151	**150**	5,66%	161	151	**149**	7,45%
	2	80,1	78,7	**78,4**	2,12%	99,6	90,9	**90,1**	9,54%
	4	56,7	45,3	**45,2**	20,28%	65	**55,7**	**55,7**	14,31%
	8	**27,8**	**27,8**	29,6	-6,47%	45,9	**40,9**	43,4	5,45%
Matr6	1	144	141	**140**	2,78%	178	150	**149**	16,29%
	2	75,3	77,3	**74,7**	0,80%	101	92,3	**92,1**	8,81%
	4	43	42,1	**41,6**	3,26%	59,8	58,2	**55,8**	6,69%
	8	**25,3**	26,1	27,7	-9,49%	44,5	**43**	44	1,14%
Audi	1	86,9	82,6	**81,7**	5,98%	100	100	**97,5**	2,50%
	2	**43,8**	45,9	45,8	-4,57%	67,3	66,8	**62,4**	7,28%
	4	**24,3**	24,4	**24,3**	0%	33,6	**32,9**	33,2	1,20%
	8	**14,6**	15,9	14,8	-1,37%	23,2	**22,1**	23,5	-1,13%
3DSpectralWave	1	563	545	**543**	3,55%	603	544	**538**	10,78%
	2	285	**279**	286	-0,35%	315	315	**306**	2,86%
	4	**151**	154	152	-2,65%	189	173	**171**	9,52%
	8	**84,3**	85,7	85	-0,83%	144	**110**	112	22,22%

De même que sur les résultats en mémoire partagée, on constate que la méthode ESP n'apporte pas un gain très important sur cet ensemble de cas tests lorsqu'on utilise plusieurs processus. Ces résultats ont plusieurs explications. La première est le seuil utilisé pour la création dynamique de tâches. Nous avons choisi d'utiliser le même que pour la version en mémoire partagée, or la phase de répartition proportionnelle décrite dans la section 4.1.1 page 56 découpe plus finement les blocs-colonnes du haut de l'arbre d'élimination lorsqu'il y a plusieurs processus candidats à leur factorisation. Il y a donc moins de calculs qui dépassent le seuil utilisé dans ces tests. Les benchmarks effectués avec un seuil plus faible permettent d'améliorer les temps de factorisation obtenus mais pas de manière significative. En effet, en étudiant les diagrammes de Gantt sur le cas présenté dans le chapitre précédent, on s'aperçoit qu'il n'y a que très peu de gain possible sur l'exécution de ces cas tests. En revanche, sur la machine SMP, la méthode ESP apporte un gain de façon plus régulière et sur un nombre plus important de processus. Elle n'est en retrait sur la version avec l'ordonnancement dynamique sans la méthode ESP que sur les cas tests à huit processus.

La méthode ESP permet d'améliorer l'efficacité du solveur en mémoire partagée mais, comme pour une distribution 2D statique des données, les cas tests utilisés ici ne mettent pas suffisamment en avant la nécessité d'avoir un découpage plus fin des tâches de calcul. Nous verrons dans la section suivante le comportement sur un cas test plus important où le découpage par blocs des données devient nécessaire pour obtenir une bonne scalabilité du solveur. Les développements exposés dans les deux derniers chapitres, l'ordonnancement dynamique et la méthode ESP, donnent globalement de bons résultats particulièrement en mémoire partagée où les deux versions cumulées apportent des gains moyens de 10% sur l'ensemble des architectures testées. Sur la version hybride MPI/Threads, nous obtenons les mêmes gains sur le cluster SMP. Les résultats sur le cluster NUMA, Borderline, montrent également des gains, mais moins importants, car les possibilités de réordonnancement sont limitées par un nombre cœurs plut petit. En revanche, les résultats du cluster Titane montrent l'efficacité de l'ordonnancement statique calculé dans le solveur PaStiX car avec le nouveau schéma de communication qui permet un meilleur recouvrement des communications par les calculs, il ne reste plus de gains possibles sur ces cas tests. On constate donc que notre ordonnancement dynamique ne dégrade pas les performances sur le cluster Titane lorsqu'il y a peu de gains possibles et rattrape les quelques erreurs possibles comme sur la matrice *Matr5* avec quatre processus. Au contraire, sur la machine Decrypthon, les résultats sont globalement améliorés par l'ordonnancement dynamique et la méthode ESP.

5.3 Validation sur un cas challenge

L'augmentation de la puissance des ressources de calcul et les évolutions dans les méthodes de résolution, notamment avec l'introduction du parallélisme, ont permis aux scientifiques de raffiner les calculs en introduisant de nouvelles inconnues. Nous nous sommes intéressés dans cette thèse à des problèmes de taille relativement conséquente allant jusqu'à un million d'inconnues mais il existe déjà des problèmes de taille plus importante. La version hybride MPI/Threads du solveur PaStiX a déjà permis aux scientifiques du CEA/Cesta de résoudre un problème à 83 millions d'inconnues. Ce problème a été résolu en 5 heures sur 768 processeurs du cluster TERA10, 48 nœuds de 16 cœurs. Cependant, il y avait de grandes disparités entre les temps de calcul de chaque processus. Pour étudier la scalabilité du solveur sur de grands cas challenges, le CEA/Cesta nous a soumis une version plus petite de son problème comportant 10 millions d'inconnues et qui constitue déjà un challenge difficile pour les solveurs basés sur une méthode directe. Le problème étudié est issu d'un code d'électromagnétisme du CEA/Cesta qui travaille sur des maillages de cônes 3D. Les caractéristiques de la matrice sont décrites ci-dessous :

Nombre d'inconnues	10 423 737
Nombre de termes non nuls de A	89 072 871
Nombre de termes non nuls de L	6 724 303 039
Nombre d'OPC pour la factorisation de Cholesky	4,42e+13
Espace mémoire pour le stockage des coefficients	104 Go
Complexes double précision	
Problème symétrique	

L'étude se fait sur la machine Vargas de l'IDRIS. Il s'agit d'un cluster de nœuds IBM à base de processeurs Power6 qui sont interconnectés par un réseau Infiniband X4 DDR. Chaque nœud du cluster comporte 16 Power6 bi-cœurs, soit 32 cœurs par nœud associés à 128 Go de mémoire. Cette machine est SMP et donc sans facteur NUMA. Dans cette section, nous étudions uniquement l'apport de l'ordonnancement dynamique sur ce cas, ainsi que celui de la méthode ESP de distribution des calculs. Pour cette étude, nous nous basons sur les traces d'exécution sur huit nœuds obtenues avec le logiciel ViTE [25]. ViTE est un logiciel de visualisation de traces d'exécution au format Pajé [29] réalisé par un groupe de sept étudiants de l'ENSEIRB que j'ai encadré au cours de ma dernière année de thèse. Ce logiciel sera présenté dans l'annexe A.

Ordonnancement statique et distribution 1D La première exécution de ce cas test que nous avons est celle qui va nous servir de base pour étudier l'apport des développements de cette thèse. Il s'agit de l'utilisation de l'ordonnancement statique du solveur avec une distribution 1D des données sur les processus. Le diagramme de Gantt de cette exécution est illustré par la figure 5.4.

Sur ce diagramme, chaque ligne représente un thread de calcul. Ces threads sont regroupés par processus. On représente par des blocs de couleur les temps de calcul des tâches 1D et par des blocs sombres les temps d'inactivité. Les différentes couleurs symbolisent le nombre de threads candidats à l'exécution de la tâche lors du parcours décrit dans la section 4.1.1 page 56. Elles représentent également le niveau de profondeur de la tâche dans l'arbre d'élimination. Les temps d'inactivité sont de trois sortes : une attente de contribution locale, une attente de contribution distante ou une attente de tâches disponibles. Cette dernière remplace les deux précédentes dans le cas de la version avec ordonnancement dynamique que l'on étudiera ensuite. Enfin chaque trait blanc représente une communication point à point pour l'échange des contributions.

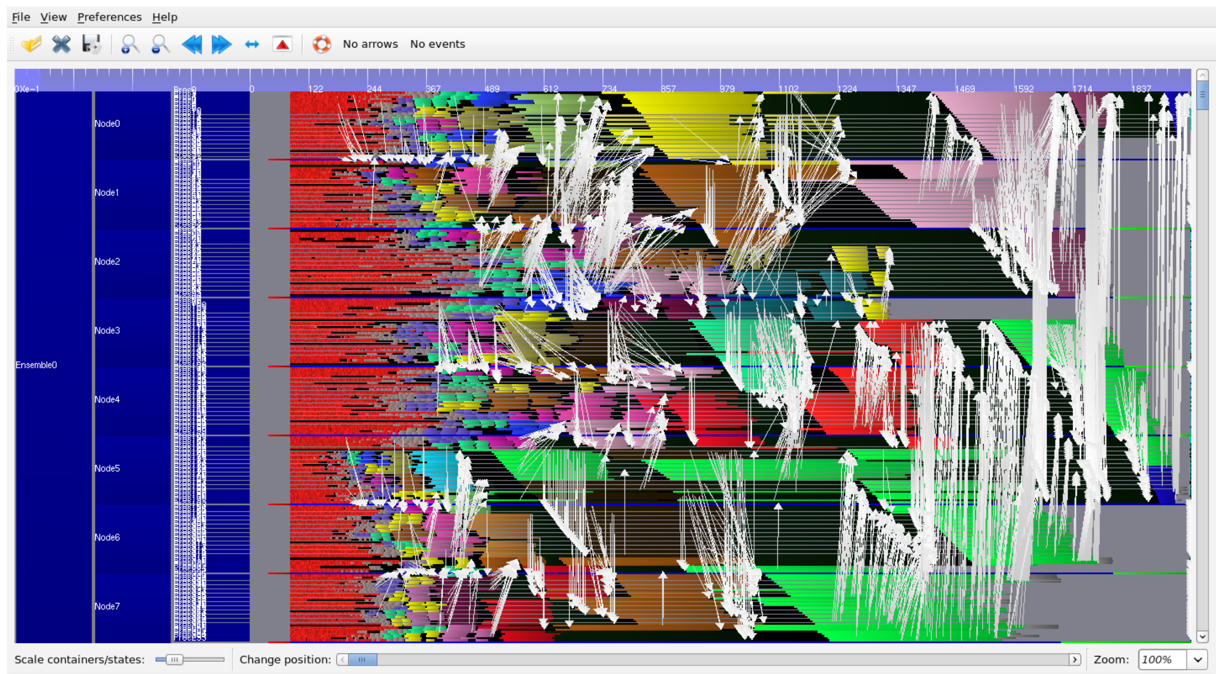


FIG. 5.4 – Diagramme de Gantt de la factorisation sur 8 nœuds de 32 cœurs de la matrice 10Millions avec l'ordonnancement statique et la distribution des données par blocs-colonnes.

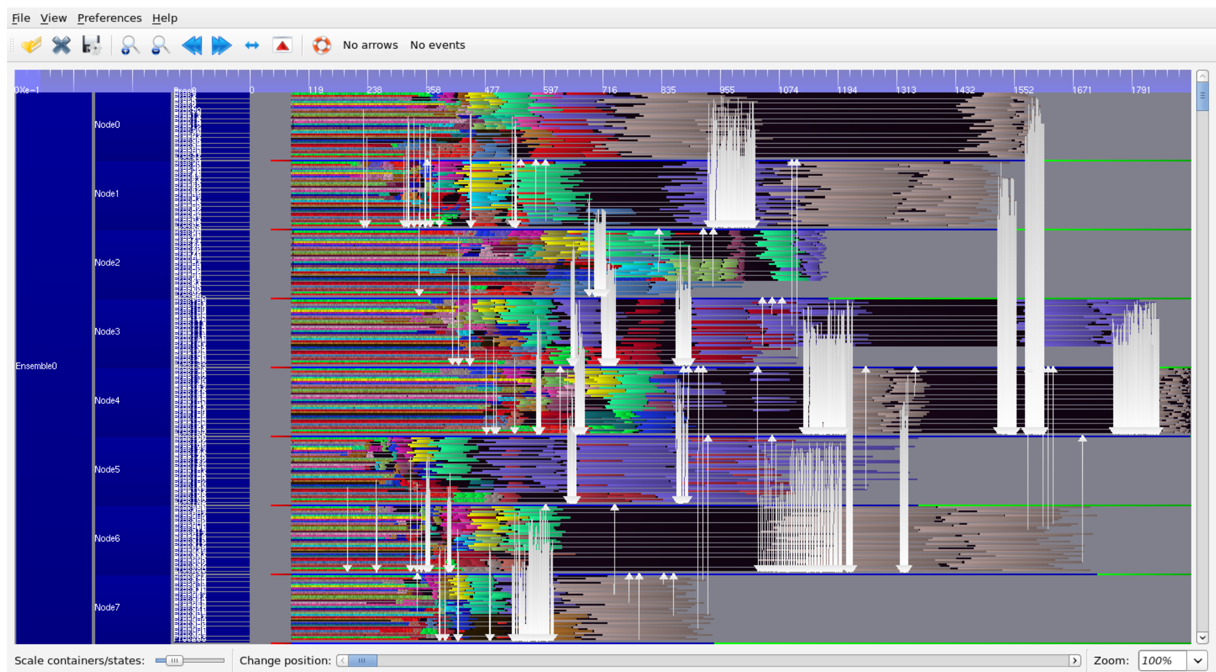


FIG. 5.5 – Diagramme de Gantt de la factorisation sur 8 nœuds de 32 cœurs de la matrice 10Millions avec l'ordonnancement dynamique et la distribution des données par blocs-colonnes.

On constate sur le diagramme de Gantt de la figure 5.4 qu'il existe encore de nombreux temps d'inactivité lors de la factorisation avec ordonnancement statique de cette matrice, particulièrement sur le haut de l'arbre d'élimination, où la taille de la matrice et celle de ses derniers super-nœuds montrent les limites de la distribution 1D des données. Le schéma classique de factorisation de matrices denses apparaît sur ces super-nœuds, comme on peut le voir sur les tâches vertes ou jaunes par exemple. Le calcul de chacune des tâches vertes du nœud 7 représente environ 10 à 15 secondes de calcul. Il serait donc intéressant de débloquer des calculs plus rapidement pour occuper les threads au niveau de ce "*triangle*" d'inactivité.

Ordonnancement dynamique et distribution 1D Le diagramme de la figure 5.5 montre les gains possibles avec l'ordonnancement dynamique par rapport au diagramme vu précédemment. La coloration des tâches est faite sur cette trace (et sur toutes celles avec ordonnancement dynamique) en fonction de chaque nœud de l'arbre de vol de travail. Celle-ci représente également le niveau des tâches dans l'arbre d'élimination mais sur le sous-arbre local présenté dans la section 4.2.1 et non plus sur l'arbre d'élimination complet comme sur la version statique. Cette coloration montre, par l'inversion des couleurs dans l'exécution des tâches sur un thread, les vols de travail dans les niveaux supérieurs ou voisins.

On constate sur la figure 5.5 que le premier point important qui améliore les performances du solveur est la gestion des communications. En effet, si on observe la trace de plus près, on voit qu'il existe une ligne supplémentaire par nœud qui correspond à l'exécution du thread de progression des communications dans le mode *ThComm* présenté dans le chapitre 3. Ce thread permet une meilleure progression des communications principalement car ce sont des communications par *Rendez-Vous* et qu'il fait progresser le protocole plus rapidement.

Le deuxième point sur lequel on gagne du temps lors de la factorisation est la suppression massive des temps d'inactivité sur la premier quart de la factorisation numérique. Ceci permet de débloquer plus rapidement les calculs suivants sur lesquels on constate un réordonnancement par rapport à la version statique. Sur les nœuds 3 et 5, on observe également des permutations dans le calcul des super-nœuds violets et rouges qui débloquent des contributions plus rapidement que sur la version avec ordonnancement statique. Les gains observés sur les traces d'exécution se retrouvent sur le temps de factorisation comme le montre le tableau 5.3.

Tab. 5.3 – Comparaison du temps de factorisation en secondes de la matrice 10Millions sur 8 nœuds de 32 cœurs avec les ordonnancements statique et dynamique sur le cluster Vargas de l'IDRIS. Puis, avec trois niveaux de l'arbre d'élimination distribués par blocs et avec la méthode ESP.

	8×32
Version Statique	154
Version Dynamique	138
Version Dynamique + Distribution 2D	132
Version Dynamique + Méthode ESP	130

Ordonnancement dynamique et distribution 2D ou ESP Les deux versions du solveur PaStiX évaluées jusqu'ici ne permettent toujours pas d'obtenir une bonne scalabilité du solveur. En effet, comme le montre les figures 5.4 et 5.5, elles ne découpent pas suffisamment le calcul des derniers super-nœuds pour alimenter tous les cœurs de calcul disponibles. Pour cela, nous avons vu qu'il est nécessaire de passer à une distribution 2D des données. Cependant, le passage

à une distribution 2D des données apporte un surcoût important lors de la phase d'analyse qui simule la factorisation. Le tableau 5.4 récapitule les temps de prétraitement en secondes sur la matrice 10Millions en fonction du nombre de processus.

TAB. 5.4 – Étude du temps d'analyse en secondes sur la matrice 10Millions en fonction du nombre de niveaux de l'arbre d'élimination gérés en 2D. La prédiction est le temps minimal et le temps maximal en secondes nécessaires aux processus pour effectuer la factorisation.

Nb. Niveaux	8 Processus				16 Processus			
	Analyse	Nb. Tâches	Prédiction		Analyse	Nb. Tâches	Prédiction	
			Min	Max			Min	Max
0	41	388941	72,3	90,3	58	389168	24,7	81,8
1	43	459957	82,7	88,6	64	459961	60,2	80,8
2	66	1428395	79,2	82,7	101	1429258	70,6	73,1
3	159	4131521	70,0	70,5	238	4132793	40,9	45,7
4	516	14801342	67,1	67,6	905	14802801	35,2	35,7
5	791	22557280	67,9	69,1	1480	22562495	33,8	34,4

Sur ce tableau, on constate que le temps de calcul de la distribution statique des données explose rapidement avec un faible nombre de niveaux de l'arbre d'élimination distribué en 2D. Il est multiplié par vingt sur huit processus et par trente sur seize avec cinq niveaux distribués en 2D par rapport à une distribution complètement 1D. De même, le nombre de tâches augmente de façon importante mais on constate qu'il ne suit que le nombre de niveaux distribués en 2D. Le redécoupage effectué lors de la distribution en fonction du nombre de processus candidats influe peu sur le nombre de tâches. Enfin, les temps de factorisation prédits pour chaque processus confirment que la distribution 1D des données ne permet pas d'obtenir une répartition équilibrée des données et des calculs, mais elle peut être rapidement rééquilibrée en introduisant quelques niveaux de distribution par blocs.

Le diagramme de la figure 5.6 montre la factorisation de la matrice 10Millions sur huit processus avec deux niveaux de l'arbre d'élimination distribués en 2D. Les tâches du haut de l'arbre sont bel et bien découpées pour exploiter tous les cœurs disponibles en parallèle. Nous n'avons pas affiché les communications sur cette trace pour mieux voir les temps d'inactivité introduits entre les calculs de chaque ensemble de tâches sur le haut de l'arbre. En effet, la distribution des données en 2D introduit de nouvelles communications et répartit sur un nombre plus important de processus le calcul des super-nœuds distribués en 2D. Les calculs découpés sont donc mieux répartis mais on introduit de nouvelles périodes d'inactivité dues aux communications.

En comparaison, le diagramme 5.7 montre la factorisation du même cas test dans les mêmes conditions en remplaçant la distribution statique des données par la méthode ESP de distribution dynamique. On constate qu'avec le même coût de calcul que celui de la distribution 1D, on arrive à compléter le gain de temps sur la factorisation car on alimente plus régulièrement l'ensemble des cœurs disponibles sur chaque nœud et ce en descendant beaucoup plus bas dans l'arbre d'élimination. Cette méthode nous a permis sur ce cas d'obtenir un temps de factorisation de 130 secondes équivalent à celui obtenu en distribuant trois niveaux de l'arbre d'élimination en 2D, 132 secondes (voir tableau 5.3). Cependant, la méthode ESP ne permet pas de récupérer complètement les défauts de la distribution initiale utilisée en raison du trop grand nombre de super-nœuds de taille importante et du nombre de processeurs utilisés.

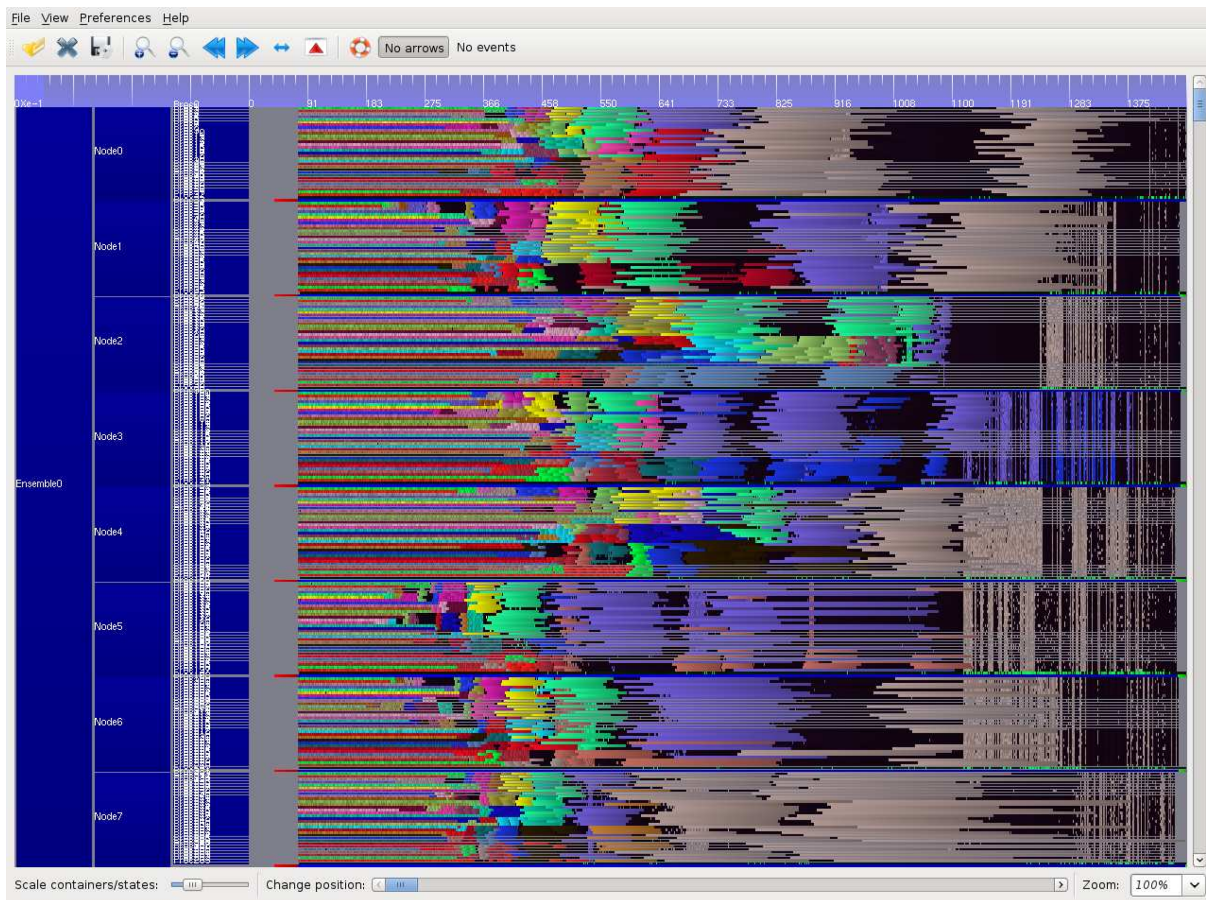


Fig. 5.6 – Diagramme de Gantt de la factorisation sur 8 nœuds de 32 cœurs de la matrice 10Millions avec l'ordonnancement dynamique et deux niveaux de l'arbre d'élimination distribués par blocs.

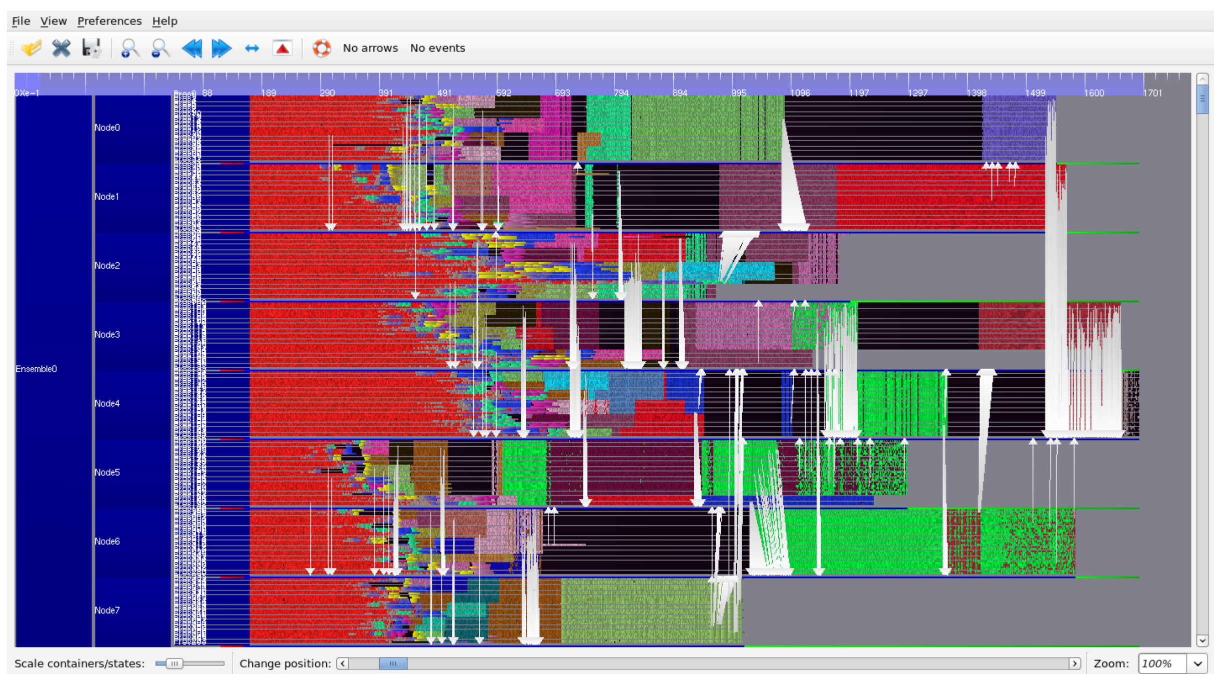


FIG. 5.7 – Diagramme de Gantt de la factorisation sur 8 nœuds de 32 cœurs de la matrice 10Millions avec l'ordonnancement dynamique et la méthode ESP de distribution des calculs à partir d'une distribution 1D des données.

Conclusion et perspectives

Conclusion

La démocratisation des microprocesseurs multi-cœurs a apporté dans le calcul hautes performances de nouvelles architectures aux topologies fortement hiérarchiques et aux accès mémoire non uniformes. Cette tendance, créée par les constructeurs de processeurs comme Intel et AMD pour des raisons de coût et de contrainte de fabrication, continue d'accroître la densité des puces disponibles et on attend pour bientôt celles comportant 16 ou 32 cœurs. Il est ainsi devenu essentiel, pour exploiter au mieux ces ressources, de prendre en compte cette hiérarchie au niveau de la programmation et notamment dans les solveurs de systèmes linéaires par méthode directe qui sont très consommateurs de mémoire et de temps CPU. Pour s'adapter à ces architectures, il existe de nombreuses solutions plus ou moins difficiles à intégrer dans une application donnée comme des bibliothèques de calcul déjà parallélisées ou des directives de compilation qui permettent d'exprimer le parallélisme de l'application. Cependant, peu de solveurs directs — pourtant à la base de nombreuses applications de calcul scientifique — sont actuellement capables d'exploiter pleinement les clusters de machines multi-cœurs.

L'objectif de cette thèse s'inscrivait dans ce contexte et visait à proposer des solutions pour adapter les bibliothèques de résolution de systèmes linéaires creux à ces nouvelles architectures NUMA et de les valider au sein du solveur PaStiX développé à Bordeaux. Les modèles de coût utilisés par PaStiX n'intégraient pas la non-uniformité des accès mémoire propre aux architectures NUMA. L'idée fondamentale de cette thèse était donc de remplacer l'ordonnancement *statique* des calculs du solveur PaStiX par un ordonnancement *dynamique* afin de récupérer les erreurs introduites dans l'ordonnancement par les modèles de coût. Le développement de cet ordonnanceur dynamique a donné lieu à plusieurs travaux préalables : une étude du coût de la localité des données et une refonte du modèle de communication utilisé dans le solveur.

La première étude a montré que le surcoût d'un mauvais placement des données sur ces architectures est très important et augmente avec le nombre de cœurs. Il est donc important de prendre en compte le placement des données dans les applications multi-threadées utilisant ces architectures. Ceci nous a conduit à concevoir une nouvelle procédure d'initialisation des données inspirée du modèle de programmation par passage de messages. Cette solution implémentée dans le solveur PaStiX apporte un gain important, de l'ordre de 20 à 40% sur les architectures utilisées en mémoire partagée.

Le deuxième travail préalable était destiné à permettre un meilleur recouvrement des communications par les calculs pour intégrer l'ordonnancement dynamique et pouvoir réordonnancer plus facilement les communications dans le solveur en fonction de l'ordonnancement des calculs. Nous avons donc modifié le schéma de communication du solveur en introduisant un thread supplémentaire pour gérer la progression des communications, selon le principe de fonctionnement de la bibliothèque OpenMPI [99]. Ce nouveau schéma de communication améliore également

les performances du solveur par rapport au modèle de communication initial. Ces développements ont permis de proposer une version du solveur moins contraignante vis-à-vis du niveau de support des threads requis dans la bibliothèque MPI. Cette version offre la possibilité de diffuser le solveur PaStiX à un plus grand nombre d'utilisateurs qui n'ont pas toujours accès à une bibliothèque de communication offrant un support complet des threads sur le cluster qu'ils utilisent.

À partir de ces travaux préalables, nous avons proposé une solution d'ordonnancement dynamique des calculs basée sur un vol de travail limité dans l'espace pour conserver l'affinité mémoire présente sur les architectures NUMA. En testant plusieurs parcours de l'arbre de vol de travail, cette méthode d'ordonnancement dynamique a permis de constater qu'il est préférable de privilégier au maximum la localité des données sur ce type d'architecture. L'algorithme développé obtient les mêmes performances que l'ordonnancement statique initialement présent dans le solveur PaStiX et permet de les améliorer sur les cas limites où la prédiction réalisée dans la phase d'analyse crée des temps d'inactivité sur les processeurs.

Grâce au développement du logiciel ViTE, il nous a été possible de visualiser les traces d'exécution sur des cas challenges comme la matrice 10Millions. Ces traces ont en particulier mis en évidence les limites de notre ordonnancement dynamique basé sur une distribution 1D statique des données. Pour remédier à ce problème et éviter le coût de calcul important d'une distribution 2D, nous avons proposé une nouvelle méthode de distribution des calculs, appelée ESP. Cette méthode permet de raffiner le grain de calcul pour l'adapter au grand nombre de processeurs utilisés sur ces cas challenges. Elle apporte un gain limité sur les petites matrices mais permet de mieux passer à l'échelle sur les cas plus importants comme la matrice 10Millions. Au final, l'ordonnancement dynamique associé au schéma de calcul ESP permet d'obtenir un gain de 5 à 10% supplémentaires sur les matrices utilisées dans nos résultats et un peu plus de 15% avec la matrice 10Millions sur la machine Vargas.

Les travaux de cette thèse ont donc abouti à des propositions pour exploiter au mieux les architectures NUMA dans les solveurs de systèmes linéaires creux et ces propositions ont été implémentées avec succès dans le solveur PaStiX. La conception d'un solveur direct mieux adapté aux architectures NUMA, et par conséquent aux clusters de demain, est un objectif qui peut désormais être considéré comme atteint. L'ensemble des développements réalisés au cours de cette thèse sont intégrés dans la nouvelle version du solveur PaStiX.

Perspectives

Parmi les perspectives immédiates de ces travaux, nous pouvons citer le besoin d'étendre la validation des résultats sur de grands cas challenges avec des clusters de machines NUMA. Cette validation se fera en collaboration avec CEA/Cesta sur la machine TERA10 et sur la future machine TERA100 à l'aide de la nouvelle version du solveur PaStiX issue de cette thèse. Une nouvelle machine de 96 cœurs va également être installée prochainement à l'Université de Bordeaux 1 sur laquelle nous souhaitons valider nos approches algorithmiques en mémoire partagée.

Par la suite, plusieurs autres perspectives sont ouvertes pour poursuivre ces travaux.

Une meilleur intégration des *"runtimes"* existants. Nous souhaitions au cours de cette thèse nous baser sur les travaux de l'équipe Runtime pour développer l'ordonnancement dynamique à l'aide de l'idée de bulles proposée par Samuel Thibault dans sa thèse [93]. La bibliothèque Marcel et son ordonnancement à base de bulles ont donc été intégrés dans le solveur

PaStiX, cependant les résultats obtenus à l'époque s'étaient révélés en deçà du niveau escompté. Il serait donc intéressant de reprendre l'évaluation de ces travaux avec le nouvel ordonnanceur développé dans BubbleSched pour ForestGOMP [19], à la lumière de l'expérience acquise depuis, à la fois du côté du solveur PaStiX et du côté du support exécutif Marcel. En effet, celui-ci intègre la nouvelle bibliothèque MaMI [46] qui fournit des informations sur la localité des données exploitées. Il choisit ensuite de migrer les threads au plus proche de cette zone ou, inversement, de migrer les données au plus proche du thread. On pourrait de cette façon accroître la localité des données dans les nœuds de l'arbre de vol possédant plusieurs threads candidats.

Étudier la complémentarité entre la méthode ESP et la distribution des données en 2D. Avec l'étude des traces d'exécution de la matrice 10Millions, nous avons observé que la méthode ESP ne peut suffire seule à résoudre les problèmes de scalabilité du solveur lorsqu'on utilise une distribution 1D des données. Nous souhaitons étudier l'apport d'un mélange entre une distribution 2D sur un nombre restreint de niveaux de l'arbre d'élimination pour conserver un coût raisonnable sur le temps d'analyse et la méthode ESP sur le reste de l'arbre d'élimination. L'autre aspect de la méthode ESP qu'il faudrait étudier est le choix adaptatif du seuil de création des tâches dynamiques. En effet, nous avons vu que le nombre de tâches créées diminue rapidement avec le nombre de processus utilisés car la phase d'analyse découpe plus finement les blocs-colonnes dans cette configuration. De même, le choix d'un seuil trop petit crée trop de tâches, ce qui apporte un surcoût important lors de l'insertion dans les files de tâches disponibles et de leur parcours.

Architectures hétérogènes Enfin, une dernière perspective offerte par ces travaux est l'étude de l'utilisation de machines hétérogènes. Actuellement, le domaine s'oriente vers l'exploitation massive des GPU. À première vue, cela ne semble pas intéressant pour les solveurs directs de matrices creuses car la taille des blocs utilisés est trop petite pour être efficace sur GPU. Cependant nous savons que les derniers super-nœuds des matrices à plus de 10 millions d'inconnues sont des matrices denses de grande taille. Il serait intéressant d'étudier une solution qui bascule sur l'utilisation des GPUs pour ces derniers super-nœuds en utilisant, par exemple, une bibliothèque de factorisation de matrice dense externe comme le fait MUMPS avec ScaLAPACK ou en intégrant la bibliothèque StarPU [16] qui permet d'ordonnancer les tâches sur CPU ou GPU indépendamment. L'ordonnancement dynamique intégré dans le solveur PaStiX constituerait une bonne base pour cette étude.

Annexe A

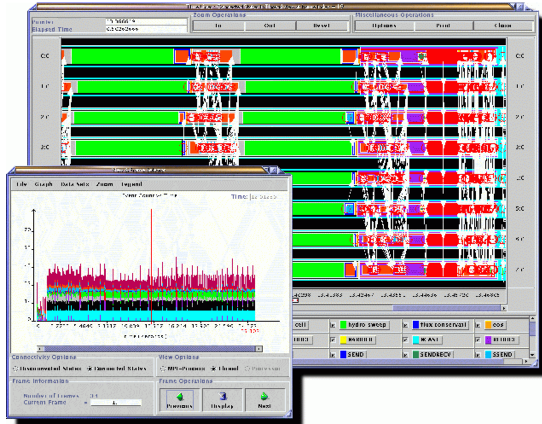
ViTE : Un logiciel de visualisation de traces d'exécution

Le développement d'applications à grande échelle et principalement le développement d'algorithmes d'ordonnancement nécessitent des outils de visualisation pour en comprendre le fonctionnement et détecter leurs défauts. Pour cela, plusieurs outils existent avec chacun leurs spécificités. On peut citer entre autres : ParaGraph [57], Jumpshot [71], Pajé [28] ou Vampir [20] dont on peut voir quelques images sur la figure A.1.

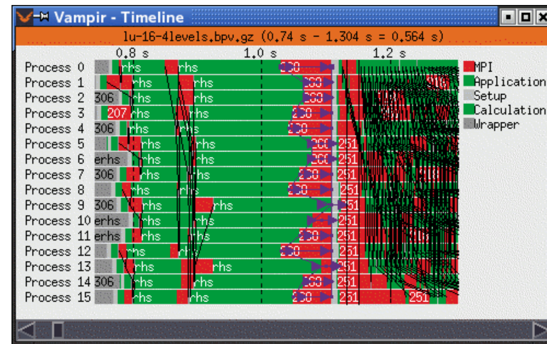
Paragraph utilise le format de traces de la bibliothèque PICL [43]. Il s'agit d'un des logiciels les plus anciens de visualisation de traces et il n'est plus maintenu depuis 1994. Jumpshot est un logiciel java permettant d'afficher les traces au format Slog [23]. Ce format de traces est avantageux car il est déjà intégré dans les bibliothèques MPI. Il suffit, pour l'utiliser, de recompiler la bibliothèque MPI en activant l'option MPE qui permet de tracer dans ce format le temps passé dans les routines de communication et les communications elles-mêmes. Ce format est donc facile d'intégration, mais il est fortement axé sur l'étude du temps passé dans les routines MPI. De plus, Jumpshot n'est pas des plus conviviaux à l'utilisation. Pajé est un logiciel codé en Objective C et GNUStep et initialement développé au laboratoire ID Imag. Il n'est plus maintenu depuis quelques années mais utilise un format de traces très générique, Pajé Trace Format [29], permettant de décrire facilement l'ensemble des événements, états ou communications d'une application donnée. Enfin, le dernier logiciel cité, Vampir, est le logiciel le plus utilisé actuellement pour la visualisation de traces d'exécution. Il utilise depuis 2008 le format de trace Open Trace Format [72] qui dans sa description est similaire au format Pajé. La différence vient de la possibilité donnée au programmeur de ne pas spécifier les informations nécessaires pour décrire un événement dans le format Pajé. L'autre différence est le format de sortie. L'OTF a un format de données compressées, alors que Pajé utilise un format texte. Il propose également une bibliothèque pour faciliter l'instrumentation du code.

Le solveur PaStiX était déjà instrumenté avant les travaux de cette thèse pour obtenir des traces au format PICL afin de les visualiser avec Paragraph. Cependant, comme ce logiciel n'est plus maintenu, nous avons décidé de changer le format de traces utilisé dans le solveur. Le format Pajé était alors le format le plus intéressant. Nous avons rapidement été confrontés au problème de la taille des fichiers de sortie générés, le logiciel Pajé ne permettant pas un affichage fluide de traces comportant plus de quelques centaines de milliers d'événements. Pour remédier à ce problème, nous avons proposé comme projet de deuxième année la création d'un nouveau logiciel de visualisation de traces au format Pajé basé sur des technologies plus récentes aux

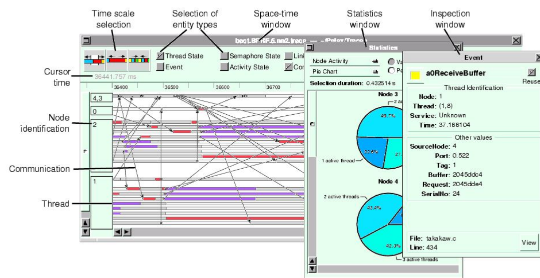
étudiants de l'ENSEIRB[9]. Ce projet a été réalisé par sept étudiants de l'ENSEIRB : K. Coulomb, J. Jazeix, O. Lagrasse, P. Noisette, A. Redondy et C. Vuchener que j'ai encadré avec trois autres doctorants de l'INRIA Bordeaux - Sud-Ouest : C. Augonnet, J. Gaidamour et N. Richart qui avaient les mêmes besoins de visualisation sur leur code respectif. Ce projet a abouti avec succès au développement du logiciel ViTE qui nous a permis de visualiser des traces de plusieurs millions d'événements.



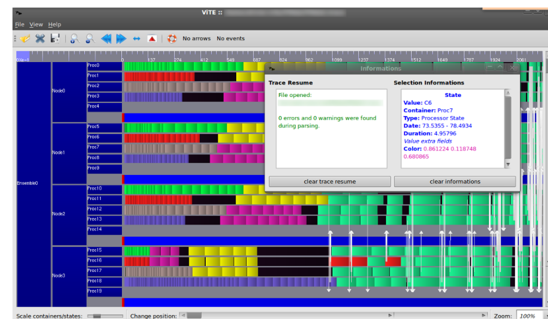
(a) Jumpshot



(b) Vampir



(c) Pajé



(d) Vite

Fig. A.1 – Exemples de traces d'exécution sous différents logiciels existants.

Le logiciel ViTE a été développé en C++ avec une interface en QT. Il est disponible sous Linux, Mac et Windows à l'adresse suivante : `http://vite.gforge.inria.fr` avec sa documentation [25]. Il est également directement inclus dans la distribution Debian et est distribué sous licence Cecill-C pour le rendre accessible à tous. L'avantage principal de ce logiciel est son affichage et la navigation fluide qu'il fournit même avec des traces d'exécution de taille importante grâce à l'utilisation de la bibliothèque OpenGL. Une autre fonctionnalité importante de ViTE est la possibilité d'exporter directement la trace affichée dans le format vectoriel SVG et il sera possible d'exporter en JPG dans la prochaine version. ViTE fournit également une interface de visualisation de statistiques sur les données affichées qui peuvent être exportées dans un format externe pour une meilleure intégration dans des documents de présentation.

[9] `http://www.enseirb-matmeca.fr`, École Nationale Supérieure d'Électronique, Informatique et Radiocommunications de Bordeaux.

Annexe B

Utilisation des options implémentées dans le solveur PaStiX au cours de cette thèse

Les options du solveur PaStiX sont de deux natures : les options de compilation et les paramètres d'exécution. Pour les premières, elles doivent être spécifiées dans le fichier `config.in` utilisé par le `Makefile` pour compiler le solveur. Les secondes sont des paramètres contenus dans deux vecteurs appelés `iparm` et `dparm`, le premier pour les paramètres *entiers* et le second pour les paramètres *réels*.

Les développements réalisés au cours de cette thèse sont principalement accessibles par des options de compilations :

-DNUMA_ALLOC
Active la nouvelle méthode d'allocation.
Cette option est activée par défaut dans le `config.in` initial.

-DPASTIX_THCOMM
Active la création d'un thread supplémentaire pour faire progresser la réception des communications.
Cette option n'est pas activée par défaut.

-DPASTIX_FUNNELED
Active le mode "dégradé" du solveur pour les bibliothèques MPI ne supportant par le niveau MPI_THREAD_MULTIPLE.
Cette option n'est pas activée par défaut. Elle active l'option **PASTIX_THCOMM**.

-DPASTIX_DYNSCHED
Active l'ordonnancement dynamique dans le solveur.
Elle active automatiquement l'option **PASTIX_THCOMM**.

-DPASTIX_BUBBLESCHED
Active le support de l'ordonnanceur BubbleSched dans le solveur. Nécessite de remplacer la bibliothèque de threads par la bibliothèque Marcel.
Elle active automatiquement l'ordonnancement dynamique.

La méthode ESP s'active au contraire grâce aux paramètres d'exécution.
Elle nécessite cependant la présence de l'ordonnancement dynamique et de l'option de compilation **PASTIX_DYNSCHED**.

IPARM_ESP
Active l'ESP si positionné à API_YES. (Par défaut : API_NO) Cette option permet de créer dynamiquement des tâches de calcul 2D au sein des processus MPI et ainsi conserver la possibilité d'utiliser l'étape de résolution du solveur sur la distribution 1D des données.

IPARM_ESP_THRESHOLD
Seuil de création des tâches 2D par la méthode ESP, il correspond à la surface du bloc résultant de la multiplication des deux blocs extra-diagonaux. (Par défaut : $16384 = 128^2$)

IPARM_FACTORIZATION
Permet de choisir le type de factorisation : API_FACT_LLT, API_FACT_LDLT ou API_FACT_LU. (Remarque : API_FACT_LDLT désactive automatiquement la méthode ESP)

IPARM_THCOMM_MODE
Cette option permet de spécifier le mode de communication utilisé lorsque le solveur est compilé avec l'option **PASTIX_THCOMM** et sans l'option **PASTIX_FUNNELED**.
API_THCOMM_DISABLE correspond au fonctionnement standard de la version originale.
API_THCOMM_ONE crée un seul thread de communication.
API_THCOMM_DEFINED crée autant de threads de communication que spécifié par le paramètre IPARM_NB_THREAD_COMM. La configuration par défaut utilise un seul thread de communication mais il est peut être intéressant d'utiliser plusieurs threads de communication lorsque le nombre d'échanges par processus augmente fortement.
API_THCOMM_NBPROC crée un thread de communication par thread de calcul et lui dédie la réception des données associées.

IPARM_NB_THREAD_COMM
Spécifie le nombre de threads de communication si IPARM_THCOMM_MODE est positionné à API_THCOMM_DEFINED.

Bibliographie

[1] Grid'5000. `https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home`.

[2] *Intel MPI Library*. `http://software.intel.com/en-us/intel-mpi-library/`.

[3] Leibniz-rechenzentrum lrz. `http://www.lrz.de`.

[4] *SDP specification*. `http://www.rdmaconsortium.org`.

[5] Thread Building Blocks. `http://www.intel.com/software/products/tbb/`.

[6] Top500 supercomputer sites. `http://www.top500.org`.

[7] *MPICH-2 Home Page*, 2007. `http://www.mcs.anl.gov/mpi/mpich/`.

[8] AMD : The AMD Core Math Library. `http://developer.amd.com/cpu/libraries/acml/Pages/default.aspx`.

[9] P. Amestoy, T. A. Davis et I. S. Duff : An Approximate Minimum Degree Ordering Algorithm. *SIAM J. Matrix Anal. and Appl.*, pages 886–905, 1996.

[10] P. Amestoy, I. S. Duff, J. Koster et J.-Y. L'Excellent : A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIMAX*, 23(1):15–41, 2001.

[11] P. Amestoy, I. S. Duff et J.-Y. L'Excellent : Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods Appl. Mech. Eng*, 184:501–520, 1998.

[12] J. Antony, P. P. Janes et A. P. Rendell : Exploring thread and memory placement on NUMA architectures : Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport. *In HiPC*, pages 338–352, 2006.

[13] Argonne National Laboratory. *Analysis of thread safety needs of MPI routines*. `http://www.mcs.anl.gov/mpi/mpich2/developer/design/threadlist.htm`.

[14] A. Ashcraft : The fan-both family of column-based distributed Cholesky factorization algorithms. *Graph Theory and Sparse Matrix Computation, IMA*, 56:159–190, 1993.

[15] A. Ashcraft, S. C. Eisenstat et J. W.-H. Liu : A fan-in algorithm for distributed sparse numerical factorization. *SIAM Journal on Scientific and Statistical Computing*, 11:593–599, 1990.

[16] C. Augonnet, S. Thibault, R. Namyst et P.-A. Wacrenier : StarPU : A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *In Proceedings of the 15th International Euro-Par Conference, Lecture Notes in Computer Science*, volume 5704 de *Lecture Notes in Computer Science*, pages 863–874, Delft, The Netherlands, août 2009. Springer.

[17] O. Aumage, E. Brunet, N. Furmento et R. Namyst : Newmadeleine : a fast communication scheduling engine for high performance networks. *In CAC 2007 : Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2007*, March.

[18] 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 et 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)*, page 5, Washington, DC, USA, 1996. IEEE Computer Society.

[19] F. Broquedis, N. Furmento, B. Goglin, R. Namyst et P.-A. Wacrenier : Dynamic Task and Data Placement over NUMA Architectures : an OpenMP Runtime Perspective. *In Evolving OpenMP in an Age of Extreme Parallelism, 5th International Workshop on OpenMP, IWOMP 2009*, volume 5568 de *Lecture Notes in Computer Science*, pages 79–92, Dresden, Germany, June 2009. Springer.

[20] H. Brunst, D. Kranzlmüller, M. S. Muller et W. E. Nagel : Tools for scalable parallel program analysis: ; vampir ng, marmot, and dewiz. *Int. J. Comput. Sci. Eng.*, 4(3):149–161, 2009.

[21] Bull : MPIBull2. `http://www.bull.com`.

[22] W. Carlson, J.M. Draper, D.E. Culler, K. Yelick, E. Brooks et K. Warren : Introduction to UPC and Language Specification. Rapport technique CCS-TR-99-157, George Mason University, mai 1999.

[23] A. Chan, W. Gropp et E. Lusk : An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. *Scientific Programming*, 16(2-3):155–165, 2008.

[24] P. Charrier et J. Roman : Algorithmique et calculs de complexité pour un solveur de type dissections emboîtées. *Numerische Mathematik*, 55:463–476, 1989.

[25] K. Coulomb, J. Jazeix, O. Lagrasse, P. Noisette, A. Redondy et C. Vuchener : *ViTE User Manual*, 2007.

[26] E. Cuthill et J. McKee : Reducing the bandwidth of sparse symmetric matrices. *In Proceedings of the 1969 24th national conference*, pages 157–172, New York, NY, USA, 1969. ACM.

[27] T. A. Davis : University of Florida sparse matrix collection. *NA Digest*, 92, 1994.

[28] J. Chassin de Kergommeaux et B. de Oliveira Stein : Flexible performance visualization of parallel and distributed applications. *Future Gener. Comput. Syst.*, 19(5):735–747, 2003.

[29] J. Chassin de Kergommeaux, B. de Oliveira Stein et G. Mounié : Paje input data Format. Technical Report draft, INRIA Rhônes-Alpes, 2003.

[30] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li et J. W. H. Liu : A supernodal approach to sparse partial pivoting. Rapport technique UCB/CSD-95-883, EECS Department, University of California, Berkeley, Jul 1995.

[31] J. R. Diamond, B. Robatmili, S. W. Keckler, R. A. van de Geijn, K. Goto et 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*, pages 63–72, New York, NY, USA, 2008. ACM.

[32] J. Dongarra : Performance of various computers using standard linear equations software (linpack benchmark report). Rapport technique, University of Tennessee, 2009.

[33] J. Dongarra, J. Du Croz, S. Hammarling et I. S. Duff : A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, 1990.

[34] J. Dongarra, J. Du Croz, . Hammarling et R. J. Hanson : An extended set of fortran basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14(1):1–17, 1988.

[35] I. S. Duff : A review of frontal methods for solving linear systems. *Computer Physics Communications*, 1996.

[36] I. S. Duff : Sparse numerical linear algebra : direct methods and preconditioning. Rapport technique TR/PA/96/22, CERFACS, 1996.

[37] I. S. Duff, A. M. Erisman et J. K. Reid : *Direct methods for sparse matrices.* Oxford University Press, Inc., New York, NY, USA, 1986.

[38] L. Facq et J. Roman : Algèbre linéaire creuse : distribution par bloc pour une factorisation parallèle de Cholesky. *In* G. Authié, J. M. Garcia, A. Ferreira, J. L. Roch, G. Villard, J. Roman, C. Roucairol et B. Virot editors, éditeurs : *Parallélisme et applications irrégulières*, pages 135–147. Hermès, 1995.

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

[40] F.Song, A. YarKhan et J. Dongarra : Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. Technical Report UT-CS-09-638, Univ. of Tennessee Computer Science, avril 2009.

[41] J. Gaidamour et P. Hénon : A parallel direct/iterative solver based on a Schur complement approach. *In* IEEE SEP, éditeur : *IEEE 11th International Conference on Computational Science and Engineering*, pages page 98–105, Sao Paulo Brésil, 2008. 8 pages double colonnes.

[42] K. A. Gallivan et al. : Parallel algorithms for matrix computations. *SIAM, Philadelphia, PA*, 1990.

[43] G. A. Geist, M. T. Heath, B. W. Peyton et P. H. Worley : PICL : a portable instrumented communication library. Rapport technique ORNL/TM-11130, Oak Ridge National Laboratory, July 1990.

[44] G. A. Geist et E. Ng : Task scheduling for parallel sparse Cholesky factorization. *Internat. J. Parallel Programming*, 18(4):291–314, 1989.

[45] A. George et J. W.-H. Liu : *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[46] B. Goglin et N. Furmento : Memory migration on next-touch. *In Proceedings of the Linux Symposium*, pages 101–110, Montreal, Canada, July 2009.

[47] Brice Goglin et Nathalie Furmento : Enabling high-performance memory-migration in linux for multithreaded applications. *In MTAAP'09 : Workshop on Multithreaded Architectures and Applications, held in conjunction with IPDPS 2009*, Rome, Italy, May 2009. IEEE Computer Society Press.

[48] K. Goto et R. A. van de Geijn : Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):1–25, 2008.

[49] R. L. Graham, T. S. Woodall et J. M. Squyres : Open MPI : A Flexible High Performance MPI. *In The 6th Annual International Conference on Parallel Processing and Applied Mathematics*, 2005.

[50] L. Grigori, J. A. Demmel et X. S. Li : Parallel Symbolic Factorization for Sparse LU with Static Pivoting. *SIAM J. Sci. Comput.*, 29(3):1289–1314, 2007.

[51] W. Gropp et R. Thakur : Thread-safety in an MPI implementation : Requirements and analysis. *Parallel Computing*, 33(9):595–604, 2007.

[52] A. Gupta : Recent progress in general sparse direct solvers. *In LNCS*, volume 2073, pages 823–840, 2001.

[53] A. Gupta : A shared- and distributed-memory parallel general sparse direct solver. *Appl. Algebra Eng., Commun. Comput.*, 18(3):263–277, 2007.

[54] A. Gupta, G. Karypis et V. Kumar : Highly scalable parallel algorithms for sparse matrix factorization. IEEE Trans. on Parallel and Distributed Systems, 8(5):502–520, mai 1997.

[55] A. Gupta et V. Kumar : A scalable parallel algorithm for sparse matrix factorization. Rapport technique 94-19, Department of Computer Science, University of Minnesota, Minneapolis, 1994.

[56] A. Gupta, Vinpin Kumar et Mahesh Joshi : WSSMP :a high-performance shared- and distributed-memory parallel sparse symmetric linear equation solver. *PARA'98 Workshop on Applied Parallel Computing in Large Scale Scientific and Industrial Problems*, June 1998.

[57] M. T. Heath : Paragraph : A tool for visualizing performance of parallel programs. Rapport technique, 1993.

[58] M. T. Heath, Esmond Ng et Barry W. Peyton : Parallel algorithms for sparse linear systems. *SIAM Rev.*, 33(3):420–460, 1991.

[59] P. Hénon : *Distribution des Données et Régulation Statique des Calculs et des Communications pour la Résolution de Grands Systèmes Linéaires Creux par Méthode Directe*. Thèse de doctorat, LaBRI, Université Bordeaux I, Talence, Talence, France, novembre 2001.

[60] P. Hénon, P. Ramet et J. Roman : PaStiX : A Parallel Sparse Direct Solver Based on a Static Scheduling for Mixed 1D/2D Block Distributions. *In Irregular'2000*, volume 1800 de *LNCS*, pages 519–525, Cancun, Mexique, mai 2000.

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

[62] P. Hénon, P. Ramet et J. Roman : On using an hybrid MPI-Thread programming for the implementation of a parallel sparse direct solver on a network of SMP nodes. *In PPAM'05*, volume 3911 de *LNCS*, pages 1050–1057, Poznan, Pologne, septembre 2005.

[63] J. D. Hogg, J. K. Reid et J. A. Scott : A DAG-based sparse Cholesky solver for multicore architectures. 2009.

[64] W. Huang, G. Santhanaraman, H.-W. Jin, Q. Gao et D. K. x. D. K. Panda : Design of High Performance MVAPICH2 : MPI2 over InfiniBand. *In CCGRID '06 : Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, pages 43–48, Washington, DC, USA, 2006. IEEE Computer Society.

[65] Intel : Intel Math Kernel Library. `http://software.intel.com/en-us/intel-mkl`.

[66] M. Joshi, G. Karypis, V. Kumar, A. Gupta et Gustavson F. : PSPASES : Scalable Parallel Direct Solver Library for Sparse Symmetric Positive Definite Linear Systems. Rapport technique, University of Minnesota and IBM Thomas J. Watson Research Center, mai 1999.

[67] C. L. Lawson, R. J. Hanson, D. R. Kincaid et F. T. Krogh : Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, 1979.

[68] X. S. Li et J. W. Demmel : SuperLU_DIST : A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Mathematical Software*, 29(2):110–140, June 2003.

[69] J. W.-H. Liu : The role of elimination trees in sparse factorization. *SIAM J. Matrix Anal. Appl.*, 11:134–172, 1990.

[70] H. Löf et S. Holmgren : Affinity-on-next-touch : increasing the performance of an industrial PDE solver on a cc-NUMA system. *In ICS '05 : Proceedings of the 19th annual international conference on Supercomputing*, pages 387–392, New York, NY, USA, 2005. ACM.

[71] E. Lusk et A. Chan : Early experiments with the OpenMP/MPI hybrid programming model. *In* R. Eigenmann et B. R. de Supinski, éditeurs : *OpenMP in a New Era of Parallelism*, volume 5004 de *Lecture Notes in Computer Science*, pages 36–47. Springer, 2008. IWOMP, 2008.

[72] A. D. Malony et W. E. Nagel : The open trace format (OTF) and open tracing for HPC. *In SC '06 : Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 24, New York, NY, USA, 2006. ACM.

[73] T. Mattson et R. EigenMann : OpenMP : An API for Writing Portable SMP Application Software. *In SuperComputing 99 Conference*, novembre 1999.

[74] G. E. Moore : Cramming more components onto integrated circuits. pages 56–59, 2000.

[75] MPI-2.0 Standards. `http://www.mpi-forum.org/docs/docs.html`, 1997.

[76] Myricom Inc. : Myrinet EXpress (MX) : A High Performance, Low-level, Message-Passing Interface for Myrinet, 2003. `http://www.myri.com/scs/`.

[77] R. Namyst : *Contribution à la conception de supports exécutifs multithreads performants.* Habilitation à diriger des recherches, Université Claude Bernard de Lyon, pour des travaux effectués à l'école normale supérieure de Lyon, December 2001.

[78] E. Ng et B. W. Peyton. : Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM J. Sci. Comput.*, 14:1034–1056, 1993.

[79] F. Pellegrini et J. Roman : Sparse matrix ordering with scotch. *In Proceedings of HPCN'97*, numéro 1225 de Lecture Notes in Computer Science, pages 370–378. Springer Verlag, April 1997.

[80] F. Pellegrini, J. Roman et P. Amestoy : Hybridizing Nested Dissection and Halo Approximate Minimum Degree for Efficient Sparse Matrix Ordering. *In Proceedings of Irregular'99*, numéro 1586 de Lecture Notes in Computer Science, pages 986–995. Springer Verlag, April 1999. Extended paper appeared in Concurrency : Practice and Experience, 12 :69-84, 2000.

[81] M. Pérache : *Contribution à l'élaboration d'environnements de programmation dédiés au calcul scientifique hautes performances.* Thèse de doctorat, Univ. de Bordeaux, octobre 2006.

[82] P. Ramet : *Optimisation de la Communication et de la Distribution des Données pour des Solveurs Parallèles Directs en Algèbre Linéaire Dense et Creuse.* Thèse de doctorat, LaBRI, Université Bordeaux I, Talence, France, janvier 2000.

[83] J. K. Reid : Direct methods for sparse matrices. *In* Evans, D. J., éditeur : *Software for Numerical Mathematics*, pages 29–47. Academic Press, 1974.

[84] H. Richardson : High Performance Fortran : history, overview and current developments. Rapport technique, Thinking Machines Corporation, 1996.

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

[86] D. J. Rose, R. E. Tarjan et G. S. Lueker : Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput*, (5):266–283, 1976.

[87] E. Rothberg : Performance of panel and block approaches to sparse Cholesky factorization on the iPSC/860 and Paragon multicomputers. *SIAM J. Sci. Comput.*, 17(3):699–713, mai 1996.

[88] E. Rothberg et A. Gupta : An efficient block-oriented approach to parallel sparse Cholesky factorization. *SIAM J. Sci. Comput.*, 15(6):1413–1439, novembre 1994.

[89] E. Rothberg et R. Schreiber : Improved load distribution in parallel sparse Cholesky factorization. *In Proceedings of Supercomputing'94*, pages 783–792. IEEE, 1994.

[90] Y. Saad : *Iterative Methods For Sparse Linear Systems*. Ed. PWS publishing Compagny, 1996.

[91] O. Schenk et K. Gärtner : Solving unsymmetric sparse systems of linear equations with pardiso. *Future Gener. Comput. Syst.*, 20(3):475–487, 2004.

[92] R. Schreiber : Scalability of sparse direct solvers. Technical Report TR 92.13, RIACS, NASA Ames Research Center, mai 1992.

[93] S. Thibault : *Ordonnancement de processus légers sur architectures multiprocesseurs hiérarchiques : BubbleSched, une approche exploitant la structure du parallélisme des applications*. Thèse de doctorat, Université Bordeaux 1, 351 cours de la Libération — 33405 TALENCE cedex, décembre 2007. 128 pages.

[94] S. Thibault, F. Broquedis, B. Goglin, R. Namyst et P.-A. Wacrenier : An Efficient OpenMP Runtime System for Hierarchical Architectures. *In International Workshop on OpenMP (IWOMP)*, pages 148–159, Beijing Chine, 2007.

[95] W. F. Tinney et J. W. Walker : Direct solutions of sparse network equations by optimally ordered triangular factorization. *J. Proc. IEEE*, 55:1801–1809, 1967.

[96] F. Trahay, A. Denis, O. Aumage et R. Namyst : Improving Reactivity and Communication Overlap in MPI using a Generic I/O Manager. *In* F. Cappello, T. Herault et J. Dongarra, éditeurs : *EuroPVM/MPI*, volume Recent Advances in Parallel Virtual Machine and Message Passing Interface de *Lecture Notes in Computer Science*, pages 170–177. Springer, 2007.

[97] R. Clint Whaley et J. Dongarra : Automatically tuned linear algebra software. Rapport technique 131, LAPACK Working Note, décembre 1997.

[98] R. Clint Whaley, A. Petitet et J. Dongarra : Automated empirical optimization of software and the ATLAS project. Rapport technique 147, LAPACK Working Note, septembre 2000.

[99] T. S. Woodall, R. L. Graham, R. H. Castain, D. J. Daniel, M. W. Sukalski, G. E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett et A. Lumsdaine : Open MPI's TEG Point-to-Point Communications Methodology : Comparison to Existing Implementations. *In Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 105–111, Budapest, Hungary, September 2004.

Liste des publications

Revues internationales avec comité de lecture :

[1] M. Faverge, P. Ramet. *A NUMA Aware Scheduler for a Parallel Sparse Direct Solver.* soumis à Parallel Computing, 2008.

Congrès internationaux avec sélection :

[2] M. Faverge, X. Lacoste et P. Ramet. *A NUMA Aware Scheduler for a Parallel Sparse Direct Solver.* Proceedings de PMAA'2008, Neuchâtel, Suisse, Juin 2008.

[3] M. Faverge et P. Ramet. *Dynamic Scheduling for sparse direct Solver on NUMA architectures.* Proceedings LNCS de PARA'2008, Trondheim, Norvège, Mai 2008.

Workshops internationaux avec sélection :

[4] M. Faverge. *Dynamic Scheduling for Sparse Direct Solver on NUMA and Multicore Architectures.* Présentation lors des Sparse Days 2009, Cerfacs, Toulouse, France, Juin 2009.

Congrès nationaux avec sélection :

[5] M. Faverge. *Vers un solveur de systèmes linéaires creux adapté aux machines NUMA.* Proceedings de RENPAR'19, Toulouse, France, Septembre 2009.

Workshops nationaux avec sélection :

[6] M. Faverge et P. Ramet. *Un ordonnancement dynamique* NUMA-aware *pour un solveur linéaire direct.* présentation aux Journées Informatique Massivement Multiprocesseur et Multicœur, Rocquencourt, Février 2009.