

**THÈSE**  
PRÉSENTÉE À  
**L'UNIVERSITÉ BORDEAUX I**  
ÉCOLE DOCTORALE DE MATHÉMATIQUES ET  
D'INFORMATIQUE  
Par **Pierre RAMET**  
POUR OBTENIR LE GRADE DE  
**DOCTEUR**  
SPÉCIALITÉ : INFORMATIQUE

---

**Optimisation de la Communication et de la Distribution des  
Données pour des Solveurs Parallèles Directs en Algèbre  
Linéaire Dense et Creuse.**

---

Soutenue le : 12 janvier 2000

Après avis des rapporteurs :

Patrick Amestoy Maître de Conférence  
Michel Cosnard . Professeur

Devant la commission d'examen composée de :

Patrick Amestoy	Maître de Conférence	Rapporteur
Michel Cosnard .	Professeur .....	Président
Frédéric Despres	Chargé de Recherche	Rapporteur
Jack Dongarra ..	Professeur .....	Examinateur
Iain Duff .....	Professeur .....	Examinateur
Jean Roman ....	Professeur .....	Directeur de thèse



# Remerciements

Je tiens vivement à remercier ici :

Patrick Amestoy, maître de conférences à l'IRIT-Toulouse, pour avoir bien voulu rapporter sur ce travail. Ses nombreuses remarques m'ont permis de compléter utilement ce document et son aide a été précieuse ;

Michel Cosnard, professeur, directeur de l'unité de recherche de l'INRIA-Lorraine, pour l'honneur qu'il m'a fait en acceptant de rapporter sur ce travail et présider ce jury ;

Frédéric Despres, chargé de recherche à l'INRIA-Rhône-Alpes, pour avoir encadré cette thèse, pour son dynamisme et pour m'avoir donné le goût de poursuivre dans cette voie ;

Jack Dongarra, professeur à l'université du Tennessee (Etats-Unis), pour avoir bien voulu détourner son avion et accepté d'être membre de ce jury. Je lui suis très reconnaissant de l'attention qu'il a portée à mes travaux ;

Iain Duff, directeur de recherche au CERFACS-Toulouse et au Rutherford Appleton Laboratory (Grande-Bretagne), pour avoir bien voulu être membre de ce jury, pour ses questions et sa vision très large et précise du domaine ;

Avec un remerciement tout particulier à Jean Roman, professeur à l'ENSERB-Bordeaux, pour avoir encadré et dirigé ce travail avec un remarquable sérieux et une grande compétence pendant ces trois années ; pour tous ces souvenirs, en espérant qu'il y en aura d'autres, qui ont marqué ce parcours, le verre de Montparnasse, la côte de bœuf du Dégustoire, les moules de Claouey, le ping-pong à Aussois ...

Ce travail n'aurait pu se faire dans d'aussi bonnes conditions sans l'intervention, consciente ou inconsciente, de personnes que je veux également remercier ici.

Ma première pensée va à Pascal (prépare le sombrero et le pulco !) et à David pour l'aide immense qu'ils m'ont apportée. Cette pensée va aussi à François avec qui c'est mardi-gras tous les jours et à toute l'équipe ALiENor du LaBRI de m'avoir accueilli durant ces années. Merci à Serge pour avoir partagé avec moi cette fameuse journée du 12 janvier. Merci aussi à Kubrick et Stanley, sans eux le développement aurait été beaucoup plus difficile.

Je remercie Macha et Jean-François pour cette ambiance à la fois sérieuse et vivante qui fait le charme du bureau 119. Un grand merci à tous les anciens de la salle A, Patrice, Olivier, Hélène, Benoît, et François ; ainsi qu'à leurs nouveaux remplaçants, Guillaume, Davy et le président Valère pour son aide la veille de la soutenance.

Merci à Frédérique pour son accueil lors de mes séjours à Lyon et qui attend un heureux évènement alors que j'écris ces lignes. Merci aussi à Frédéric, Jean-Christophe et Laurent qui m'ont guidé en me précédant de quelques semaines.

Une pensée particulière à Lionel et Delphine sur qui j'ai toujours pu compter, ainsi qu'aux anciens de l'ENSERB, en particulier, Bruno, Damien, Estelle et Anne-Laure.

A la joyeuse bande des Lauréades, les deux Nicos, Sandra, Manue, Katia, Rachel, les deux Valéries, Laurence et la paire Pastis/Nico : un gros merci aussi pour les sorties du jeudi soir et autres jours de la semaine.

Merci beaucoup aux membres de l'équipe Analyse Numérique du service informatique du CEA/CESTA qui, pendant l'année que j'ai passée parmi eux, m'ont permis de ne pas interrompre totalement mes travaux et qui ont motivé l'orientation de mes recherches. Merci donc à Michel, Murielle, Jean-Jacques, Bruno malgré ses hurlements dans le couloir, et MoMo :)

Merci à mes parents Martine et Alain ainsi qu'à ma sœur Catherine pour leurs conseils et leur soutien constant depuis le début de cette thèse et bien avant aussi.

Enfin, un remerciement spécial à Marieve pour avoir supporté les soirées et nuits blanches à terminer le travail de la veille, surtout sur la fin ...

# Optimisation de la Communication et de la Distribution des Données pour des Solveurs Parallèles Directs en Algèbre Linéaire Dense et Creuse

**Résumé :** Cette thèse traite des problèmes du calcul haute performance et plus spécifiquement du calcul parallèle scientifique pour des applications irrégulières en vraie grandeur.

Dans une première partie, nous présentons une contribution aux optimisations du recouvrement calcul/communication sur des architectures parallèles à mémoire distribuée, avec en particulier le calcul du grain optimal et de la taille optimale des paquets à communiquer. Nous nous sommes également intéressés au calcul de la granularité maximisant le recouvrement calcul/communication pour l'algorithme de factorisation de Cholesky pour des matrices pleines en exploitant l'irrégularité due à la symétrie de cette matrice. Ces travaux ont débouché sur le développement d'une bibliothèque portable intégrant ces mécanismes de découpage des messages.

La seconde partie décrit un ordonnancement statique des calculs pour le problème de la résolution parallèle directe de grands systèmes linéaires creux, conduisant au masquage quasi-total des communications. La mise en œuvre de ces travaux nous a conduit à implémenter un solveur direct parallèle pour la factorisation de Cholesky par blocs, avec des distributions 1D et/ou 2D, intégrant l'approche “Fan-In” et présentant des performances qui se comparent très favorablement aux meilleurs solveurs parallèles directs actuels.

**Mots clés :** calcul haute performance, recouvrement calcul/communication, algorithme macro-pipeline, taille optimale de paquet, factorisation de Cholesky, résolution parallèle directe, matrice creuse.

**Discipline :** Informatique

---

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



# Optimization of Communication and Data Distribution for Parallel Direct Solvers in Dense and Sparse Linear Algebra

**Abstract :** This thesis deals with the high performance computation problems and more specifically with those of scientific parallel computation for irregular real-world applications.

In the first part, we describe a method for overlapping communications on parallel computers with distributed memory. This method has resulted in a generic computation scheme for the optimal packet size. We also tackle the problem of finding the optimal computation grain for the Cholesky factorization algorithm for dense matrices. The goal of this study is to exploit the irregularity induced by the matrix symmetry. Based on this work we have developed a portable software library providing an efficient application context for these techniques.

The second part of this thesis presents and analyses a general algorithm for the computation of an efficient static scheduling of block computations, developed especially for a parallel direct sparse linear factorization based on a combination of 1D and 2D block distributions. Our solver uses a supernodal Fan-In approach and is fully driven by our static scheduling algorithm. Compared to the existing parallel direct solvers our solver shows very favorable performance results.

**Keywords :** high performance computation, communication/computation overlap, pipelined algorithm, optimal packet size, Cholesky factorization, parallel direct solver, sparse matrix.

**Discipline :** Computer Science

---

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



# Table des matières

<b>Remerciements</b>	<b>i</b>
<b>Introduction générale</b>	<b>1</b>
<b>I Calculs Pipelinés et Recouvrements</b>	<b>3</b>
<b>1 Introduction et problématique</b>	<b>5</b>
<b>2 Etat de l'art sur les recouvrements</b>	<b>9</b>
2.1 Possibilités de recouvrements dans les architectures et les logiciels . . . . .	9
2.2 Calcul du gain d'un pipeline . . . . .	11
2.3 Aspects relatifs à la compilation de langages data-parallèles . . . . .	12
2.4 Technique de macro-pipeline et équilibrage de charge dynamique . . . . .	14
2.5 Autres approches . . . . .	16
2.6 Applications des techniques de recouvrements . . . . .	17
2.6.1 Applications des recouvrements . . . . .	17
2.6.2 Applications des macro-pipelines . . . . .	18
2.7 Conclusion et perspectives . . . . .	18
<b>3 Calcul de la taille optimale de paquets</b>	<b>21</b>
3.1 Introduction . . . . .	21
3.2 Recouvrement calcul/communication . . . . .	22
3.2.1 Formulation du problème . . . . .	22
3.2.2 Résolution du modèle théorique . . . . .	24
3.2.3 Etudes de cas pour différentes fonctions de complexités . . . . .	26
3.2.4 Comment éviter de recalculer la taille optimale des paquets ? .	30
3.3 Implémentation de la bibliothèque OPIUM . . . . .	31
3.3.1 Implémentation de l'algorithme générique . . . . .	31
3.3.2 La Bibliothèque OPIUM . . . . .	31
3.4 Résultats expérimentaux . . . . .	34

3.4.1	Factorisation LU par colonnes et par blocs colonnes . . . . .	34
3.4.2	Algorithme du SOR . . . . .	38
3.5	Conclusion . . . . .	41
<b>4</b>	<b>Vers un calcul adaptatif de la taille optimale</b>	<b>43</b>
4.1	Introduction et positionnement du problème . . . . .	43
4.2	Calcul de la suite optimale . . . . .	44
4.3	Validations expérimentales . . . . .	48
4.4	Conclusion et perspectives . . . . .	50
<b>Bibliographie</b>		<b>51</b>
<b>II</b>	<b>Solveurs Parallèles Directs pour Matrices Symétriques Définies Positives Creuses</b>	<b>59</b>
<b>5</b>	<b>Introduction et positionnement de l'étude</b>	<b>61</b>
<b>6</b>	<b>Solveur <math>LDL^t</math> avec distribution 1D</b>	<b>69</b>
6.1	Introduction . . . . .	69
6.2	Factorisation parallèle et algorithme de distribution . . . . .	71
6.2.1	Etape de repartitionnement de la partition initiale . . . . .	76
6.2.2	Etape de distribution . . . . .	77
6.3	Expérimentations numériques . . . . .	80
6.3.1	L'algorithme de repartitionnement et de distribution . . . . .	81
6.3.2	La factorisation parallèle . . . . .	82
6.3.3	La résolution du système . . . . .	86
6.4	Conclusion . . . . .	88
<b>7</b>	<b>Solveur <math>LDL^t</math> avec distribution 2D</b>	<b>89</b>
7.1	Introduction . . . . .	89
7.2	Factorisation parallèle et algorithme de distribution . . . . .	89
7.3	Expérimentations numériques . . . . .	92
7.3.1	Performances de la factorisation parallèle . . . . .	92
7.3.2	Comparaisons avec PSPASES . . . . .	95
7.4	Étude de la scalabilité mémoire . . . . .	97
7.5	Application au cas d'une matrice pleine . . . . .	98
7.6	Conclusion et Perspectives . . . . .	100
<b>Bibliographie</b>		<b>101</b>

<b>Conclusion générale et perspectives</b>	<b>109</b>
<b>Annexe</b>	<b>111</b>
<b>A Fonctions de complexité</b>	<b>113</b>
A.1 Algorithme de Cholesky-CROUT en dense . . . . .	113
A.2 Algorithme de Cholesky-CROUT en creux . . . . .	114
<b>B Intégration dans une chaîne logicielle</b>	<b>117</b>
B.1 Modélisation expérimentale des BLAS . . . . .	119
B.2 Modélisation expérimentale des routines de communications . . . . .	119
<b>C Liste des publications de l'auteur</b>	<b>123</b>



# Liste des tableaux

6.1	Description de nos cas tests. . . . .	81
6.2	Temps d'exécution des étapes de partitionnement/distribution et factorisation symbolique en 1D. . . . .	82
6.3	Temps d'exécution de la factorisation parallèle en 1D. . . . .	83
6.4	Temps d'exécution de la résolution parallèle en 1D. . . . .	87
7.1	Temps d'exécution de la factorisation parallèle en 2D. . . . .	93
7.2	Influence du seuil de distribution sur le temps de factorisation. . . . .	94
7.3	Description des cas tests de comparaison. . . . .	96
7.4	Comparaison des performances entre PASTIX et PSPASES. . . . .	96



# Liste des figures

1.1	Boucle DOACROSS séquentielle . . . . .	5
1.2	Boucle DOACROSS parallèle avec distribution par blocs. . . . .	5
1.3	Diagramme de Gantt d'un programme pipeline entre deux processeurs.	7
1.4	Programme pipeline entre plusieurs processeurs. . . . .	7
1.5	ADI séquentiel. . . . .	7
1.6	ADI pipeline. . . . .	7
3.1	Schéma pipeline entre 2 processeurs. . . . .	23
3.2	Prise en compte de la contrainte. . . . .	25
3.3	Factorisation <i>LU</i> par colonnes. . . . .	27
3.4	Ecart maximal n'engendrant pas de re-calcul. . . . .	30
3.5	Algorithme principal utilisé dans la bibliothèque OPIUM. . . . .	31
3.6	Temps de factorisation en fonction du nombre de processeurs. . . .	35
3.7	Rapport entre les temps de factorisation avec et sans recouvrement.	35
3.8	Temps de factorisation en fonction de la taille des paquets. . . . .	36
3.9	Temps d'exécution de chaque étape de la factorisation. . . . .	36
3.10	Diagrammes Paragraph de Gantt. . . . .	36
3.11	Temps d'exécution de chaque étape de la factorisation. . . . .	37
3.12	Temps de factorisation en fonction du nombre de processeurs. . . .	37
3.13	Temps de factorisation par bloc en fonction de la taille des paquets.	37
3.14	Temps de factorisation par bloc en fonction du nombre de processeurs.	37
3.15	SOR sans pipeline. . . . .	39
3.16	SOR avec pipeline. . . . .	39
3.17	Détail sur une frontière. . . . .	39
3.18	Temps du SOR en fonction de la taille de la matrice. . . . .	40
3.19	Temps du SOR en fonction du nombre de processeurs. . . . .	40
3.20	Temps du SOR en fonction de la taille des paquets. . . . .	40
4.1	Factorisation de Cholesky. . . . .	45
4.2	Pipeline dans un cas irrégulier. . . . .	46
4.3	Suite théorique de taille de paquets. . . . .	48
4.4	Recouvrement sans BLAS 1. . . . .	49

4.5	Extensibilité sans BLAS 1 . . . . .	49
4.6	Recouvrement avec BLAS 1 . . . . .	49
4.7	Extensibilité avec BLAS 1 . . . . .	49
5.1	Une matrice factorisée structurée par bloc et son arbre d'élimination . . . . .	63
5.2	Schéma Fan-Out . . . . .	66
5.3	Schéma Fan-In . . . . .	67
5.4	Schéma Multifrontal . . . . .	67
6.1	Sans Halo . . . . .	70
6.2	Avec Halo . . . . .	70
6.3	Un exemple de matrice factorisée structurée par bloc . . . . .	72
6.4	Algorithme de la factorisation parallèle pour une distribution 1D . . . . .	73
6.5	Modifications apportées à l'algorithme de factorisation 1D . . . . .	75
6.6	Repartitionnement des nœuds de l'arbre d'élimination par bloc . . . . .	76
6.7	Ajout des contributions, affectation d'un processeur . . . . .	79
6.8	Efficacité relative pour une distribution 1D . . . . .	84
6.9	CUBE39 : diagramme de Gantt et de communications . . . . .	85
6.10	BMW3 : diagramme de Gantt et de communications . . . . .	85
6.11	CRANKSEG1 : diagramme de Gantt et de communications . . . . .	86
7.1	Algorithme de la factorisation parallèle pour une distribution 2D . . . . .	91
7.2	Diagramme de Gantt pour BMWCRA1 sur 64 processeurs . . . . .	95
7.3	GRID511 : surcoût mémoire pour une distribution 1D . . . . .	98
7.4	GRID511 : surcoût mémoire pour une distribution 2D . . . . .	98
7.5	OILPAN : surcoût mémoire pour une distribution 1D . . . . .	98
7.6	OILPAN : surcoût mémoire pour une distribution 2D . . . . .	98
7.7	Diagramme de Gantt pour une matrice pleine en 1D . . . . .	99
7.8	Diagramme de Gantt pour une matrice pleine en 2D . . . . .	99
B.1	Enchaînement des différentes fonctionnalités . . . . .	118
B.2	SP2 LaBRI sans interruptions . . . . .	120
B.3	SP2 LaBRI avec interruptions . . . . .	120
B.4	SP2 CINES sans interruptions . . . . .	121
B.5	SP2 CINES avec interruptions . . . . .	121

# Introduction générale

L'utilisation de calculateurs parallèles hautes performances, en particulier pour résoudre des problèmes nécessitant une grande puissance de calcul, est incontournable. On considère d'autre part qu'environ 90% des problèmes scientifiques peuvent se ramener à des noyaux d'algèbre linéaire. On comprend donc l'intérêt de disposer de bibliothèques réalisant ces opérations génériques, de façon portable, avec une grande efficacité, réduisant ainsi considérablement le coût de développement pour l'utilisateur final. C'est le cas par exemple, pour l'algèbre linéaire dense, de la bibliothèque ScaLAPACK basée elle-même sur les routines BLAS, encapsulant ainsi l'efficacité et la portabilité sur les différentes architectures de processeurs. D'autres bibliothèques existent dans le domaine de l'algèbre linéaire creuse ; elles sont pour la plupart en cours de développement.

Si l'on considère maintenant les machines parallèles à mémoire distribuée, une des principales difficultés rencontrées est d'obtenir le niveau de parallélisme et la granularité optimale pour réduire au maximum le surcoût des communications induites par les échanges de données non locales. Ces optimisations peuvent être prises en compte, soit de façon explicite en essayant de recouvrir le temps des communications en adaptant la granularité du calcul, soit de façon algorithmique par exemple en proposant un ordonnancement statique des calculs qui tienne compte des échanges de données distantes. Dans les deux cas, il convient d'utiliser efficacement les primitives de communications asynchrones.

Nos travaux de recherche traitent des problèmes du calcul hautes performances et plus spécifiquement du calcul parallèle scientifique pour des applications irrégulières en vraie grandeur. Cette thèse est constituée de deux parties ; pour chacune on trouvera une introduction présentant le contexte et la problématique considérée. On présente également une bibliographie séparée pour chaque partie.

La première partie de cette thèse concerne la problématique du recouvrement calcul/communication que l'on présente dans le chapitre 1.

Au chapitre 2, nous passons en revue les principales approches qui ont été étudiées, des aspects compilation aux techniques des macro-pipelines, en passant par

les techniques de “prefetch” pour les machines à mémoire partagée.

Dans le chapitre 3 nous présentons une contribution aux optimisations du recouvrement calcul/communication sur des architectures parallèles à mémoire distribuée, avec en particulier le calcul du grain optimal et de la taille optimale des paquets à communiquer. Ces travaux ont débouché sur le développement d'une bibliothèque portable intégrant ces mécanismes de découpage des messages.

La suite logique de ces travaux a été d'étudier l'extension des mécanismes proposés dans le cadre régulier aux problèmes irréguliers. Nous nous sommes intéressés, dans le chapitre 4 au calcul de la granularité maximisant le recouvrement calcul/communication pour l'algorithme de factorisation de Cholesky pour des matrices pleines en exploitant l'irrégularité due à la symétrie de la matrice.

Dans la seconde partie de cette thèse, nous nous sommes intéressés à des techniques algorithmiques de distribution de données et d'ordonnancement des calculs conduisant au masquage quasi-total des communications pour le problème de la résolution parallèle directe de grands systèmes linéaires creux. Une introduction aux méthodes de résolution parallèle directe de grands systèmes linéaires creux est tout d'abord donnée au chapitre 5. Un des objectifs de cette étude est d'intégrer ces techniques et de les valider dans un cadre industriel ; pour cela nous avons utilisé un jeu de 33 problèmes tests, en grande partie issus d'applications industrielles<sup>1</sup>.

La mise en œuvre de ces travaux nous a conduit à implémenter un solveur direct parallèle pour une factorisation de Cholesky par blocs, intégrant l'approche Fan-In. Dans le chapitre 6 nous décrivons l'algorithme proposé pour une distribution 1D, ainsi qu'une analyse des performances obtenues sur un IBM SP2.

Le chapitre 7 présente une extension à des distributions mixtes 1D/2D offrant une meilleur scalabilité. Les performances obtenues se comparent très favorablement aux meilleurs solveurs parallèles directs actuels.

Le dernier chapitre présente un bilan des travaux effectués et les perspectives de recherche à moyen et à long terme.

---

<sup>1</sup>Ce travail est supporté par le *Commissariat à l'Energie Atomique* CEA/CESTA sous le contrat No. 7V1555AC, et par le GDR ARP (groupe iHPerf) du CNRS

# Première partie

## Calculs Pipelinés et Recouvrements



# Chapitre 1

## Introduction et problématique

L'utilisation de machines parallèles est maintenant entrée dans la plupart des domaines applicatifs. Cependant, l'utilisation de machines parallèles à mémoire distribuée induit un surcoût dû aux communications. Ce coût doit être réduit à son minimum si l'on veut garantir l'obtention de bonnes performances et surtout si l'on souhaite conserver une bonne extensibilité d'une application lorsque le nombre de processeurs augmente. Il va de soi que la première étape pour la réduction des communications lors de la parallélisation d'une application sur une machine à mémoire distribuée est le choix de bonnes distributions pour les données. Ces distributions sont "bonnes" si elles réduisent les communications tout en conservant un bon parallélisme. Il faut ensuite être en mesure de recouvrir un maximum de communications par des calculs.

Par contre, du fait de certaines dépendances dans le code, il n'est pas toujours facile (voire impossible) de paralléliser un code. Un bon exemple est la compilation des boucles **DOACROSS** dans les langages data-parallèles. Les dépendances entre les instructions ou à l'intérieur d'une même instruction peuvent empêcher l'exécution en parallèle d'un code. Prenons l'exemple de la Figure 1.1. La dépendance entre les itérations au sein de l'instruction  $S_1$  donnera une exécution séquentielle de la boucle.

```
do i=0, time_step
  receive(left, a(local_b-1)
  do j=local_b, local_u
    a(i) = 0.5*(a(i-1) + a(i+1))
  end do
end do
S1
  a(i) = 0.5*(a(i-1) + a(i+1))
  end do
end do
```

FIG. 1.1: Boucle DOACROSS séquentielle.

FIG. 1.2: Boucle DOACROSS parallèle avec distribution par blocs.

Si la distribution choisie pour le vecteur  $\mathbf{a}$  est **BLOCK**, la boucle sera parfaitement séquentielle. Une solution consiste à choisir une distribution de type **CYCLIC(K)** où  $K$  est choisi en fonction de la machine cible. Par contre, celle-ci entraîne la génération de nombreuses communications, ce qui peut conduire à une augmentation du temps total d'exécution.

Parfois, il se peut que la distribution choisie pour une matrice dépende de calculs précédents et ne puisse être changée. On peut alors faire appel à des optimisations de type macro-pipeline pour briser cette séquentialité. L'utilisation d'un macro-pipeline consiste à réduire le grain de calcul afin de communiquer plus tôt les données nécessaires au démarrage des calculs sur le processeur suivant. On peut bien sûr envoyer chaque donnée dès qu'elle a été calculée ce qui a pour effet de faire démarrer le processeur suivant au plus tôt (et ainsi de suite). Par contre, le coût de communication des machines actuelles induit un temps d'initialisation d'une communication qui est indépendant du nombre de données envoyées. Ce coût est généralement très élevé et il faut donc réduire le nombre de communications en envoyant plusieurs données à la fois (ce qui retarde le démarrage du processeur suivant!). Le grain de calcul (et de communication) est donc un paramètre très important qu'il convient de calculer avec soin. Nous présentons diverses méthodes de calcul pour des cas de figure différents dans les chapitres 3 et 4. L'exécution d'un macro-pipeline donnera un diagramme de Gantt comme celui présenté dans la Figure 1.3. On notera sur ce schéma qu'il y a en fait deux types de recouvrements : un recouvrement des communications sur un processeur, et un recouvrement des calculs entre deux processeurs. Si l'on utilise un tel schéma sur plusieurs processeurs, on aura un gain supérieur puisque les calculs seront recouverts entre tous les processeurs (voir Figure 1.4). Le calcul de tels gains est présenté dans la section 2.2.

Une application de telles optimisations est illustrée dans le code ADI (Alternative Direction Implicit) de la Figure 1.5. Cette application est citée dans de nombreux articles comme application de référence [49, 56, 58, 73, 74]. Ce problème, divisé en deux phases de calculs, nécessite une modification du code si l'on veut réduire le nombre des communications. Les solutions sont, soit de redistribuer les données après la première phase de calcul et ainsi éliminer les communications pour la seconde phase, ou pipeliner les communications dans la seconde phase. La version pipeline de l'ADI est donnée dans la Figure 1.6 (la redistribution est une simple transposition).

Le chapitre suivant présente un état de l'art sur les recouvrements calculs/communications et les techniques de macro-pipelines. Nous avons essayé de présenter ces techniques avec une division en sous-thèmes.

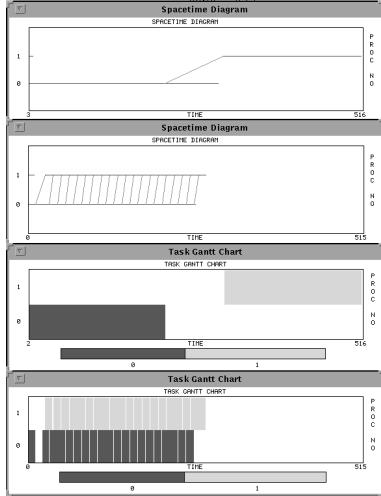


FIG. 1.3: Diagramme de Gantt d'un programme pipeline entre deux processeurs.

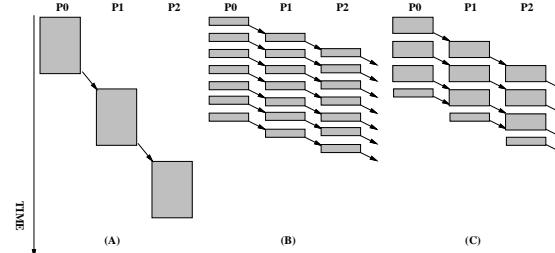


FIG. 1.4: Programme pipeline entre plusieurs processeurs.

```

do iter=1, numiter
  /* boucle sur les lignes */
  do i=0, N-1
    do j=0, N-1
      x(i,j) = f(x(i,j),x(i,j-1),a(i),b(i))
    end do
  end do

  /* boucle sur les colonnes */
  do i=0, N-1
    do j=0, N-1
      x(i,j) = f(x(i,j),x(i-1,j),a(i),b(i))
    end do
  end do
end do
  
```

FIG. 1.5: ADI séquentiel.

```

do iter=1, numiter
  /* boucle sur les lignes */
  do i=start, end
    do j=0, N-1
      x(i,j) = f(x(i,j),x(i,j-1),a(i),b(i))
    end do
  end do

  /* boucle sur les colonnes (pipeline) */
  do jj=0, N-1, NB
    if (my_id != 0)
      recv x(start-1,jj : jj+NB) de my_id-1
    do i=start, end
      do j=jj, jj+NB-1
        x(i,j) = f(x(i,j),x(i-1,j),a(i),b(i))
      end do
    end do
    if (my_id != p-1)
      send x(end,jj : jj+NB) à my_id+1
    end do
  end do
end do
  
```

FIG. 1.6: ADI pipeline.

Dans une première section, nous présentons les diverses études sur les possibilités de recouvrements dans les architectures et dans les logiciels. Notre but étant de faire un tour d'horizon des techniques macro-pipelines, nous présentons ensuite les études des gains que peuvent apporter ces techniques dans un algorithme parallèle. Ces optimisations ont été largement utilisées dans la compilation de langages data-parallèles comme HPF afin de réduire le coût des communications et nous présentons donc également un tour d'horizon des compilateurs qui les utilisent. Puis nous présentons les articles donnant des techniques de macro-pipelines (modélisation, calculs de grain) ainsi que ceux abordant le problème de l'équilibrage des charges dynamique dans de tels algorithmes. Nous présentons également d'autres approches, comme celles utilisées pour le pipeline des communications ou des entrées/sorties, ou les techniques de *prefetch* dans les architectures à mémoire partagée. Enfin, avant une conclusion et une présentation des perspectives de ce type de travaux, nous donnons des applications des techniques de recouvrements simples et de recouvrements macro-pipelines.

# Chapitre 2

## Etat de l'art sur les recouvrements calculs/communications

### 2.1 Possibilités de recouvrements dans les architectures et les logiciels

Si l'on souhaite recouvrir calculs et communications dans un algorithme parallèle, il faut savoir si l'architecture cible est capable de tels recouvrements et si le logiciel qui l'utilise permet de tirer partie des capacités de l'architecture.

Au point de vue architectural, si les recouvrements sont complets, c'est qu'il existe des dispositifs permettant d'envoyer les données sur les canaux de communications sans interrompre le processeur de calcul. Cela est réalisé généralement grâce à des DMA<sup>1</sup> ou à des co-processeurs de communication. Il faut aussi que les accès aux données pour les calculs ou les communications puissent se faire de manière concurrente, sans partage de bus mémoire par exemple. Dans les machines à mémoire partagée, on peut également avoir des contrôleurs qui effectueront les accès aux bancs distants de manière asynchrone [11, 23]. Le livre de Culler et al. [23] présente de manière précise les aspects architecturaux des recouvrements calculs/communications, et ceci pour les machines à mémoire distribuée et pour les machines à mémoire partagée.

Au point de vue du logiciel, il faut que la couche de communication ou le support d'exécution permette les recouvrements. On peut distinguer trois modèles de programmation utilisables pour obtenir des recouvrements calculs/communications suivant que l'on a une architecture à mémoire distribuée ou à mémoire partagée. On peut utiliser une bibliothèque de communication avec des appels non-bloquants (comme `MPI_Isend` ou `MPI_Irecv` dans l'interface MPI). Avec ce type d'appel, le

---

<sup>1</sup>Direct Memory Access.

programmeur “reprend la main” immédiatement après l’appel de la routine. Pour peu qu’il n’ait pas à utiliser le tampon de communication, il peut effectuer d’autres tâches pendant que la communication s’exécute en arrière plan. Lorsque le programmeur voudra (ré-)utiliser le tampon de communication, il ne pourra le faire qu’après l’appel d’une routine d’attente/test (`MPI_Wait/MPI_Test` en MPI). Dans une architecture à mémoire partagée, on pourra effectuer des opérations de *prefetch* asynchrone grâce auxquelles on pourra récupérer les données nécessaires à des calculs futurs pendant l’exécution d’autres opérations. Nous parlerons de ce type d’opérations dans une autre section (2.5). Enfin, pour ces deux types de machines, on peut mettre les communications dans des processus légers (*threads*) qui permettront de minimiser les latences de communications en s’exécutant de manière asynchrone avec les calculs. En effet, un changement de contexte sera effectué dès qu’une communication (ou une entrée/sortie en général) sera en attente.

Pour ce qui est des bibliothèques de communications, de nombreux articles étudient les capacités de recouvrements de diverses interfaces de communication et de diverses architectures. Dans [69], les auteurs évaluent les capacités de deux machines (EM-X et IBM SP-2) à recouvrir les calculs et les communications. Ces résultats sont étendus dans [68], avec l’étude des capacités de la SGI PowerCHALLENGE et de l’IBM SP-2. Leurs conclusions sont que le SP-2 permet de meilleurs recouvrements des communications que la SGI grâce à l’utilisation d’un co-processeur de communication. Les applications tests sont décrites dans la section 2.6.2. Certaines bibliothèques de communications ont été conçues pour tirer parti des recouvrements calculs communications. C’est le cas de MPI, qui est présenté par la suite, et de la bibliothèque décrite dans [8]. Il s’agit de découper chaque appel de communication en trois composantes : initialisation, exécution et complétion. L’exécution peut alors se faire de manière totalement asynchrone. L’intérêt de cette bibliothèque est que ce principe a été également appliqué aux communications collectives, ce qui permet par exemple d’obtenir un all-to-all non-bloquant.

L’interface MPI fournit la possibilité d’effectuer des émissions ou des réceptions non-bloquantes. Par contre, elle ne requiert pas que ces communications soient effectuées de manière asynchrone.

Le réseau Myrinet est l’un des réseaux les plus rapides pour connecter des grappes de PC. Les possibilités de recouvrements de ce type de réseau avec l’interface BIP ou MPI sont évalués dans [19]. Myrinet possède une interface réseau (le LANAI) sur laquelle on peut exécuter un code de gestion des communications. Malgré tout, les implémentations de MPI classiques n’utilisent pas cette fonctionnalité. Par contre, BIP et sa version MPI l’utilise de manière efficace.

Dans [40], les auteurs constatent que dans certaines implémentations de MPI (IBM SP-2 et SGI Origin2K), les recouvrements sont quasiment inexistant malgré le

fait que les communications soient non-bloquantes. Ces résultats sont en contradiction avec les ceux présentés auparavant [68]. De notre point de vue, nous présentons dans l'annexe B.2 les résultats de nos expérimentations qui montrent que suivant la taille des messages il est possible ou non d'obtenir des recouvrements partiels. Sur le CRAY T3E, les résultats sont surprenants puisque les tests avec recouvrements ont des performances supérieures aux prédictions. La SP-2 d'IBM est encore étudiée dans [17] pour l'implémentation sur cette machine des messages actifs [35]. Ceux-ci permettent de réduire la latence des communications grâce à des appels proches des RPC. Cette bibliothèque possède également des routines asynchrones (`am_store_async`) permettant les recouvrements. Des routines non-bloquantes sont utilisées pour éviter les blocages.

Il est également possible de mixer *threads* et communications pipelines pour améliorer les performances des applications. C'est ce qui est décrit par exemple dans [70] où les auteurs optimisent un code de gradient conjugué sur une grappe de SMP. Il faut par contre que la bibliothèque de communication soit *thread-safe*. Pour cette application, un autre problème réside dans le fait que les communications doivent être hybrides (par mémoire partagée à l'intérieur d'une grappe et par passage de message "classique" entre les grappes).

## 2.2 Calcul du gain d'un pipeline

La première question à se poser lorsque l'on veut optimiser un programme parallèle à l'aide de techniques de recouvrements calculs/communications ou de macro-pipeline est *Quel gain puis-je espérer obtenir grâce à cette technique ?* En effet, suivant les paramètres de la machine cible, les caractéristiques et les dépendances de l'algorithme, il ne sera pas forcément intéressant de vouloir à tout prix utiliser ces optimisations.

Une étude récente de Quinn et Hatcher [61] donne une évaluation des gains des recouvrements calculs/communications. Grâce à une modélisation du comportement de la machine cible en terme de communication, de bufferisation et de calculs, Quinn et Hatcher calculent le gain que l'on peut espérer obtenir sur un processeur en utilisant diverses techniques de recouvrements. Les résultats prouvent qu'un gain de 2 au maximum peut être obtenu dans les meilleurs des cas et que bien souvent ce gain est inférieur. Ces résultats ont été étendus dans [33] en prouvant que des gains supérieurs à deux peuvent être obtenus par des techniques de macro-pipelines si l'on tient compte de l'ensemble de la machine et non pas uniquement des gains sur chaque processeur. En effet, l'intérêt de telles techniques réside surtout dans le fait que l'on brise la séquentialité du programme en pipelinant les messages (et

donc les calculs). Sur  $P$  processeurs, on peut ainsi obtenir un gain de  $P - 1$  grâce à la fois aux recouvrements des calculs et des communications, mais aussi grâce aux recouvrements des calculs entre eux sur différents processeurs.

Par ailleurs, des tests de techniques macro-pipelines à une dimension ont été effectués dans le cadre de l'optimisation des communications dans les compilateurs paralléliseurs [10, 12, 38, 39, 58, 71], pour des outils de parallélisation [36], ... Nous décrivons quelques tests sur de “vraies” applications dans les sections 2.6.1 et 2.6.2.

On trouve aussi dans [48] une modélisation précise de l'accélération qui peut être atteinte en utilisant à la fois des *threads* et des *prefetch* dans une machine à mémoire partagée. Il s'agit toutefois de programmes avec des accès aux données simples.

## 2.3 Aspects relatifs à la compilation de langages data-parallèles

Les langages *data-parallèles* tels que High Performance Fortran, Fortran D, Vienna Fortran ont été présentés au début des années 90 comme la future boite noire permettant de paralléliser simplement et pratiquement automatiquement la plupart des applications numériques, et ceci avec d'excellentes performances en reportant dans le compilateur tous les problèmes d'optimisations habituellement laissés à l'utilisateur avec le passage de messages. Le placement des données (et donc des calculs grâce à la “*Owner Computes Rule*”) est toutefois laissé à la responsabilité du programmeur d'applications par le biais de directives (des commentaires Fortran) insérées dans le code et prises en compte (ou pas) par le compilateur. Ces langages étant destinés dans un premier temps à la parallélisation d'applications sur machines à mémoire distribuée, une des optimisations classiques de ce type de compilateur consiste donc en la réduction du coût des communications grâce aux recouvrements ou aux pipelines. Dans cette section, nous ne nous intéressons qu'aux aspects directement liés à la compilation de tels langages. Le problème du grain de calcul est quand à lui présenté dans la section 2.4, et les applications dans la section 2.6.2.

Un compilateur paralléliseur a trois tâches à effectuer pour optimiser un code avec des recouvrements de type pipeline : il doit tout d'abord découvrir un schéma pipeline dans les nids de boucles (parfois à l'aide de transformations de boucles), puis calculer le grain de calcul et de communication en fonction des calculs effectués dans le nid, de l'espace d'itération et des paramètres de la machine cible, et enfin générer le code.

La première tâche d'un compilateur paralléliseur est donc de découvrir les schémas pipelines dans les boucles. Dans [71], on trouve un algorithme pour découvrir

les boucles *cross-processor*, c'est à dire les boucles dans lesquelles les itérations traversent les frontières des processeurs. Il faut également que le nid soit suffisamment profond pour que le grain de calcul soit intéressant. Il peut être nécessaire d'appliquer une permutation de boucle pour pouvoir exploiter un pipeline. Un algorithme plus général est donné dans [38] dans lequel une estimation du coût de la boucle est également donnée.

La seconde tâche du compilateur est de déterminer le grain de calcul et de communication. Ceci peut être fait de manière statique ou dynamique. Ces techniques sont décrites dans la section 2.4.

La génération du code pour un schéma macro-pipeline n'est pas aisée. Deux approches peuvent être utilisées : directement en générant le code du macro-pipeline, ou en faisant appel à une bibliothèque.

La première approche a été utilisée dans le cadre des compilateurs FortranD [71] et PARADIGM [58], ainsi que dans le cadre du développement d'un outil de parallélisation automatique de codes Fortran, CAPTools [36]. Les pipelines utilisés ici sont mono-dimensionnels.

Pour ce qui est de l'utilisation d'une bibliothèque spécialisée, une intégration de la bibliothèque LOCCS dans le compilateur HPF ADAPTOR a été effectuée [12]. L'utilisation de techniques macro-pipelines est toutefois laissée à la charge du programmeur, mais la gestion des communications, des bufferisations et des distributions HPF est faite par le compilateur. Cette étude pourrait être améliorée en utilisant les techniques de recherche de boucles "pipelinables" présentées dans [38] et [71].

Toutes ces techniques sont proches de celle du *tiling* qui consiste à transformer un nid de boucles en modifiant le grain des itérations soit pour améliorer l'utilisation des mémoires hiérarchiques, soit pour augmenter le parallélisme du code [41, 62, 77]. Dans la technique du tiling, deux paramètres sont étudiés : la forme des tuiles et leur taille (ou volume). Dans les problèmes auxquels nous nous intéressons, nous ne considérons pas la forme des tuiles car les dépendances entre les itérations sont parallèles ou orthogonales aux espaces d'itération, et nous ne divisons qu'une seule des dimensions. Par contre, les problèmes liés au calcul de la taille des tuiles nous concernent directement. Parmi les travaux les plus récents sur ce sujet, on peut citer les travaux effectués à Rennes et à Valenciennes autour du tiling orthogonal multi-dimensionnel [4–6]. Ils sont décrits plus précisément dans la section 2.4.

## 2.4 Technique de macro-pipeline et équilibrage de charge dynamique

Les pipelines ont été modélisés à plusieurs reprises. Dans [43–45], ils sont modélisés à l'aide de réseaux de Pétri modifiés afin d'évaluer les performances et de calculer le grain. Les articles cités dans la suite de ce chapitre utilisent des modélisations plus classiques avec un calcul de complexité du code pipeline, puis un calcul du grain afin de minimiser le temps total d'exécution comme par exemple dans [72] pour le cas d'un pipeline général.

Le grain d'un macro-pipeline est le paramètre qui conditionne l'obtention de performances. Trop petit, il occasionnera le démarrage de trop nombreuses communications ce qui entraînera des pertes de performances dues à leur surcoût ; trop gros, il réduira le parallélisme. De nombreux articles ont abordé ce problème de manière statique ou dynamique que ce soit dans le cadre de parallélisation d'applications par passage de message ou de la compilation de langages data-parallèles.

Le calcul statique du grain d'un pipeline nécessite une bonne connaissance de l'architecture cible et de l'algorithme parallélisé et donc leur modélisation précise. Un certain nombre d'articles utilisent cette technique [25, 53]. En particulier, cette option est souvent choisie pour l'optimisation de compilateurs paralléliseurs ; c'est le cas de PARADIGM [38, 56–58] ou FortranD [71]. La principale difficulté est l'évaluation du coût des calculs et des communications. Pour ces deux compilateurs, les macro-pipelines sont mono-dimensionnels et le calcul du grain n'est donné que pour des cas triviaux et avec une modélisation sommaire. Malgré tout, on trouve dans [58] une modélisation de l'enchaînement de deux pipelines. Le calcul de la taille des tuiles dans la technique du *tiling* est généralement statique. Dans [4–6], on trouve le calcul de taille de tuiles pour le tiling orthogonal à respectivement deux, trois et  $n$  dimensions. La solution pour trouver le grain du tiling revient à résoudre un problème d'optimisation non-linéaire de programmation entière. La taille et la forme des tuiles sont des variables et la fonction objectif est la minimisation du temps total d'exécution. Un algorithme en  $O(n \log n)$  est donné dans le cas d'un espace à  $n$  dimensions. Les programmes sont modélisés à l'aide d'équations récurrentes. Des résultats concernant le tiling à deux dimensions sont donnés dans [14]. Une modélisation précise de techniques macro-pipelines est donnée également dans [24] pour la parallélisation d'algorithmes numériques sur un hypercube. Les possibilités de communications  $n$ -ports sont prises en compte dans le calcul du grain.

Le calcul du grain de manière dynamique est avantageux à plus d'un titre car il permet dans un premier temps d'éviter une modélisation trop fastidieuse de l'architecture cible. En effet, l'apparition de processeurs aux caractéristiques de plus

en plus compliquées (pipelines super-scalaires, unités arithmétiques parallèles, mémoires caches à plusieurs niveaux, ...) empêche une modélisation d'une précision suffisante. Dans un second temps, il permet de s'adapter dynamiquement à la variation de charge des machines et des réseaux. Enfin, si le volume de calcul et de communications varie suivant les données, comme dans le cas de la parallélisation d'algorithmes mettant en œuvre des matrices creuses, cette évaluation doit souvent être effectuée à l'exécution.

Dans [50] puis dans [49], une optimisation du calcul du grain est effectuée à l'exécution qui consiste à prendre des mesures de performances sur les deux premières itérations du pipeline afin d'en déduire le meilleur grain. Deux itérations sont testées pour tenir compte des effets de cache. Ensuite, une taille initiale de bloc est choisie. Une recherche des blocs causant des temps d'attente est effectuée et les blocs trouvés sont redécoupés de manière récursive. Enfin, les messages excédents sont supprimés en ré-exécutant l'algorithme sur les blocs adjacents qui ne sont pas redécoupés. L'ordonnancement des calculs et des messages est calculé par un nœud spécifique. La distribution des données n'est également effectuée qu'après la troisième itération.

Le même type d'optimisation est présenté dans [66, 67] pour la parallélisation de boucles DOACROSS. L'équilibrage dynamique est réalisé par un schéma de type maître-esclave où le grain évolue en fonction de la charge des machines sur lesquelles le pipeline est exécuté. Ici la charge est constamment régulée au cours du calcul et un des problèmes est donc d'éviter l'effet *ping-pong* entre les différents processeurs. Une modélisation assez précise des gains provenant de l'équilibrage est donnée.

Un équilibrage dynamique utilisé dans un algorithme d'ordonnancement de tuiles est présenté dans [54]. La distribution est fixée au départ de l'algorithme et elle évolue au cours de son exécution en fonction de la charge des processeurs. Des migrations sont effectuées avec les voisins et il s'agit donc d'un algorithme décentralisé qui rend l'approche plus "scalable".

Dans [31], on donne une méthode de calcul du grain d'un macro-pipeline dans le cas général, c'est-à-dire quand les calculs avant ou après les communications sont des fonctions quelconques de la taille des données et du nombre de processeurs. Ces méthodes sont implémentées dans un bibliothèque, OPIUM<sup>2</sup>, pour laquelle l'utilisateur ne doit fournir que les paramètres de la machine cible et les fonctions de complexité de ces calculs. Ces résultats ont été ensuite étendus dans le cas où la taille de paquets doit évoluer au cours du temps en fonction de la variation des calculs (connue à priori). On calcule alors une suite de tailles [3, 63].

Les pipelines peuvent ensuite être implémentés directement en les écrivant avec des communications non-bloquantes ou en faisant appel à une bibliothèque. C'est

---

<sup>2</sup>Optimal Packet sIze Optimization Methods.

cette dernière option qui est choisie dans [25, 26, 31, 32]. La bibliothèque LOCCS<sup>3</sup> permet d'effectuer des schémas macro-pipelines avec plusieurs types de schémas de communication (communications simples entre deux processeurs, *shifts*, diffusions, etc.). La gestion des boucles et des tampons de communication est entièrement gérée à l'intérieur de la bibliothèque et l'utilisateur ne doit fournir que des routines de calcul sur un paquet de données et le grain du pipeline (qui lui peut être calculé à l'aide de la bibliothèque OPIUM citée précédemment). Cette dernière approche est présentée dans cette thèse aux chapitres 3 et 4.

## 2.5 Autres approches

Dans cette section, nous faisons un tour d'horizon d'autres techniques proches des recouvrements calculs/communications tels que nous les étudions. Nous présentons ici les travaux concernant les pipelines de communications et d'entrées/sorties disques, puis les techniques de *prefetch* dans les machines à mémoire partagée.

Les pipelines de communications ont été au départ utilisés dans les machines parallèles à mémoire distribuée utilisant le protocole *store-and-forward*. En effet, dans de telles architectures, un message passait de mémoire en mémoire jusqu'à atteindre la mémoire du processeur destination. Afin de masquer la distance, le message était découpé en messages de plus petites tailles qui étaient envoyés les uns à la suite des autres. Ces techniques sont toujours utilisées pour masquer les latences dans les réseaux rapides de type Myrinet ou Fast-Ethernet. Dans [60] est présentée une technique de pipeline adaptatif pour une couche de communication de bas-niveau à hautes performances pour le réseau Myrinet. Le pipeline est calculé pour chaque taille de message ce qui permet de minimiser la latence. Des travaux similaires sont également présentés dans [76]. Une présentation plus générale de ces techniques permettant la tolérance à la latence des communications sont présentées dans [23].

Dans les machines à mémoire partagée (ou virtuellement partagée), on peut également masquer les “communications” avec des techniques de *prefetch*. Ces techniques peuvent être combinées avec l'utilisation de *threads* [11, 20, 70]. Dans [48], on trouve une modélisation très précise des limites de l'utilisation de telles techniques appliquées à la machine MIT Alewife. Dans [1] sont présentées les techniques de prefetch dans les machines COMA. Pour ces machines, la mémoire est distribuée et les mémoires locales sont vues comme des caches. Il s'agit de récupérer les données en mémoire locale avant d'avoir des fautes de cache. Les pipelines peuvent être implémentés de la même manière qu'avec des machines à mémoire distribuée [9].

---

<sup>3</sup>Low Overhead Communication and Computation Subroutines.

Enfin, on peut également essayer de cacher la latence des entrées/sorties disques à l'aide de pipelines. C'est ce qui est présenté dans [22] avec comme application un algorithme de tri. La modélisation des accès aux fichiers remplace celle des communications mais les principes restent les mêmes.

## 2.6 Applications des techniques de recouvrements

Dans cette section, nous faisons un tour d'horizon des applications pouvant bénéficier de recouvrements calculs/communications et macro-pipelines.

### 2.6.1 Applications des recouvrements

Les recouvrements calculs/communications simples, c'est-à-dire entre des calculs et des communications sans dépendance peuvent être trouvés relativement facilement dans de nombreuses applications du calcul numérique. On peut citer par exemple le gradient conjugué [52], la factorisation  $LU$  [16, 28, 66], le produit de matrices [2, 16, 30, 66], la simulation de front de flamme [29], l'algorithme du simplex [1], Gauss-Seidel (avec un algorithme Red-Black) [7], et d'autres applications irrégulières [46].

Un grand nombre d'applications ont également été testées dans [48] pour montrer les performances des *prefetch* dans les machines à mémoire partagée. On peut citer par exemple Cholesky, gradient conjugué, FFT, Gauss, SOR, tri, etc. Les optimisations effectuées ne sont pas décrites et il est difficile de savoir s'il s'agit de recouvrements simples ou pipelines. Ces applications n'obtiennent pas toutes des gains importants.

Le calcul de la distribution des données peut tenir compte de la présence ou pas de recouvrements calculs/communications. Par exemple, dans [37], dans le cas d'une grappe de machines SMP, les auteurs optimisent la distribution irrégulière des données pour un algorithme de différences finies en introduisant les recouvrements calculs/communications dans le modèle d'évaluation. Dans [55], une telle optimisation est réalisée dans le cas de la parallélisation d'un code de factorisation  $LU$ .

Par ailleurs, toutes les applications des macro-pipelines présentées dans la section 2.6.2 sont également des applications potentielles des recouvrements puisqu'il est possible (et même conseillé) d'utiliser des communications asynchrones dans un schéma macro-pipelines.

## 2.6.2 Applications des macro-pipelines

L’application ADI décrite dans l’introduction est souvent référencée [9, 12, 49, 56, 58, 73, 74] car elle permet une implémentation simple d’un schéma macro-pipeline grâce à ses dépendances orthogonales. En conservant une distribution simple (par blocs de lignes), on peut implémenter un schéma pipeline 1D, et le calcul du grain est trivial.

La Transformée de Fourier Rapide est un algorithme qui se prête bien aussi à une parallélisation à l’aide de macro-pipelines et ceci malgré un schéma de communication relativement compliqué (schéma de type papillons). Dans [15], les auteurs présentent une distribution des données pour la FFT bidimensionnelle qui permet un recouvrement total des communications sur un hypercube d’Intel. D’autres articles présentent des optimisations de FFT 1D [1, 24, 68, 75] et 2D [20] avec des pipelines et des recouvrements.

Des algorithmes irréguliers peuvent également bénéficier d’optimisations macro-pipelines. C’est le cas de l’algorithme de rendu-volumique Shear-Warp [18]. Les principales difficultés sont de combiner un équilibrage de charge dynamique (réalisé grâce à un algorithme “élastique”) et un pipeline sur des types de données non-structurés (Run-Length Encoding). Dans [68, 69], un algorithme de tri bitonic est parallélisé avec une méthode macro-pipeline. Le schéma de communication est également non-trivial.

On peut également effectuer des pipelines partiels sur un graphe de tâches. C’est ce qui est étudié dans [34] avec la technique des BLAS enchaînés. Il s’agit de découper des routines BLAS de niveau 3 pour effectuer des macro-pipelines entre les appels. Le grain choisi optimise la réutilisation de la mémoire et le pipeline est conçu pour minimiser le temps total d’exécution. Un algorithme permettant de traiter toute chaîne de tels appels est donné.

D’autres applications numériques sont également étudiées comme par exemple la simulation Airshed [49], le code Hydro [49, 58, 71] adapté du noyau 23 du benchmark Livermoore, le SOR<sup>4</sup> [48, 54, 58, 66, 71], la résolution de systèmes triangulaires [58], les factorisations LU [1, 27, 31, 48] et Cholesky [48, 63], le gradient conjugué [48, 70], Gauss-Seidel [64], le produit de matrices [7, 42, 43, 45], le benchmark NAS-MG [7].

## 2.7 Conclusion et perspectives

Nous avons donc présenté un tour d’horizon assez complet des techniques de recouvrements calculs/communications et macro-pipelines sur différents types d’architectures et pour différentes applications numériques. Ces techniques sont maintenant

---

<sup>4</sup>Successive Over-Relaxation.

éprouvées et font partie des optimisations classiques des compilateurs paralléliseurs ou des supports d'exécution.

Des travaux restent cependant à effectuer dans un certain nombre de directions. Les techniques de macro-pipelines multi-dimensionnels restent encore du domaine théorique. Quelques études les évaluent, notamment celles autour des techniques de tiling mais la difficulté de leur implémentation les rend encore peu abordables de manière pratique. Les travaux autour de la minimisation des entrées/sorties sont encore sommaires, surtout si l'on veut tenir compte de l'algorithme. En ce qui concerne les architectures fortement hétérogènes, les travaux actuels sont encore assez limités. Malgré tout, ce domaine est particulièrement important à la vue de l'évolution actuelle des architectures. Une gestion dynamique des pipelines et de leur grain est indispensable si l'on veut obtenir les meilleures performances.

Dans les deux chapitres suivants, nous présentons une méthode pour calculer la taille optimale de paquets qui maximise le recouvrement calcul/communication dans les algorithmes macro-pipelines. Dans le chapitre 3, nous étudions en particulier un modèle théorique qui conduit à un schéma de calcul de la taille optimale des paquets. Le chapitre 4 propose une extension de cette étude vers un calcul adaptatif de la taille optimale de paquets pour la factorisation de Cholesky d'une matrice symétrique.



# Chapitre 3

## Calcul de la taille optimale de paquets dans le cas régulier

Dans ce chapitre, nous présentons une méthode pour calculer la taille optimale de paquets qui maximise le recouvrement calcul/communication dans les algorithmes macro-pipelines. Nous étudions, tout d'abord, un modèle théorique qui conduit à un schéma de calcul de la taille optimale des paquets. Nous présentons ensuite la bibliothèque OPIUM qui apporte, dans le cas général, des primitives efficaces de calcul de cette taille optimale des paquets. Enfin nous validons nos résultats avec deux applications régulières, la factorisation LU [21] pour une matrice dense et l'algorithme du SOR ; l'implémentation et les mesures de performances sont réalisées sur un Paragon Intel et un SP2 IBM.

### 3.1 Introduction

Nous avons vu précédemment que l'utilisation des machines parallèles à mémoire distribuée apporte un gain important en performances et en taille mémoire mais amène en contre-partie un surcoût en communications. Afin d'obtenir des programmes performants et extensibles, il convient de masquer ce surcoût. Plusieurs solutions existent. La première consiste en un choix judicieux de la distribution des données qui réduira au maximum le nombre et la taille des communications. De plus, si les dépendances le permettent, nous essaierons de découper les messages et d'initialiser les communications au plus tôt de manière asynchrone.

Dans le cas général présenté dans ce chapitre, le calcul de la taille optimale de découpage des messages n'est pas trivial ; nous serons amenés à introduire une formulation du problème ainsi qu'un schéma de calcul numérique de cette taille

optimale. Le calcul de la taille optimale nécessite de connaître, pour un macro-pipeline, la fonction de complexité des calculs ayant lieu avant et après les phases de communications. Il apparaît nécessaire de développer une bibliothèque pour calculer la taille optimale et effectuer certaines optimisations à l'exécution en particulier pour éviter de recalculer ces tailles lorsque ce n'est pas nécessaire. Cette bibliothèque, appelée OPIUM<sup>1</sup> est utilisée avec la bibliothèque LOCSS<sup>2</sup> [25, 32] qui implémente les macro-pipelines.

Notre travail consiste dans une première phase à utiliser de façon explicite cette bibliothèque en ajoutant des appels aux primitives OPIUM pour le calcul de la taille optimale ; la complexité de l'algorithme est passée comme argument. Il est à noter que nous nous plaçons pour l'instant dans le cadre des problèmes d'algèbre linéaire réguliers, et que les complexités dans les algorithmes cibles sont généralement connues ou simples à calculer car elles dépendent de la taille des données et non des données elles-mêmes. L'étape suivante consistera à inclure ces appels à l'aide d'un schéma de compilation automatique. Pour cela il sera nécessaire d'avoir une analyse du code donnant une estimation fine de la complexité de l'algorithme.

La suite du chapitre est organisée de la façon suivante. La section 3.2 introduit la formulation du problème et le modèle théorique, puis la section 3.3 présente la bibliothèque OPIUM avec les détails de son implémentation. Dans la section 3.4, nous décrivons les exemples de la factorisation *LU* et de l'algorithme du SOR, ainsi que leur implémentation sur un Paragon Intel et sur un SP2 IBM avec une analyse des performances du recouvrement mis en œuvre. Une conclusion termine sur les bénéfices de l'étude et sur les perspectives envisageables.

## 3.2 Recouvrement calcul/communication

Dans cette section, nous allons décrire un modèle théorique pour le recouvrement des communications par des calculs. Ensuite nous présenterons un schéma de calcul pour la résolution du modèle et l'illustrerons sur quelques études de cas. Nous exposerons enfin une méthode pour éviter de re-calculer des tailles optimales de paquets.

### 3.2.1 Formulation du problème

Soit  $L$  la Taille des Données à “Pipeliner” (en abrégé TDP). Les données sont découpées en  $\mu$  paquets de taille  $\nu$  (voir figure 3.1).

---

<sup>1</sup>Optimal Packet sIze compUtation Methods.

<sup>2</sup>Low Overhead Communication and Computation Subroutines.

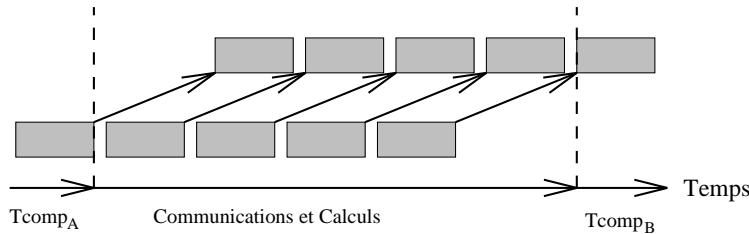


FIG. 3.1: Schéma pipeline entre 2 processeurs.

**Définition 1.** Le temps du “calcul avant” représente le temps d’exécution du calcul avant émission sur un paquet de taille  $\nu$ . Il peut être défini par

$$T_{comp_A}(\nu, L) = R_A(L) f_A(\nu) \tau_A \quad \text{où}$$

- $R_A(L)$  représente le facteur multiplicatif de la complexité du “calcul avant” et qui peut dépendre de la TDP,
- $f_A(\nu)$  représente la fonction de complexité du “calcul avant” qui dépend de son *implémentation*. En pratique nous approximerons la fonction de complexité par le terme d’ordre le plus grand au sens de son comportement asymptotique.
- $\tau_A$  représente le temps de calcul d’un élément durant le “calcul avant”.

**Définition 2.** De la même façon, le temps du “calcul après” représente le temps d’exécution du calcul après émission sur un paquet de taille  $\nu$ , avec

$$T_{comp_B}(\nu, L) = R_B(L) f_B(\nu) \tau_B.$$

**Remarque importante.** Les fonctions de complexités  $f_A(\nu)$  et  $f_B(\nu)$  peuvent éventuellement contenir un terme correspondant au coût d’une ou plusieurs communications à condition que ces communications ne soient pas intégrées dans le macro-pipeline concerné par l’étude. Ce sera par exemple le cas dans l’algorithme du SOR (voir paragraphe 3.4.2).

**Notation.** Dans la suite, nous utiliserons la notation  $\theta(f_A(\nu))/\theta(f_B(\nu))$  pour désigner un algorithme macro-pipeline ayant une complexité de “calcul avant” de l’ordre de  $\theta(f_A(\nu))$  et une complexité de “calcul après” de l’ordre de  $\theta(f_B(\nu))$ .

**Notation.**  $T_{comm}(\nu)$  est le temps d’émission d’un paquet de taille  $\nu$ . D’une façon générale, l’expression de  $T_{comm}(\nu)$  dépend du schéma de communication (diffusion, échange total, etc.), ainsi que de l’architecture du calculateur parallèle. Par la suite, nous considérerons un modèle de communication linéaire  $T_{comm}(\nu) = \beta + \nu \tau$  où  $\beta$  est le temps d’initialisation et  $\tau$  le temps de transfert unitaire. Pour prendre en compte plus précisément un schéma de type diffusion nous pourrions prendre une expression de  $T_{comm}(\nu)$  faisant intervenir le nombre de processeurs utilisés.

### 3.2.2 Résolution du modèle théorique

Le temps total d'exécution du macro-pipeline est donné par (voir figure 3.1)

$$\begin{aligned} T_{total}(\nu, L) &= T_{compA}(\nu, L) \\ &+ \frac{L}{\nu} \max \{T_{compA}(\nu, L), T_{comm}(\nu), T_{compB}(\nu, L)\} \\ &+ T_{compB}(\nu, L). \end{aligned} \quad (1)$$

Dans la suite nous noterons  $\nu_{opt}$  la Taille Optimale de Paquet (en abrégé TOP). Le calcul de la TOP consiste à déterminer  $\nu_{opt}$  qui minimise (1). La taille de paquet doit être supérieure à 1 et inférieure à la TDP. Si nous introduisons les contraintes naturelles

$$\left\{ \begin{array}{l} T_{compA}(\nu, L) < T_{comm}(\nu) \\ T_{compB}(\nu, L) < T_{comm}(\nu) \\ 1 \leq \nu \leq L \end{array} \right. \quad (2)$$

le problème (1) se réécrit sous la forme simplifiée suivante

$$T_{total}(\nu, L) = T_{compA}(\nu, L) + \frac{L}{\nu} T_{comm}(\nu) + T_{compB}(\nu, L). \quad (3)$$

D'autre part, nous pouvons faire les deux remarques suivantes. Si le temps de calcul est toujours supérieur au temps de communication, nous utiliserons le découpage minimal, soit  $\nu_{opt} = 1$ . Si nous trouvons une taille optimale de paquet supérieure à la TDP, ou si le temps de calcul est toujours inférieur au temps de communication (par exemple dans le cas des fonctions de complexités sous-linéaires), nous ne pourrons pas découper et nous aurons  $\nu_{opt} = L$ .

Nous voulons maintenant résoudre le modèle de référence (1). Il y a trois étapes :

1. nous calculons l'extremum  $\nu_{min}$  de (3) en supposant à priori que les contraintes (2) sont vérifiées;
2. ensuite nous vérifions le domaine de validité des contraintes (2), et pour cela nous calculons  $\nu_{bal}$  la plus grande taille de paquet qui vérifie ces contraintes;
3. et enfin nous en déduisons la taille optimale  $\nu_{opt}$ .

Considérons chacune de ces trois étapes.

- Si les contraintes sont vérifiées la solution de la première étape consiste à chercher  $\nu_{min}$  solution, sur le domaine  $[1, L]$ , de

$$\frac{\partial}{\partial \nu} T_{total}(\nu, L) = 0. \quad (4)$$

Si la solution est multiple, nous choisissons la plus petite, et si elle n'existe pas nous prendrons  $\nu_{min} = L$ . Nous verrons en fait que dans le cas pratique il y a existence et unicité de la solution.

- Maintenant que le minimum de (3) est connu, nous devons intégrer les contraintes. Une autre approche pour optimiser le recouvrement des communications par des calculs consiste à chercher, lorsque c'est possible, la taille de paquet telle que le temps de calcul sur un paquet équilibre le temps de communication sur ce paquet [51]. Comme le “calcul avant” et le “calcul après” jouent un rôle symétrique dans la minimisation de (3), nous pouvons supposer que  $Tcomp_A > Tcomp_B$ . L'équilibre de la contrainte est donné par

$$Tcomp_A(\nu, L) = Tcomm(\nu). \quad (5)$$

Soit  $\nu_{bal}$  la solution de cette équation. Généralement  $\nu_{bal}$  n'est pas la TOP ; pour un paquet de taille supérieure à  $\nu_{bal}$ , nous pouvons réécrire l'expression de  $Ttotal(\nu, L)$

$$Ttotal(\nu, L) = Tcomp_A(\nu, L) + \frac{L}{\nu} Tcomp_A(\nu, L) + Tcomp_B(\nu, L). \quad (6)$$

Si  $Tcomp_A(\nu, L)$  est sur-linéaire en  $\nu$  alors  $Ttotal(\nu, L)$  augmente avec  $\nu$ , et  $\nu_{bal}$  minimise  $Ttotal(\nu, L)$  pour  $\nu \geq \nu_{bal}$ . Dans le cas contraire, il faut chercher  $\nu_{bal}$  qui minimise (6).

- Finalement, nous en déduisons la valeur de  $\nu_{opt}$ . Si  $1 \leq \nu_{bal} \leq L$  alors deux cas sont possibles (voir figure 3.2) :

- si  $\nu_{bal} \leq \nu_{min}$  alors la contrainte intervient et  $\nu_{opt} = \nu_{bal}$ ,
- sinon, la contrainte n'intervient pas et  $\nu_{opt} = \nu_{min}$ .

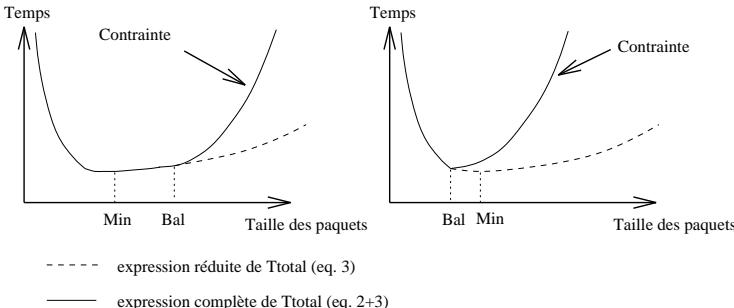


FIG. 3.2: Prise en compte de la contrainte.

### 3.2.3 Etudes de cas pour différentes fonctions de complexités

Dans cette section, nous allons appliquer le schéma général de résolution sur quelques exemples pratiques. Nous nous intéressons plus particulièrement aux fonctions de complexité polynomiale et logarithmique, essentiellement pour pouvoir expliciter des résultats analytiques.

**Lemme 3.** *Nous supposons que*  $\begin{cases} Tcomp_A(\nu) = R_A \nu \tau_A \\ Tcomp_B(\nu) = R_B \nu \tau_B \\ Tcomm(\nu) = \beta + \nu \tau \\ R_A \tau_A > R_B \tau_B \end{cases}$

*Deux cas sont possibles :*

- si  $R_A \tau_A - \tau \leq 0$  alors  $\nu_{opt} = \sqrt{\frac{L \beta}{R_A \tau_A + R_B \tau_B}}$ .
- si  $R_A \tau_A - \tau > 0$  alors
  - si  $L \leq \frac{\beta (R_A \tau_A + R_B \tau_B)}{(R_A \tau_A - \tau)^2}$  alors  $\nu_{opt} = \sqrt{\frac{L \beta}{R_A \tau_A + R_B \tau_B}}$ ,
  - sinon  $\nu_{opt} = \frac{\beta}{R_A \tau_A - \tau}$ .

**Preuve.** Le temps total d'exécution du macro-pipeline est donné par

$$Ttotal(\nu, L) = R_A \nu \tau_A + \frac{L}{\nu} (\beta + \nu \tau) + R_B \nu \tau_B.$$

Nous cherchons à résoudre l'équation  $\frac{\partial}{\partial \nu} Ttotal(\nu, L) = R_A \tau_A - \frac{L \beta}{\nu^2} + R_B \tau_B = 0$ , ce qui conduit à  $\nu_{min} = \sqrt{\frac{L \beta}{R_A \tau_A + R_B \tau_B}}$ . Nous retrouvons le résultat connu pour le cas linéaire [26].

Il faut vérifier les contraintes. Nous supposons que  $R_A \tau_A > R_B \tau_B$ . Nous devons avoir  $R_A \nu \tau_A \leq \beta + \nu \tau$ , donc  $\nu_{bal} = \frac{\beta}{R_A \tau_A - \tau}$ .

Deux cas sont alors possibles :

- si  $R_A \tau_A - \tau \leq 0$  alors la contrainte est toujours vérifiée et  $\nu_{opt} = \nu_{min}$ .
- si  $R_A \tau_A - \tau > 0$  alors :
  - si  $\nu_{min} \leq \nu_{bal}$ , ce qui correspond à avoir  $L \leq \frac{\beta (R_A \tau_A + R_B \tau_B)}{(R_A \tau_A - \tau)^2}$ , alors  $\nu_{opt} = \nu_{min}$ .
  - sinon la contrainte limite la TOP et  $\nu_{opt} = \nu_{bal}$ . □

**Exemple 1.** Pour illustrer ce cas, prenons l'exemple de la factorisation  $LU$  par colonnes. Nous utilisons une distribution cyclique pour bien équilibrer la charge en calcul tout au long de l'exécution [65]. Dans ces conditions, on peut isoler trois phases pour chaque étape de la factorisation (voir figure 3.3) :

- Le *calcul des contributions locales* : le processeur qui détient la colonne pour l'étape courante calcule les contributions et les envoie aux autres processeurs,

- La *mise à jour locale* : ce même processeur met à jour ses colonnes locales,
- La *mise à jour distante* : les autres processeurs reçoivent les contributions, et mettent à jour leurs propres colonnes.

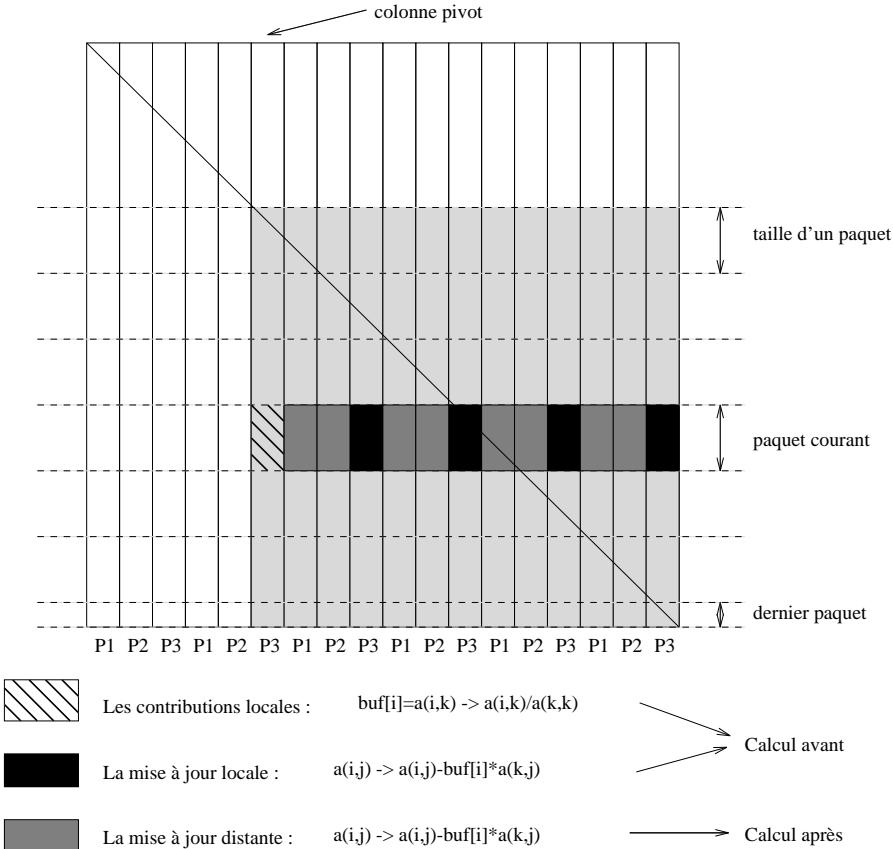


FIG. 3.3: Factorisation LU par colonnes.

Grâce à la distribution cyclique des colonnes, le schéma de factorisation est bien équilibré. Pour chaque étape de la factorisation, tous les processeurs ont un calcul de mise à jour équivalent, et le processeur qui détient la colonne courante peut utiliser une primitive de communication de type “diffusion” pour envoyer ses contributions.

Nous avons clairement un schéma macro-pipeline. Le calcul des contributions locales doit faire partie du “calcul avant” et la mise à jour distante doit faire partie du “calcul après”. La mise à jour locale fait partie du “calcul avant” de telle sorte que tous les processeurs aient un calcul de mise à jour équivalent. Dans ces conditions le recouvrement aura une efficacité maximale.

Pour cette implémentation, les complexités du “calcul avant” et du “calcul après” sont de l’ordre de  $\theta(\Delta \nu)$ , où  $\Delta$  représente le nombre de colonnes locales à modifier.

Il est important de noter que ces complexités dépendent de l'implémentation et de la distribution des données, et plus particulièrement, de la façon dont intervient le nombre de processeurs dans le découpage du macro-pipeline. En effet la factorisation  $LU$  par colonne est typiquement un algorithme  $\theta(\nu)/\theta(\nu^2)$  – la mise à jour de la colonne courante est linéaire et la mise à jour de la sous matrice à sa droite est quadratique – alors que nous avons un macro-pipeline  $\theta(\nu)/\theta(\nu)$ .

Nous pouvons remarquer que  $R_A$  et  $R_B$  dépendent de la TDP ( $R_A = R_B = L/NPROC$  où  $NPROC$  représente le nombre de processeurs), et donc, en accord avec les résultats du lemme 3,  $\nu_{min} = \sqrt{\frac{NPROC\beta}{\tau_A + \tau_B}}$  qui est indépendant de la TDP.

**Lemme 4.** Nous supposons que  $\begin{cases} Tcomp_A(\nu) = R_A \nu^2 \tau_A \\ Tcomp_B(\nu) = R_B \nu \tau_B \\ Tcomm(\nu) = \beta + \nu \tau \end{cases}$

Le domaine de validité de la TOP est  $\nu \in \left[1, \frac{\tau + \sqrt{\tau^2 + 4 R_A \tau_A \beta}}{2 R_A \tau_A}\right]$  et il existe une et une seule solution réelle positive  $\nu_{min}$ .

**Preuve.** Le temps total d'exécution du macro-pipeline est donné par

$$Ttotal(\nu, L) = R_A \nu^2 \tau_A + \frac{L}{\nu} (\beta + \nu \tau) + R_B \nu \tau_B.$$

La contrainte  $R_A \nu^2 \tau_A \leq \beta + \nu \tau$  impose le domaine de validité de la TOP :

$$\nu \in [1, \nu_{bal}] \text{ avec } \nu_{bal} = \frac{\tau + \sqrt{\tau^2 + 4 R_A \tau_A \beta}}{2 R_A \tau_A}.$$

Nous avons à résoudre  $\frac{\partial}{\partial \nu} Ttotal(\nu, L) = 0$ , que nous pouvons écrire sous la forme suivante ( $\nu \neq 0$ ) :

$$2 R_A \nu^3 \tau_A + R_B \nu^2 \tau_B - L \beta = 0.$$

Nous posons  $A = \frac{R_B \tau_B}{2 R_A \tau_A}$  et  $B = \frac{L \beta}{2 R_A \tau_A}$ . Les équations de Jordan donnent 3 solutions (réelles ou complexes) :

$$\begin{cases} \nu_1 = \sqrt[3]{\frac{-q+\delta}{2}} + \sqrt[3]{\frac{-q-\delta}{2}} - \frac{A}{3} \\ \nu_2 = \sqrt[3]{\frac{-q+\delta}{2}} \cdot j + \sqrt[3]{\frac{-q-\delta}{2}} \cdot j^2 - \frac{A}{3} \\ \nu_3 = \sqrt[3]{\frac{-q+\delta}{2}} \cdot j^2 + \sqrt[3]{\frac{-q-\delta}{2}} \cdot j - \frac{A}{3} \end{cases} \text{ avec } \begin{cases} p = -\frac{A^2}{3} \\ q = \frac{2A^3}{27} - B \\ \delta^2 = \frac{4p^3 + 27q^2}{27} \end{cases}$$

Maintenant il faut vérifier l'unicité de la solution. Si  $\delta^2 > 0$  alors nous avons une seule solution réelle sinon nous avons 3 solutions. Dans ce dernier cas, nous pouvons réécrire les solutions sous la forme

$$\nu_{k+1} = \frac{2A}{3} \cdot \cos\left(\frac{\phi}{3} + \frac{2k\pi}{3}\right) - \frac{A}{3} \quad \text{with } k = 0, 1, 2 \quad \text{and } \cos(\phi) = \frac{3B}{2A^3} - \frac{1}{9}$$

et seulement une de ces solutions est positive. Enfin, il faut vérifier que cette solution fait partie du domaine de validité (simple comparaison entre  $\nu_{min}$  et  $\nu_{bal}$ ).  $\square$

**Lemme 5.** Nous supposons que

$$\begin{cases} Tcomp_A(\nu) = R_A \ln(\nu) \tau_A \\ Tcomp_B(\nu) = R_B \ln(\nu) \tau_B \\ Tcomm(\nu) = \beta + \nu \tau \end{cases}$$

alors  $\nu_{min} = \frac{L\beta}{R_A \tau_A + R_B \tau_B}$ .

Le calcul de  $\nu_{min}$  est évident. Néanmoins, pour calculer  $\nu_{bal}$ , nous avons à résoudre l'équation  $R_A \ln(\nu) \tau_A = \beta + \nu \tau$  dont nous ne pouvons pas extraire la solution explicitement. Nous sommes confronté au même problème lorsque nous cherchons à calculer  $\nu_{min}$  avec, par exemple, un macro-pipeline du type  $\theta(\nu \ln(\nu)) / \theta(\nu \ln(\nu))$  ou du type  $\theta(\nu^n) / \theta(\nu^m)$  avec  $n, m > 4$ .

Nous voyons donc sur ce dernier exemple qu'il n'est pas toujours possible de calculer explicitement la TOP. Pour des fonctions de complexités d'ordre élevé ou pour des modèles non-linéaires de communication, nous utiliserons des schémas de résolution numérique de type Newton ou Brent [13].

**Lemme 6.** Nous supposons que

$$\begin{cases} Tcomp_A(\nu) = R_A \nu^{\alpha_A} \tau_A \\ Tcomp_B(\nu) = R_B \nu^{\alpha_B} \tau_B \\ Tcomm(\nu) = \beta + \nu \tau \end{cases}$$

avec  $\alpha_A, \alpha_B > 1$ .

Il existe une solution  $\nu_{min}$  unique.

**Preuve.** Nous avons  $\frac{\partial}{\partial \nu} Ttotal(\nu) = \alpha_A R_A \nu^{\alpha_A - 1} \tau_A + \alpha_B R_B \nu^{\alpha_B - 1} \tau_B - \frac{L\beta}{\nu^2}$ . Nous supposons  $\nu \neq 0$ , il faut donc résoudre  $\alpha_A R_A \nu^{\alpha_A + 1} \tau_A + \alpha_B R_B \nu^{\alpha_B + 1} \tau_B = L\beta$ . Le terme gauche de cette équation est une fonction monotone de  $\nu$  variant de  $[0, +\infty]$  sur  $[0, +\infty]$ . En effet, nous vérifions que sur l'intervalle  $[0, +\infty]$  nous avons

$$\frac{\partial^2}{\partial \nu^2} Ttotal(\nu) = \alpha_A (\alpha_A - 1) R_A \nu^{\alpha_A - 2} \tau_A + \alpha_B (\alpha_B - 1) R_B \nu^{\alpha_B - 2} \tau_B + \frac{2L\beta}{\nu^3} > 0.$$

Il existe donc une solution  $\nu_{min}$  unique sur le domaine  $[0, +\infty]$ . Comme dans le lemme précédent,  $\nu_{bal}$  ne peut pas être calculé explicitement.  $\square$

**Remarque.** Ce résultat est important dans le cas où nous sommes amené à utiliser un solveur numérique pour résoudre l'équation (3), l'unicité de la solution assurant une bonne convergence du schéma itératif.

Dans certains cas, il est possible d'utiliser des bibliothèques optimisées pour améliorer les performances en calcul. C'est le cas des bibliothèques BLAS en algèbre linéaire ; dans ce cas précis, il faut que la taille des données traitées soit suffisante pour obtenir une bonne efficacité. Dans notre cadre de travail, il faudrait donc modifier l'expression des temps de calcul  $Tcomp_A$  et  $Tcomp_B$  avec un terme multiplicatif mesurant le gain apporté par les primitives BLAS en fonction de la taille des paquets. Le même type de modification est envisageable pour tenir compte des phénomènes de cache, mais la modélisation de l'influence du cache en fonction de la taille des paquets semble plus délicate.

### 3.2.4 Comment éviter de recalculer la taille optimale des paquets ?

Dans la pratique, les macro-pipelines sont souvent enchaînés de telle sorte que seule la TDP varie. Il n'est donc pas toujours nécessaire de recalculer la TOP pour chaque macro-pipeline. En fait, il est parfois possible de trouver l'écart  $l_{max}$  entre deux tailles de problèmes pour lequel la taille optimale reste inchangée.

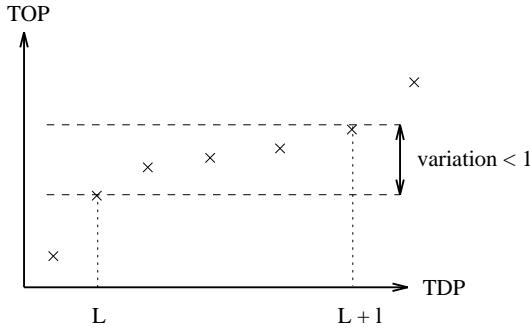


FIG. 3.4: Ecart maximal n'engendrant pas de re-calculation.

**Définition 7.** Nous supposons que l'expression analytique de  $\nu_{opt}(L)$  est connue. L'écart de taille de problème correspondant à une TOP constante est donné par le plus grand entier  $l_{max}$  parmi tous les  $l \geq 0$  tel que (voir figure 3.4)

$$|\nu_{opt}(L+l) - \nu_{opt}(L)| < 1. \quad (7)$$

**Lemme 8.** Si  $\nu_{opt}(L)$  peut être réécrit sous la forme  $CL^\gamma$ , avec  $C > 0$  et  $\gamma > 0$ , alors  $l_{max} = (L^\gamma + 1/C)^{1/\gamma} - L$ .

**Preuve.** Nous avons

$$\begin{cases} \nu_{opt}(L) &= CL^\gamma \\ \nu_{opt}(L+l) &= C(L+l)^\gamma \end{cases}$$

Nous cherchons  $l \geq 0$  tel que  $(L+l)^\gamma - L^\gamma < 1/C$  c'est-à-dire  $L+l < (L^\gamma + 1/C)^{1/\gamma}$ . Nous trouvons donc  $l_{max} = (L^\gamma + 1/C)^{1/\gamma} - L$ .  $\square$

**Remarque.**

- Considérons l'exemple linéaire (lemme 3). Nous avons  $C = \sqrt{\frac{\beta}{R_A \tau_A + R_B \tau_B}}$  et  $\gamma = 1/2$ , donc

$$l_{max} = \frac{R_A \tau_A + R_B \tau_B}{\beta} + 2 \sqrt{\frac{L(R_A \tau_A + R_B \tau_B)}{\beta}}.$$

- Pour la factorisation LU par colonnes (voir exemple 1), la taille optimale est indépendante de la TDP et donc nous ne calculons  $\nu_{opt}$  qu'une seule fois.
- Si l'expression analytique de  $\nu_{opt}$  ne peut pas être mise sous la forme  $CL^\gamma$  (voir lemme 4), la valeur de  $l_{max}$  peut être calculée à l'aide d'un schéma numérique.

## 3.3 Implémentation de la bibliothèque OPIUM

### 3.3.1 Implémentation de l'algorithme générique

Dans ce paragraphe, nous présentons l'algorithme utilisé dans la bibliothèque OPIUM pour calculer la TOP dans le cas général ; nous montrons l'enchaînement des calculs et décrivons quelques détails d'implémentation. L'algorithme est commun à toutes les méthodes (voir figure 3.5).

Si la TDP ne varie pas de manière régulière au cours du calcul, il peut être utile de mettre à jour une “table de hachage” qui enregistre la TOP et l'écart  $l_{max}$  associé pour une taille de problème donnée.

Lorsque nous utilisons un schéma numérique pour calculer la taille optimale, il est souvent difficile d'estimer l'écart  $l_{max}$ , mais nous pouvons utiliser la table pour trouver la TOP correspondant à la TDP la plus proche, et l'utiliser comme valeur initiale pour le schéma itératif. De cette façon nous réduisons fortement le nombre d'itérations ; le schéma numérique est alors parfois plus rapide que le calcul direct de la TOP.

Pour une TDP donnée, on calcule la TOP :

rechercher la plus proche TOP déjà calculée dans la table

**Si** l'écart  $l_{max}$  correspondant est atteint **Alors**

- Si** on doit recalculer  $\nu_{bal}$  **Alors**
  - calculer  $\nu_{bal}$
  - vérifier la validité de  $\nu_{bal}$
  - calculer  $\nu_{min}$
  - vérifier la validité de  $\nu_{min}$
  - calculer le nouvel écart  $l_{max}$
  - mettre à jour la table
- $\text{TOP} = \min(\nu_{bal}, \nu_{min})$

FIG. 3.5: Algorithme principal utilisé dans la bibliothèque OPIUM.

### 3.3.2 La Bibliothèque OPIUM

Nous présentons maintenant l'interface de la bibliothèque OPIUM (Optimal Pa-cket sIze compUtation Methods). La version actuelle de cette bibliothèque permet le calcul direct de la TOP dans les cas linéaires et quadratiques. Elle contient également une fonction de calcul générale, utilisant un solveur de type Newton, qui accepte des fonctions de complexités quelconques.

La bibliothèque OPIUM offre une interface simple d'utilisation pour calculer la TOP sans connaissance, à priori, du calculateur cible et des méthodes utilisées. Elle contient en particulier 4 fonctions (2 pour le cas linéaire et 2 pour la méthode générique) que nous illustrons ci-dessous.

Cas linéaire :

```
struct comp_linear
{
    double (*coeff_fct)(int),
    double tau_comp;
    flag flag_same_func;
    flag flag_recalc;
}

struct comm_linear
{
    double startup;
    double bandwidth;
}

Opium_Init_Linear(
    struct comp_linear comp_fwd,
    struct comp_linear comp_bwd,
    struct comm_linear comm);

Opium_Linear(int pipeline_size);
```

Cas générique :

```
struct comp_generic
{
    double (*coeff_fct)(int);
    double tau_comp;
    double (*complexity_fct)(double);
    double (*complexity_derivate_fct)(double);
    flag flag_same_func;
    flag flag_recalc;
}

struct comm_generic
{
    double (*comm_fct)(double);
    double (*comm_derivate_fct)(double);
}

Opium_Init_Generic(
    struct comp_generic comp_fwd,
    struct comp_generic comp_bwd,
    struct comm_generic comm,
    double precision);

Opium_Generic(int pipeline_size);
```

Dans le cas linéaire, il faut donner la fonction décrivant la constante de complexité, à la fois pour le “calcul avant” et le “calcul après” (rappelons qu’elle peut dépendre de la TDP). Pour cela, nous utilisons la structure de donnée **comp\_linear**. Les paramètres du modèle de communication sont aussi stockés dans **comm\_linear**. Ces paramètres peuvent éventuellement être calculés lors de l’exécution à l’aide de “benchmarks”. La fonction **Opium\_Init\_linear** doit être appelée pour initialiser ces différents paramètres et effectuer quelques pré-calculs. Ensuite, nous appelons **Opium\_Linear** pour calculer la taille optimale de paquet pour un TDP donné.

Le cas générique utilise le même schéma, mais les fonctions décrivent les complexités des calculs et du modèle de communication ainsi que leur dérivée. L’utilisateur remplit les structures de données, appelle **Opium\_Init\_Generic** puis, lorsque c’est nécessaire, **Opium\_Generic**.

Toutes les complexités sont décrites à l'aide de pointeurs de fonction **C**. Le premier paramètre (**flag\_same\_func**) est utilisé pour indiquer si le “calcul avant” et le “calcul après” admettent la même fonction de complexité. Si c'est le cas, on

n'évaluera la fonction qu'une seule fois. Le second paramètre (`flag_recalc`) permet de préciser si la constante de complexité dépend de la TDP, auquel cas nous ne recalculons pas la valeur de  $\nu_{bal}$  pour chaque macro-pipeline. La valeur de `precision` est utilisée comme test d'arrêt pour le schéma itératif.

L'exemple suivant montre comment utiliser la bibliothèque OPIUM pour la factorisation *LU*. Nous utilisons le schéma générique. La primitive `Opium_Init_Generic` n'est appelée qu'une seule fois ; `Opium_Generic` calcule alors la taille optimale de paquet `TOP` qui est ensuite utilisée dans la fonction `loccks_oto`. Cette fonction implémente le macro-pipeline. Grâce à nos optimisations, la `TOP` ne sera pas re-calculée à chaque étape, même si l'appel à la fonction est effectué à chaque fois.

“Calcul Avant” :

```
void sender_fwd(...) {
    for /* taille du paquet */
        buffer[i]=A(i,k)/=A(k,k);

    for /* toutes les colonnes locales */
        for /* taille du paquet */
            A(i,j) = A(i,j) - buffer[i]*A(k,j);
}
```

“Calcul Après” :

```
void receiver_bwd(...) {
    for /* toutes les colonnes locales */
        for /* taille du paquet */
            A(i,j) = A(i,j) - buffer[i]*A(k,j);
}
```

Programme principal :

```
double nb_local_columns_to_modify(int ADP) { return ADP/NBPROC; }

double func_complex(double x) { return x; }

double func_complex_derivate(double x) { return 1.0; }

double type_comm(double x) { return beta+x*tau; }

double tau_fwd/* temps élémentaire du calcul avant */;
double tau_bwd/* temps élémentaire du calcul après */;

struct comp_generic fwd_comp =
{ nb_local_columns_to_modify, tau_fwd, func_complex, func_complex_derivate };

struct comp_generic bwd_comp =
{ nb_local_columns_to_modify, tau_bwd, func_complex, func_complex_derivate };

/* initialisations OPIUM */
Opium_Init_Generic(fwd_comp, bwd_comp, type_comm, precision, SAME_FUNC, RECOMP);
/*... initialisation du buffer LOCCTS ...*/
for /* toutes les colonnes de la matrice */
    /* calcul de la taille optimale OPIUM */
    OPS=Opium_Generic(ADP);
    if /* émetteur */
        /*... initialisations LOCCTS ...*/
        loccs_oto(OPS, ADP, sender_fwd, NULL, ...);
    else /* receveur */
        /*... initialisations LOCCTS ...*/
        loccs_oto(OPS, ADP, NULL, receiver_bwd, ...);
```

## 3.4 Résultats expérimentaux

### 3.4.1 Factorisation LU par colonnes et par blocs colonnes

Dans ce paragraphe nous allons évaluer la validité du modèle théorique et l'efficacité du recouvrement sur l'exemple de la factorisation *LU* par colonnes (voir exemple 1) sur un Paragon Intel et un SP2 IBM. Nous utilisons la bibliothèque LOCCS qui offre des primitives efficaces pour gérer le recouvrement [26, 32]. Nous utilisons les paramètres suivants (en secondes), spécifiques à chaque architecture cible :

Paragon Intel :	SP2 IBM :
$T_{comm}(\nu) = 3 \cdot 10^{-5} + 2 \cdot 10^{-9}\nu$ $\tau_A = \tau_B = 5.2 \cdot 10^{-7}$	$T_{comm}(\nu) = 1.4 \cdot 10^{-4} + 4.6 \cdot 10^{-8}\nu$ $\tau_A = \tau_B = 3.8 \cdot 10^{-7}$

Les paramètres de calcul sont les résultats de mesures des temps d'exécution des implémentations du “calcul avant” et du “calcul après”. Les paramètres de communication sont les résultats de mesures sur les primitives de communication utilisées dans l'implémentation de l'algorithme.

Il faut noter que nous n'avons pas cherché dans un premier temps à optimiser complètement les performances du calcul ; en particulier, nous ne faisons pas appel aux primitives BLAS pour rester conforme aux fonctions de complexité de l'exemple.

Les performances sont mesurées sur un Paragon Intel avec 32 nœuds sous la bibliothèque Nx et sur un SP2 IBM à 16 nœuds sous MPI. Rappelons que grâce à la distribution cyclique des colonnes, le processeur qui détient la colonne pivot envoie ses contributions à tous les autres processeurs. La nature du macro-pipeline nous permet d'effectuer cette communication sur un anneau de processeur ; la routine qui implémente le macro-pipeline sur un anneau de processeurs a été ajoutée à la bibliothèque LOCCS. Remarquons toutefois que l'algorithme LU original permet déjà un recouvrement calcul/communication entre les étapes de calcul de chaque colonne, entraînant un gain moins important que prévu.

Le gain apporté par le recouvrement décroît avec le nombre de processeurs, cela est dû au fait que le travail exécuté par chaque processeur devient trop faible pour que le recouvrement joue un rôle significatif (voir figure 3.6 et figure 3.12). On remarque cependant que le rapport entre les deux méthodes (avec et sans recouvrement) est pratiquement constant, ce qui traduit une bonne extensibilité de l'optimisation.

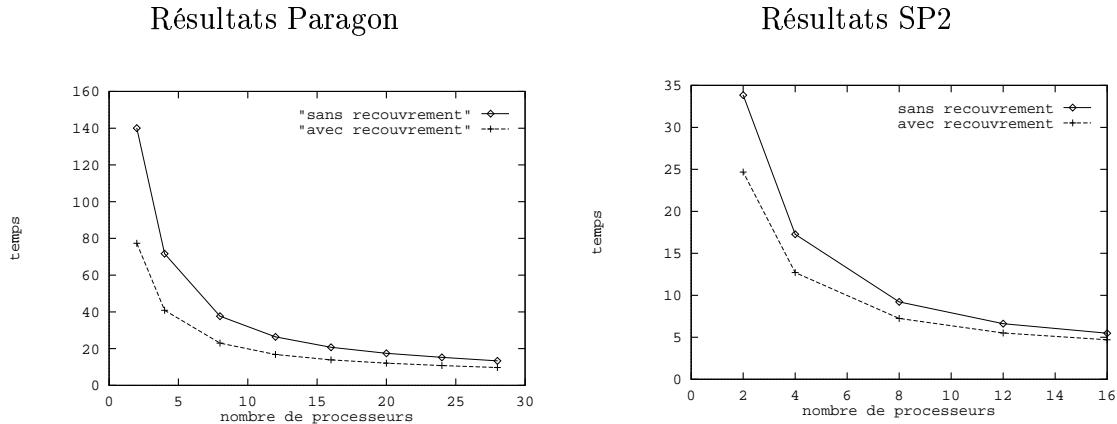


FIG. 3.6: Temps de factorisation d'une matrice  $960 \times 960$  (avec et sans recouvrement) en fonction du nombre de processeurs.

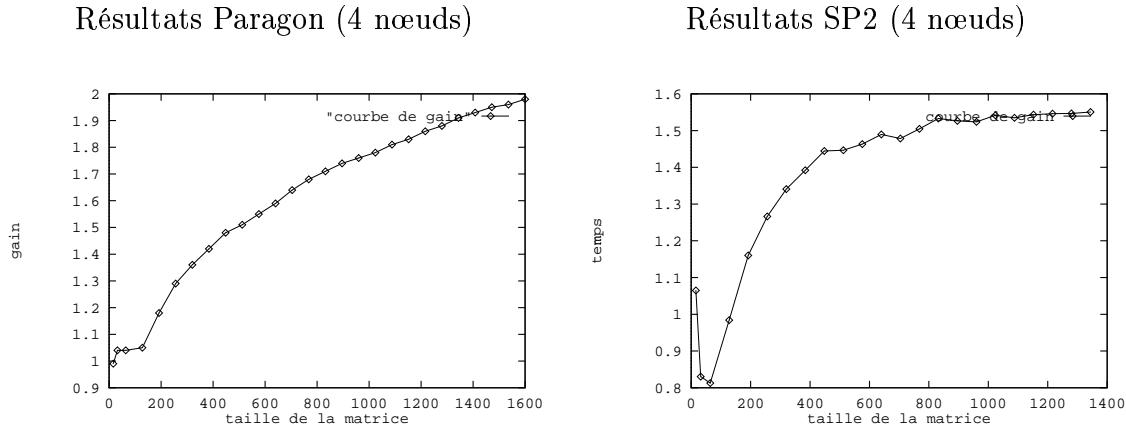


FIG. 3.7: Rapport entre les temps de factorisation avec et sans recouvrement en fonction de la taille de la matrice.

La courbe du gain, figure 3.7, montre que pour une matrice de taille suffisamment grande, le temps de factorisation avec recouvrement est réduit de manière significative par rapport au temps de factorisation sans recouvrement (entre 1.6 et 2). Quelle que soit la taille de la matrice, nous voyons également, au moins dans le cas du Paragon, que le surcoût engendré par la gestion du recouvrement est négligeable ( $\text{gain} > 1$ ). Remarquons que pour  $L < 100$ , nous ne découpons pas car  $\nu_{opt} > L$ .

Dans le cas du SP2, nous notons que la TOP est sensiblement plus importante que celle obtenue sur la Paragon, ce qui dégrade l'effet du recouvrement.

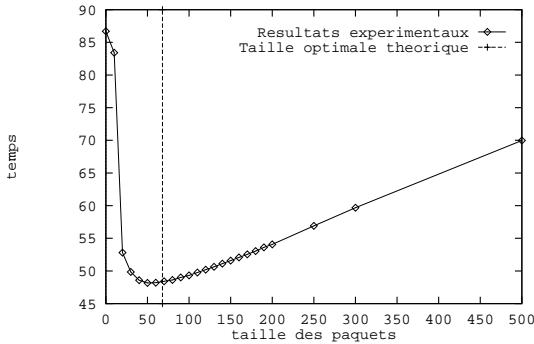
La taille prédite et calculée à partir de notre modèle s'avère être très proche de la valeur obtenue expérimentalement (voir figure 3.8); et de plus, elle est comme

prévue constante pour chaque macro-pipeline jusqu'à l'intervention de la contrainte ; elle n'est donc calculée qu'une seule fois.

Les tests réalisés pour comparer la méthode de calcul générique (avec le schéma numérique) et la méthode analytique directe montrent que les TOP sont identiques dans les deux cas et, que si la TDP varie progressivement au cours de l'algorithme, le surcoût engendré par le schéma itératif est négligeable (l'utilisation des TOP précédemment calculées permet de diminuer sensiblement le nombre d'itérations).

Nous voyons sur les figures 3.9 et 3.11 que le recouvrement est d'autant plus efficace que la taille des données à pipeliner est importante. La figure 3.10 représente l'activité des 4 processeurs du Paragon alloués pour la factorisation d'une matrice de taille 512 avec et sans recouvrement. Ce diagramme montre la bien meilleure utilisation du parallélisme avec les mécanismes de recouvrement.

Résultats Paragon (4 nœuds)



Résultats Paragon (4 nœuds)

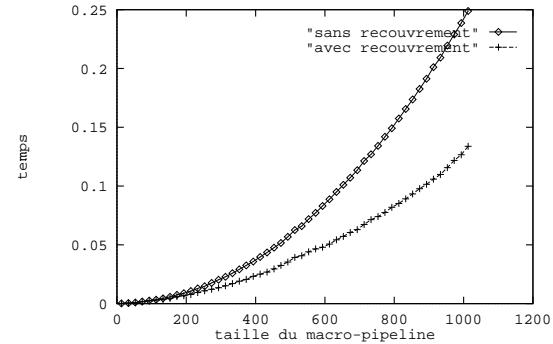


FIG. 3.8: Temps de factorisation d'une matrice  $1024 \times 1024$  en fonction de la taille des paquets.

FIG. 3.9: Temps d'exécution de chaque étape de la factorisation d'une matrice  $1024 \times 1024$  (avec et sans recouvrement).

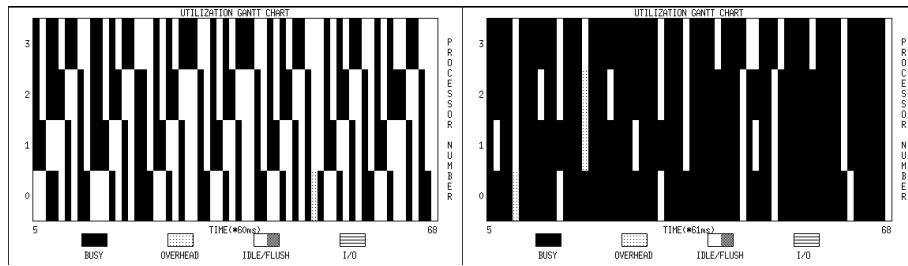
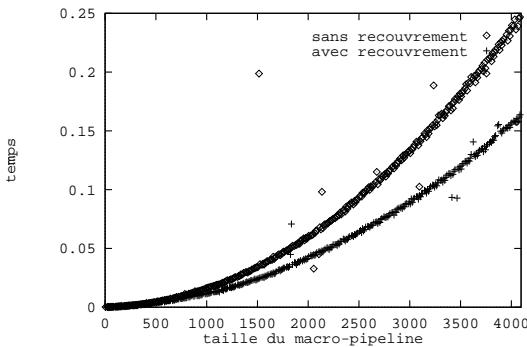
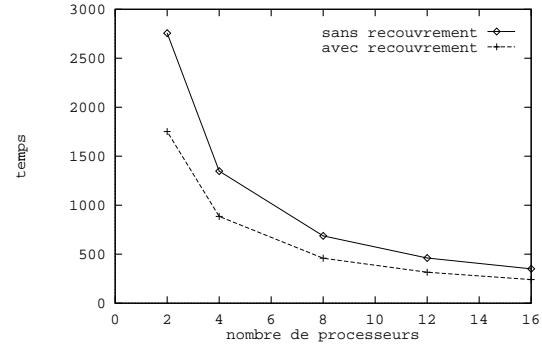


FIG. 3.10: Diagrammes Paragraph de Gantt (en blanc l'inactivité et en noir l'activité des processeurs) sans et avec recouvrement.

Résultats SP2 (16 noeuds)

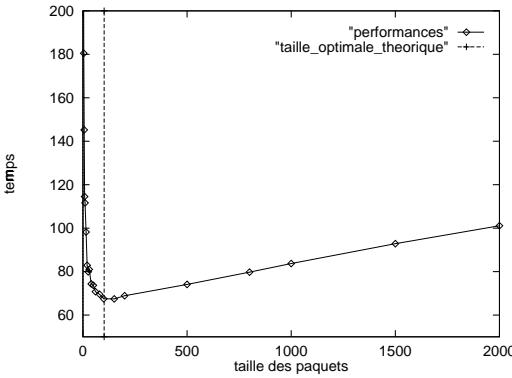
FIG. 3.11: Temps d'exécution de chaque étape de la factorisation d'une matrice  $4096 \times 4096$  (avec et sans recouvrement).

Résultats SP2

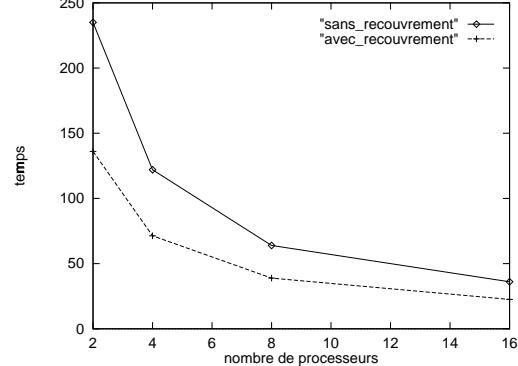
FIG. 3.12: Temps de factorisation d'une matrice  $4096 \times 4096$  (avec et sans recouvrement) en fonction du nombre de processeurs.

Nous présentons maintenant les résultats obtenus avec une distribution par bloc-colonne offrant la possibilité d'effectuer des traitements par bloc et donc d'utiliser les opérateurs BLAS 3. C'est expérimentations ont été effectuées sur un SP2 IBM ; les fonctions de complexité dans “calcul avant” et “calcul après” sont modifiées pour prendre en compte la taille des paquets correspondant à des blocs de contributions et non plus à des vecteurs de contributions.

Résultats SP2 (16 noeuds)

FIG. 3.13: Temps de factorisation par bloc d'une matrice  $3360 \times 3360$  en fonction de la taille des paquets.

Résultats SP2

FIG. 3.14: Temps de factorisation par bloc d'une matrice  $3360 \times 3360$  (avec et sans recouvrement) en fonction du nombre de processeurs.

Nous pouvons remarquer que l'efficacité des opérateurs BLAS dépend de la taille de paquet  $\nu$ ; ceci peut être pris en compte en considérant que les variables  $\tau_A$  et  $\tau_B$  restent constantes mais en intégrant un facteur d'efficacité dépendant de  $\nu$  dans les fonctions  $f_A$  et  $f_B$  (voir la définition 1).

De même que pour la version colonne, nous voyons sur les figures 3.13 et 3.14 le gain apporté par une bonne gestion du recouvrement; la taille prédite et calculée à partir de notre modèle s'avère également très proche de la valeur obtenue expérimentalement.

### 3.4.2 Algorithme du SOR

Dans ce paragraphe nous allons évaluer la validité du modèle théorique et l'efficacité du recouvrement sur l'exemple de l'algorithme du *SOR*<sup>3</sup> sur un SP2 IBM. Dans cet exemple, les étapes de “calcul avant” et “calcul après” font intervenir des communications indépendantes du macro-pipeline (voir la remarque page 23).

Il s'agit d'un algorithme proche de celui de Jacobi. Etant donné une grille de points  $U(i, j)$ , nous calculons pour l'itération  $k + 1$  la valeur  $U_{k+1}(i, j)$  à l'aide des informations de voisinage  $U_{\text{latest}}(i-1, j)$ ,  $U_{\text{latest}}(i, j-1)$ ,  $U_{\text{latest}}(i+1, j)$  et  $U_{\text{latest}}(i, j+1)$ ; *latest* prend la valeur  $k$  ou  $k + 1$  suivant l'ordre dans lequel sont évaluées les boucles en  $i$  et en  $j$ .

Si nous considérons, une progression de type Gauss-Seidel nous aurons la formule suivante :

$$U_{k+1}(i, j) = \frac{U_k(i, j) + U_{k+1}(i - 1, j) + U_{k+1}(i, j - 1) + U_k(i + 1, j) + U_k(i, j + 1)}{4}$$

Nous considérons une distribution des données de type **BLOCK** sur les lignes, nous voyons donc une synchronisation forte imposée par cette progression ligne (voir figure 3.15), le processeur  $P_2$  doit attendre les mises à jour des points frontières avec  $P_1$ . Si maintenant nous autorisons à communiquer partiellement les données frontières, il est possible d'obtenir un meilleur recouvrement des calculs et des communications (voir figure 3.16).

---

<sup>3</sup>Successive Over-Relaxation

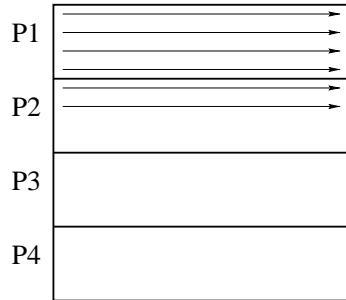


FIG. 3.15: SOR sans pipeline.

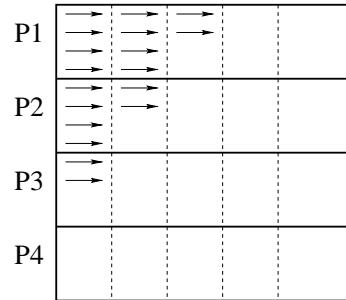


FIG. 3.16: SOR avec pipeline.

La communication “descendante” ( $P_i$  vers  $P_{i+1}$ ) correspond clairement au macro-pipeline. Par contre les mises à jour des valeurs frontières pour la communication “montante” ( $P_i$  vers  $P_{i-1}$ ) ne sont synchronisantes que pour l’itération suivante (voir figure 3.17). Par opposition à l’étude expérimentale précédente (LU), nous autorisons donc les phases de “calcul avant” et “calcul après” à effectuer les communications “montantes” partielles ; ce surcoût devra bien sûr être pris en compte dans le coût des phases du macro-pipeline.

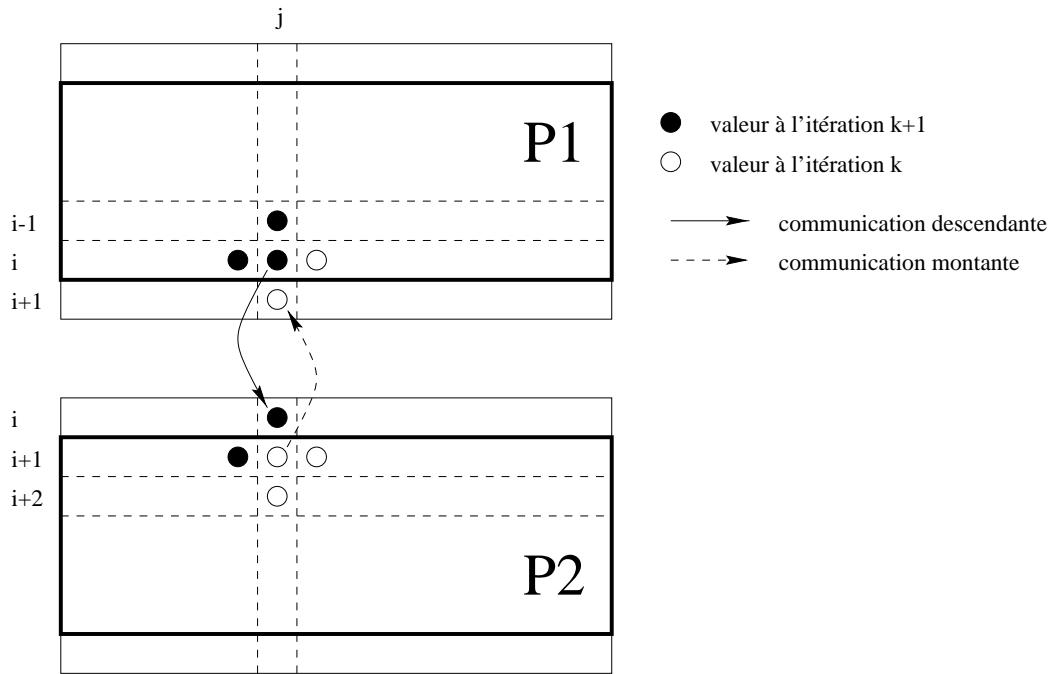


FIG. 3.17: Détail sur une frontière.

Les figures suivantes présentent les performances obtenues, avec et sans recouvrement, sur un SP2 IBM, pour 10 itérations de l’algorithme du SOR sur une matrice de dimension  $3360 \times 3360$ . Les coûts des “calculs avant” et des “calculs après”

prennent en compte le temps des communications “montantes”. Ces communications sont effectuées de manière asynchrone et c'est donc essentiellement le surcoût lié à l'initialisation qui est comptabilisé.

On voit sur les figures 3.18 et 3.19 un gain significatif pour la version avec recouvrement. La méthode présente de plus une très bonne scalabilité. La taille prédictée et calculée à partir de notre modèle s'avère être très proche de la valeur obtenue expérimentalement (voir figure 3.20).

Résultats SP2 (16 nœuds)

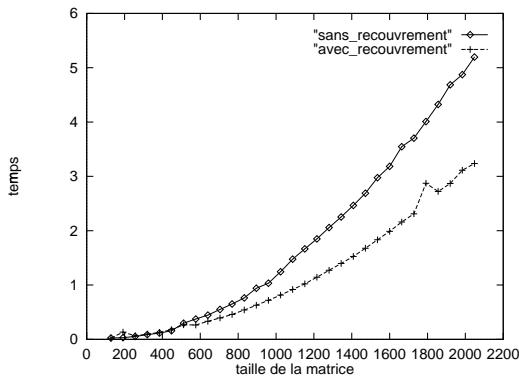


FIG. 3.18: Temps du SOR (avec et sans recouvrement) en fonction de la taille de la matrice.

Résultats SP2

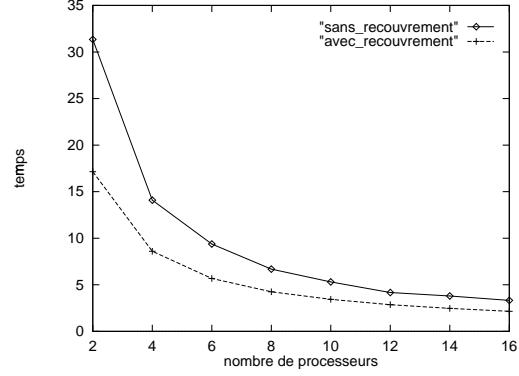


FIG. 3.19: Temps du SOR pour une matrice  $3360 \times 3360$  (avec et sans recouvrement) en fonction du nombre de processeurs.

Résultats SP2 (16 noeuds)

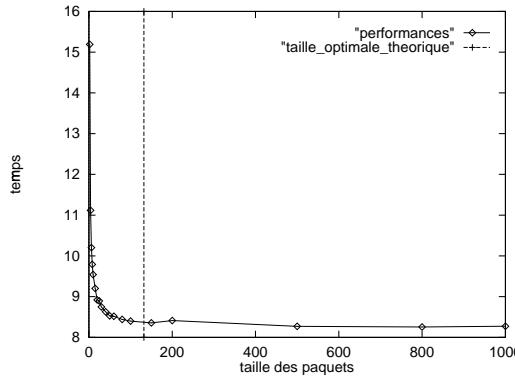


FIG. 3.20: Temps du SOR pour une matrice  $3360 \times 3360$  en fonction de la taille des paquets.

## 3.5 Conclusion

Dans ce chapitre, nous avons présenté et validé expérimentalement une méthode générale pour le calcul de la taille optimale de paquets dans le cadre du recouvrement calculs/communications. Les résultats obtenus montrent l'efficacité d'une bonne gestion du recouvrement sur une application complète. Nous retrouvons, de manière expérimentale, les résultats théoriques du modèle.

La suite logique de ces travaux sera d'étudier l'extension des mécanismes proposés dans le cadre régulier aux problèmes irréguliers. Nous nous intéressons, dans le chapitre suivant au calcul de la granularité maximisant le recouvrement calcul/communication pour l'algorithme de factorisation de Cholesky pour des matrices pleines en exploitant l'irrégularité due à la symétrie de la matrice.



# Chapitre 4

## Vers un calcul adaptatif de la taille optimale de paquets pour les problèmes irréguliers

### 4.1 Introduction et positionnement du problème

Cette étude se place dans le cadre des problèmes irréguliers. L'irrégularité entraîne, par opposition aux études précédentes, une valeur variable de la taille optimale de paquets. Relativement peu de travaux ont été publiés dans ce cadre ; dans [47], une solution adaptative pour recouvrir les communications est proposée dans le cas du produit matrice-vecteur. Un protocole de communication est proposé dans [59] pour offrir un grain de calcul adaptatif, dans le cadre des calculateurs à mémoire partagée.

Dans ce chapitre, nous présentons un schéma de calcul de la suite de taille de paquets qui optimise le recouvrement pour le problème de la factorisation de Cholesky d'une matrice symétrique. La symétrie du problème engendre déjà une irrégularité dans le sens où la masse des calculs à traiter évolue au cours du macro-pipeline ; à chaque étape de la factorisation, une suite de taille de paquets doit être calculée si l'on veut maintenir un recouvrement optimal. Nous proposons alors une méthode pour déterminer analytiquement le découpage du macro-pipeline, en tenant compte des contraintes de granularité imposées par l'utilisation des opérateurs BLAS. La démarche peut être généralisée pour les algorithmes où la taille des paquets doit évoluer au cours du temps en fonction de la variation des calculs, connue analytiquement à priori. C'est le cas pour une grande partie des problèmes de l'algèbre linéaire creuse. L'objectif étant à terme d'intégrer ces techniques dans le cadre d'un

schéma de compilation automatique, il sera nécessaire d'avoir une analyse du code donnant une estimation fine de la complexité.

Dans la suite du chapitre nous introduisons la formulation du problème et sa résolution, puis nous présentons les résultats obtenus sur un SP2 IBM.

## 4.2 Calcul de la suite optimale

Dans cette partie, nous présentons une méthode pour calculer une suite de taille de paquets qui maximise le recouvrement pour l'algorithme de factorisation de Cholesky. Le point de départ est l'algorithme séquentiel colonne classique que nous avons modifié pour mettre en évidence le macro-pipeline :

Algorithme Séquentiel :

```

Pour k := 1 à n faire
    Pour j := k à n faire
         $a_{j,k} := a_{j,k} / \sqrt{a_{k,k}}$ 
    Pour j := k + 1 à n faire
        Pour i := j à n faire
             $a_{i,j} := a_{i,j} - a_{i,k} * a_{j,k}$ 
```

Algorithme modifié par paquets :

```

Pour k := 1 à n faire
     $a_{k,k} := \sqrt{a_{k,k}}$ 
    Pour p := 1 à  $P_k$  faire
        Pour j :=  $j_p$  à  $j_{p+1} - 1$  faire
             $a_{j,k} := a_{j,k} / \sqrt{a_{k,k}}$ 
        Pour q := 1 à  $p - 1$  faire
            Pour jj :=  $j_q$  à  $j_{q+1} - 1$  faire
                Pour j :=  $j_p$  à  $j_{p+1} - 1$  faire
                     $a_{j,jj} := a_{j,jj} - a_{j,k} * a_{jj,k}$ 
                Pour j :=  $j_p$  à  $j_{p+1} - 1$  faire
                     $a_{i,j} := a_{i,j} - a_{i,k} * a_{j,k}$ 
```

La redistribution des calculs correspond à une progression “ligne” dans la factorisation. La symétrie conduit à des tailles de paquets variables au cours du pipeline. A chaque étape de la factorisation, on découpe la colonne pivot en paquets de tailles inégales ; c'est cette suite de taille de paquets que nous cherchons à optimiser. Pour l'étape  $k$ , il y a  $P_k$  paquets de taille  $\nu_p$  correspondant aux indices de ligne  $j_p$  à  $j_{p+1} - 1$  (voir figure 4.1). Nous calculons les contributions engendrées par le paquet en cours (bloc diagonal) ainsi que le couplage avec les paquets précédents. Il est donc nécessaire de conserver les paquets précédemment reçus tant que le pipeline n'est pas terminé.

Nous appelons “calcul avant” ( $T_{calc_A}$ ) le travail qui doit être effectué sur un paquet avant de communiquer les résultats correspondants (voir paragraphe 3.2.1). De même, le “calcul après” ( $T_{calc_B}$ ) représente le travail qui doit être effectué à l'aide des résultats communiqués.

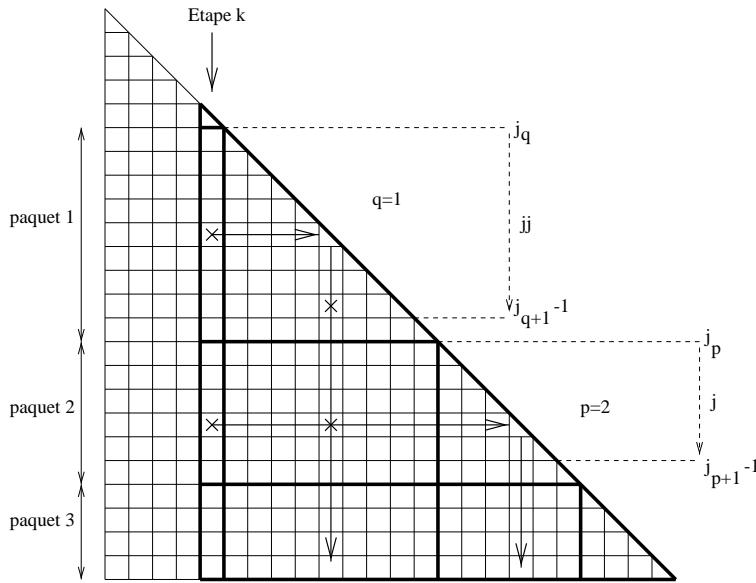


FIG. 4.1: Factorisation de Cholesky.

On peut isoler trois phases pour chaque étape de la factorisation :

- le calcul des contributions locales : le processeur qui détient la colonne pour l'étape courante calcule les contributions et les envoie aux autres processeurs,
- la mise à jour locale : ce même processeur met à jour ses colonnes locales,
- la mise à jour distante : les autres processeurs reçoivent les contributions, et mettent à jour leur propres colonnes.

Nous notons  $\tau_x$  le temps élémentaire de mise à jour et  $\tau_{\div}$  le temps élémentaire du calcul d'une contribution.

Le processeur émetteur effectue les mises à jour de ses colonnes dans le “calcul avant”, de sorte que le “calcul avant” et le “calcul après” soient équilibrés. A  $p$  fixé, nous avons un équilibre calcul avant / calcul après :  $\theta(\nu^2)/\theta(\nu^2)$ .

Pour un paquet  $p$  de taille  $\nu_p$ , pour la colonne  $k$ ,  $1 \leq k \leq n$ ,  $1 \leq p \leq P_k$ , nous pouvons écrire :

$$\left\{ \begin{array}{lcl} T_{calcA}(\nu_p) & = & \frac{\sum\limits_{q=1}^{p-1} \nu_p \nu_q \tau_x + \frac{\nu_p(\nu_p+1)}{2} \tau_x}{NProcs} + \nu_p \tau_{\div} \\ T_{calcB}(\nu_p) & = & \frac{\sum\limits_{q=1}^{p-1} \nu_p \nu_q \tau_x + \frac{\nu_p(\nu_p+1)}{2} \tau_x}{NProcs}. \end{array} \right.$$

En supposant que la taille des paquets est supérieure au nombre de processeurs, et grâce à une *distribution cyclique*, tous les processeurs sont concernés par les contributions et les paquets seront émis à l'aide d'une diffusion sur un anneau.

Lorsque  $p$  croît, la taille du calcul augmente comme  $p\nu_p^2$ , et donc nécessairement la suite de taille de paquets est décroissante. Pour expliciter le couplage entre les temps de calcul et de communication, qui évoluent de manière opposée, nous imposons la contrainte :

$$\forall p, \quad T_{\text{calc}}(\nu_p) + T_{\text{comm}}(\nu_p) = C, \quad (8)$$

avec  $C$  à déterminer (voir proposition 10). Cela revient à imposer une contrainte forte forçant à équilibrer chaque étape du macro-pipeline. D'autre part, nous considérons un modèle linéaire de communication :  $T_{\text{comm}}(\nu_p) = \beta + \nu_p\tau$ .

D'une manière générale (voir figure 4.2), nous pouvons écrire que le temps total d'un pipeline irrégulier est donné par :

$$T(C) = T_{\text{calc}}(\nu_1) + \sum_{p=2}^{P_k(C)-1} \max(T_{\text{calc}}(\nu_p), T_{\text{comm}}(\nu_p)) + T_{\text{calc}}(\nu_{P_k(C)}).$$

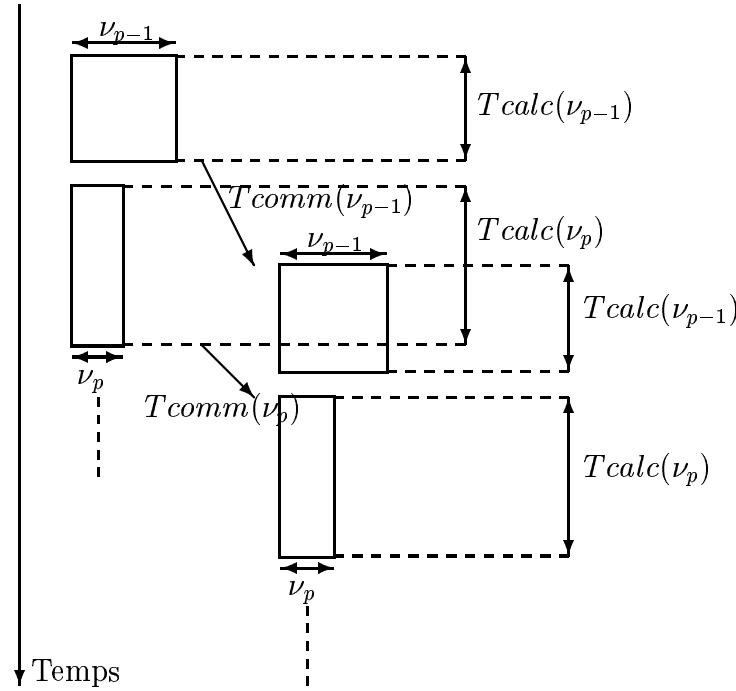


FIG. 4.2: Pipeline dans un cas irrégulier.

Comme la taille des paquets varie de façon monotone, nous cherchons  $p(C)$  réalisant  $T_{\text{calc}}(\nu_{p(C)}) = T_{\text{comm}}(\nu_{p(C)})$ . Si  $2 \leq p(C) \leq P_k(C)$ , le problème du calcul de la distribution optimale de tailles de paquet se ramène à minimiser en  $C$  :

$$T(C) = T_{\text{calc}}(\nu_1) + \sum_{p=2}^{p(C)} T_{\text{comm}}(\nu_p) + \sum_{p=p(C)+1}^{P_k(C)-1} T_{\text{calc}}(\nu_p) + T_{\text{calc}}(\nu_{P_k(C)}).$$

La minimisation de cette fonctionnelle conduit, dans le cas général, à des résultats analytiques peu exploitables. Cependant, pour l'exemple qui nous intéresse, nous obtenons le résultat suivant :

**Proposition 9.** Soit  $L$  la taille du pipeline, la suite de taille de paquets est donnée par :

$$\begin{cases} \nu_1(C) = \frac{-b + \sqrt{b^2 + 4a(C - \beta)}}{2a} \\ \forall p \geq 2, \quad \nu_p(C) = \frac{\sqrt{b^2 + 4ap(C - \beta)} - \sqrt{b^2 + 4a(p-1)(C - \beta)}}{2a} \\ \text{avec } \begin{cases} a = \frac{\tau_{\times}}{2NProcs} \\ b = \frac{\tau_{\times}}{2NProcs} + \tau + \tau_{\div} \end{cases} \end{cases}$$

De plus, le nombre de paquets du pipeline est donné par :  $P_k(C) = \frac{L(b+aL)}{C-\beta}$ .

**Preuve.** Avec l'équation (8) nous pouvons écrire :

$$\forall p, \frac{\frac{1}{2}\nu_p^2\tau_{\times} + \nu_p \left( \frac{1}{2} + \sum_{q=1}^{p-1} \nu_q \right) \tau_{\times}}{NProcs} + (\beta + \nu_p \tau) = C.$$

$$\text{Nous posons } X_p = \sum_{i=1}^p \nu_i.$$

$$\text{Par sommation nous avons : } aX_p^2 + bX_p = p(C - \beta) \text{ avec } \begin{cases} a = \frac{\tau_{\times}}{2NProcs} \\ b = \frac{\tau_{\times}}{2NProcs} + \tau + \tau_{\div} \end{cases}$$

$$\text{donc } X_p = \frac{-b + \sqrt{b^2 + 4ap(C - \beta)}}{2a} \text{ et pour } p \geq 2 \text{ nous avons } \nu_p = X_p - X_{p-1}.$$

$$\text{De plus, pour le premier paquet nous avons : } \frac{\nu_1(\nu_1+1)\tau_{\times}}{2NProcs} + \beta + \nu_1 \tau = C.$$

$$\text{Nous pouvons donc écrire : } \begin{cases} \nu_1(C) = \frac{-b + \sqrt{b^2 + 4a(C - \beta)}}{2a} \\ \forall p \geq 2, \quad \nu_p(C) = \frac{\sqrt{b^2 + 4ap(C - \beta)} - \sqrt{b^2 + 4a(p-1)(C - \beta)}}{2a}. \end{cases}$$

$$\text{Nous remarquons que } X_{P_k} = L \text{ avec } L = n - k \text{ et donc } P_k(C) = \frac{L(b+aL)}{C-\beta}. \quad \square$$

L'efficacité du noyau de calcul BLAS impose une taille minimale de paquets  $\Lambda$ . Ayant posé la contrainte (8), nous pouvons chercher à calculer  $C$  tel que :

$$\begin{cases} \forall p, \nu_p \geq \Lambda \\ \forall p, T_{calc}(\nu_p) \geq T_{comm}(\nu_p) \end{cases}$$

**Proposition 10.**  $C = \max(C_1, C_2)$  avec :

$$\begin{cases} C_1 = \beta + 2aL\Lambda + b\Lambda - a\Lambda^2 \geq 0 \quad \text{obtenue avec la 1}^{\text{ère}} \text{ contrainte,} \\ C_2 = \beta + \frac{2a\beta - 2b\tau + 4\tau^2 + 2\tau\sqrt{b^2 - 4b\tau + 4a\beta + 4\tau^2}}{2a} \quad \text{obtenue avec la 2}^{\text{ème}} \text{ contrainte.} \end{cases}$$

**Preuve.** Comme les paquets sont de tailles décroissantes, on doit vérifier la première contrainte pour le dernier paquet :  $\nu_{P_k(C)} \geq \Lambda$ .

Nous trouvons :  $C \geq \beta + 2aL\Lambda + b\Lambda - a\Lambda^2 \geq 0$  (car  $\Lambda \leq L$ ).

Il faut maintenant vérifier que nous avons bien :  $\forall p, T_{calc}(\nu_p) \geq T_{comm}(\nu_p)$  pour avoir un recouvrement complet. Du fait de la variation monotone de la taille de paquets, il suffit de vérifier que :  $C \leq 2 \cdot T_{calc}(\nu_1)$  (ou encore  $C \geq 2 \cdot T_{comm}(\nu_1)$ ). Nous devons donc avoir :  $C \geq 2\beta + \frac{\tau}{a}(-b + \sqrt{b^2 + 4a(C - \beta)})$ .

Nous trouvons :  $C \geq \beta + \frac{2a\beta - 2b\tau + 4\tau^2 + 2\tau\sqrt{b^2 - 4b\tau + 4a\beta + 4\tau^2}}{2a}$   $\square$

La figure 4.3 illustre l'évolution des tailles de paquets, pour un macro-pipeline de taille 1024, en fonction de la taille critique imposée  $\Lambda$ . Nous voyons que lorsque l'on augmente la taille critique (taille minimale pour l'efficacité des BLAS) la taille des paquets augmente, entraînant ainsi une diminution du nombre total de paquets.

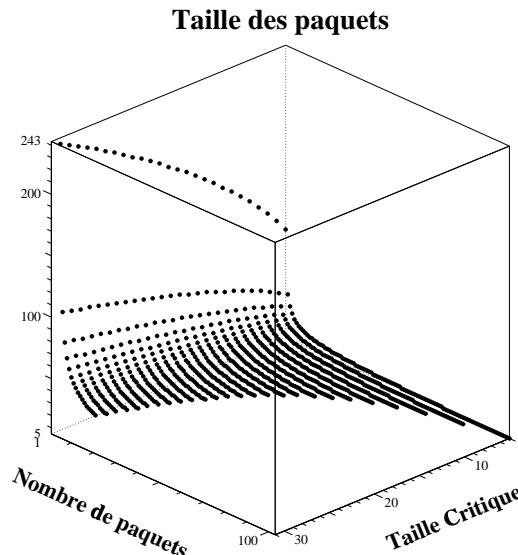


FIG. 4.3: Suite théorique de taille de paquets.

### 4.3 Validations expérimentales

L'implémentation utilise la bibliothèque OPIUM<sup>1</sup> [31], dans laquelle a été intégrée une primitive calculant la suite de taille de paquets pour la factorisation de Cholesky, et la bibliothèque LOCCS<sup>2</sup> [32], modifiée pour prendre en compte un

<sup>1</sup>Optimal Packet sIze compUtation Methods.

<sup>2</sup>Low Overhead Communication and Computation Subroutines.

découpage non constant du macro-pipeline. Les mesures de performances sont réalisées sur le calculateur SP2 IBM du LaBRI avec les paramètres (en secondes) :  $\beta = 1.4 \cdot 10^{-4}$ ,  $\tau = 4.6 \cdot 10^{-8}$ ,  $\tau_x = \tau_{\div} = 3.8 \cdot 10^{-7}$ . Le calcul est traité en *double précision* sur 16 nœuds fins 66MHz Power2. Nous nous restreignons dans cette étude expérimentale à l'utilisation du noyau BLAS 1.

Les figures 4.4 et 4.5 (sans BLAS 1) montrent que l'influence du recouvrement devient significative dès que la masse de calcul à traiter par processeur est suffisante (pour de petits macro-pipelines, le nombre de paquets générés est trop faible pour avoir un recouvrement efficace).

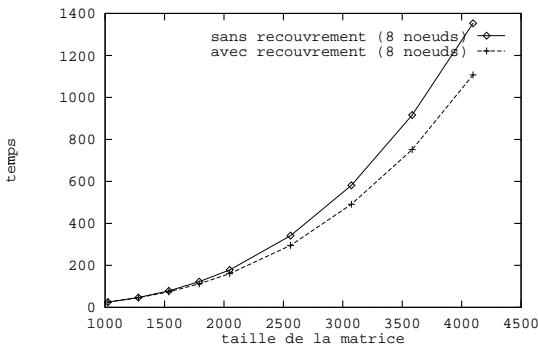


FIG. 4.4: Recouvrement sans BLAS 1.

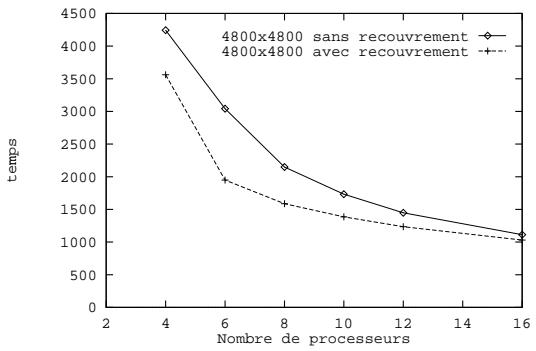


FIG. 4.5: Extensibilité sans BLAS 1.

Les tests menés avec les optimisations BLAS 1 conduisent à des résultats similaires, mais l'efficacité du recouvrement est mise en évidence pour des matrices plus grandes (figures 4.6 et 4.7). Expérimentalement nous retrouvons la taille critique attendue pour un BLAS 1 (de l'ordre de 150 éléments).

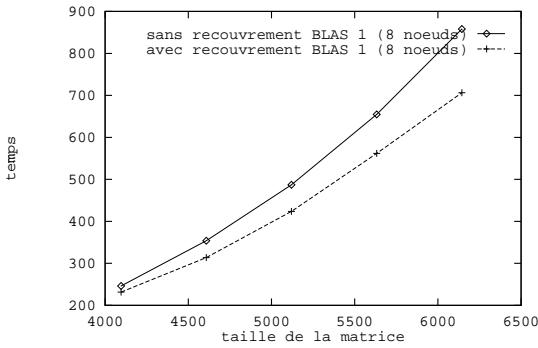


FIG. 4.6: Recouvrement avec BLAS 1.

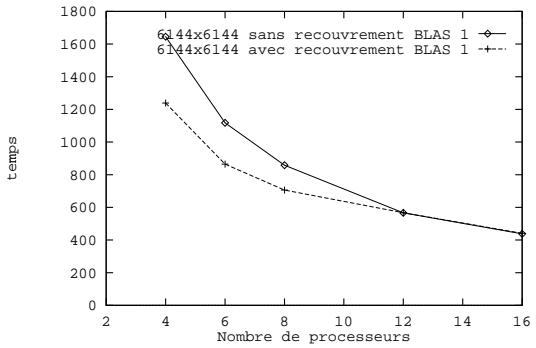


FIG. 4.7: Extensibilité avec BLAS 1.

## 4.4 Conclusion et perspectives

Dans ce chapitre, nous avons présenté et validé expérimentalement une méthode pour recouvrir les communications dans le cas de l'exemple de la factorisation de Cholesky pour les matrices symétrique denses. Les résultats obtenus montrent une relativement bonne efficacité de la méthode qui passe par le calcul d'une suite de taille décroissante de paquets qui optimise le recouvrement.

La démarche générale reste valable dès lors que l'on peut exprimer analytiquement les variations des calculs au cours du temps.

Une première évolution de ce travail consistera à prendre en compte une modélisation fine des opérateurs BLAS 3, pour utiliser une factorisation par blocs plus performante.

D'autre part, il est admis qu'une distribution bloc cyclique 2D offre de bonnes performances et une meilleur scalabilité pour des matrices denses. Une seconde évolution de ce travail consistera donc à calculer une partition irrégulière engendrant une distribution 2D d'une matrice symétrique pour laquelle on pourra construire un ordonnancement pour optimiser les recouvrements lors de la factorisation de Cholesky. L'objectif visé sera d'utiliser alors cette technique sur les calculs des blocs denses d'une matrice creuse structurée par blocs. (voir section 7.5).

# Bibliographie

- [1] Tarek S. Abdelrahman. Latency Hiding on COMA Multiprocessors. *The Journal of Supercomputing*, 10(3) :225–242, 1996.
- [2] R. Agarwal, F. Gustavson, and M. Zubair. A High Performance Matrix Multiplication Algorithm on a Distributed-Memory Parallel Computer. *IBM Journal of Research and Development*, 38(6) :673–681, Nov. 1994.
- [3] P. Amestoy, F. Despres, P. Ramet, and J. Roman. Optimisation des communications et régulation de charge pour la résolution par méthode directe de grands systèmes linéaires creux. In *ICaRE'97*, pages 467–488. Groupes thématiques CAPA-RUMEUR-EXEC du GDR PRS, Aussois, 1997.
- [4] R. Andonov and N. Yanev. *n*-Dimensional Orthogonal Tiling. Technical Report LIMAV-RR 96-6, Université de Valenciennes, LIMAV, September 1996.
- [5] Rumen Andonov, Hafid Bourzoufi, and Sanjay Rajopadhye. Two-Dimensional Orthogonal Tiling : From Theory to Practice. In *International Conference on High Performance Computing (HiPC)*, pages 225–231, Trivandrum, India, 1996.
- [6] Rumen Andonov, Nicola Yanev, and Hafid Bourzoufi. Three-Dimensional Orthogonal Tile Sizing Problem : Mathematical Programming Approach. Technical Report LIMAV-RR 96-3, Université de Valenciennes, LIMAV, May 1996.
- [7] S. B. Baden and S. J. Fink. Communication Overlap in Multi-tier Parallel Algorithms. In *Proc. of SC '98*, Orlando, Florida, Nov. 1998.
- [8] V. Bala, S. Kipnis, L. Rudolph, and M. Snir. Designing Efficient, Scalable, and Portable Collective Communication Libraries. In R.F. Sincovec, D.E. Keyes, M.R. Leuze, L.R. Petzold, and D.A. Reed, editors, *Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 862–872. SIAM, 1993.
- [9] Karthik Balasubramanian and David K. Lowenthal. Efficient Support for Pipelining in Distributed Shared Memory Systems. Submitted to Parallel and Distributed Computing Practices.
- [10] Prithviraj Banerjee, John A. Chandy, Manish Gupta, John G. Holm, Antonio Lain, Daniel J. Palermo, Shankar Ramaswamy, and Ernesto Su. The PARADIGM Compiler for Distributed-Memory Message Passing Multicomputers. In

- The First International Workshop on Parallel Processing*, pages 322–330, Bangalore, India, December 1994.
- [11] R. Bianchini, L. Kontothanassis, R. Pinto, M. De Maria, M. Abud, and C. L. Amorim. Hiding Communication Latency and Coherence Overhead in Software DSMs. In *Proceedings of the 7th ACM/IEEE International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 7)*, October 1996.
  - [12] T. Brandes and F. Desprez. Implementing Pipelined Computation and Communication in an HPF Compiler. In *Europar'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Science*, pages 459–462. Springer Verlag, August 1996.
  - [13] R. Brent. *Algorithms for Minimization without Derivatives*, chapter 3.4. Englewood Cliffs, N.J. : Prentice-Hall, 1973.
  - [14] Pierre-Yves Calland, Jack Dongarra, and Yves Robert. Tiling With Limited Resources. Technical Report UT-CS-97-350, CS Dept, The University of Tennessee, February 1997.
  - [15] C. Calvin and F. Desprez. Minimizing Communication Overhead Using Pipelining for Multi-Dimensional FFT on Distributed Memory Machines. In D.J. Evans, H. Liddell J.R. Joubert, and D. Trystram, editors, *Parallel Computing'93*, pages 65–72. Elsevier Science Publishers B.V. (North-Holland), 1993.
  - [16] R. D. Chamberlain and M. A. Franklin. Performance Effects of Synchronization in Parallel Processors. In *Symposium on Parallel and Distributed Processing (SPDP '93)*, pages 611–616, Los Alamitos, Ca., USA, December 1993. IEEE Computer Society Press.
  - [17] Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, , and Thorsten von Eicken. Low-Latency Communication on the IBM RISC System/6000 SP. In *ACM/IEEE Supercomputing '96*, Pittsburgh, PA, November 1996.
  - [18] Frédérique Chaussumier, Frédéric Desprez, and Michel Loi. Efficient Load-Balancing and Communication Overlap in Parallel Shear-Warp Algorithm on a Cluster of PCs. In P. Amestoy, P. Berger, M. Daydé, I. Duff, V. Frayssé, L. Giraud, and D. Ruiz, editors, *Proceedings of EuroPAR'99*, number 1685 in Lecture Notes in Computer Science, pages 570–577, Toulouse, 1999. Springer Verlag.
  - [19] Frédérique Chaussumier, Frédéric Desprez, and Loïc Prylli. Asynchronous Communications in MPI – the BIP/Myrinet Approach. In J. Dongarra, E. Luque, and Tomas Margalef., editors, *Proceedings of the EuroPVM/MPI'99 conference*, number 1697 in Lecture Notes in Computer Science, pages 485–492, Barcelona, Spain, September 1999. Springer Verlag.

- [20] C.-L. Chiang, J.-J. Wu, and N.-W. Lin. Toward Supporting Data Parallel Programming on Clusters of Symmetric Multiprocessors. In *Proc. of the 1998 International Conference on Parallel and Distributed Systems*, pages 607–614, Tainan, Taiwan, 1998. IEEE Computer Society Press.
- [21] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. LAPACK Working Note : ScaLAPACK : A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performances. Technical Report 95, Department of Computer Science - University of Tennessee, 1995.
- [22] M.J. Clement and M.J. Quinn. Overlapping Computations, Communications and I/O in Parallel Sorting. *Journal of Parallel and Distributed Computing*, 28 :162–172, 1995.
- [23] D.E. Culler, J. Pal Singh, and A. Gupta. *Parallel Computer Architecture : A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1998.
- [24] L. Diaz de Cerio, M. Valero-Garcia, and A. Gonzalez. Overlapping Communication and Computation in Hypercubes. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Second International EuroPar Conference*, number 1123 in Lecture Notes in Computer Science, pages 253–257, Lyon, France, August 1996. Springer Verlag.
- [25] F. Desprez. A Library for Coarse Grain Macro-Pipelining in Distributed Memory Architectures. In *IFIP 10.3 Conference on Programming Environments for Massively Parallel Distributed Systems*, pages 365–371. Birkhaeuser Verlag AG, Basel, Switzerland, 1994.
- [26] F. Desprez. *Procédures de Base pour le Calcul Scientifique sur Machines Parallèles à Mémoire Distribuée*. PhD thesis, Institut National Polytechnique de Grenoble, January 1994. LIP ENS-Lyon.
- [27] F. Desprez, S. Domas, and B. Tourancheau. Optimization of the ScaLAPACK LU Factorization Routine Using Communication/Computation Overlap. In *Europar'96 Parallel Processing*, volume 1124 of *Lecture Notes in Computer Science*, pages 3–10. Springer Verlag, August 1996.
- [28] F. Desprez, J.J. Dongarra, and B. Tourancheau. Performance Study of LU Factorization with Low Communication Overhead on Multiprocessors. *Parallel Processing Letters*, 5(2) :157–169, 1995.
- [29] F. Desprez and M. Garbey. Parallel Computing of a Combustion Front. In *Parallel CFD'93 - Implementations and Results Using Parallel Computers*. Elsevier Science Publishers B.V., 1993.

- [30] F. Despres and M. Pourzandi. A Comparison of Three Matrix Product Algorithms on the Intel Paragon and Archipel Volvox Machines. In Bl. Sendov I.T.Dimov and P.S.Vassilevski, editors, *Advances in Numerical Methods and Applications NM&A - O(h<sup>3</sup>)'94*, pages 234–244. World Scientific, 1994.
- [31] F. Despres, P. Ramet, and J. Roman. Optimal Grain Size Computation for Pipelined Algorithms. In *Europar'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Science*, pages 165–172. Springer Verlag, August 1996.
- [32] F. Despres and B. Tourancheau. LOCCS : Low Overhead Communication and Computation Subroutines. *Future Generation Computer Systems*, 10(2&3) :279–284, June 1994.
- [33] F. Despres and J. Zory. Performance des Macro-Pipelines dans les Programmes Data-Parallèles. In A. Schiper and D. Trystram, editors, *Actes des 9es Rencontres Francophones du Parallelisme RenPar'9*, pages 17–20, Lausanne, Switzerland, May 1997.
- [34] Stéphane Domas. *Contribution à l'écriture et à l'extension d'une bibliothèque d'algèbre linéaire parallèle*. PhD thesis, ENS Lyon, 1997.
- [35] T. Von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Message : a Mechanism for Integrated Communication and Computation. In ACM IEEE Computer Society, editor, *The 19th Annual Symposium on Computer Architecture*, pages 256–266. ACM Press, May 1992.
- [36] E.W. Evans, S.P. Johnson, P.F. Leggett, and M. Cross. Overlapped Communications Automatically Generated in a Parallelisation Tool. In B. Hertzberger and P. Sloot, editors, *Proceedings of High Performance Computing and Networking (HPCN'97)*, number 1225 in Lecture Notes in Computer Science, pages 801–810, Vienna, Austria, April 1997. Springer Verlag.
- [37] Stephen J. Fink and Scott Baden. Non-Uniform Partitioning for Finite Difference Methods Running on SMP Clusters.
- [38] M. Gupta and P. Banerjee. Compile-Time Estimation of Communication Costs on Multicomputers. In *Proc. of Intl. Parallel Processing Symposium*, pages 470–475, 1993.
- [39] S. Hiranandani, K. Kennedy, and C. Tseng. Evaluating Compiler Optimizations for Fortran D. *Journal of Parallel and Distributed Computing*, 21 :27–45, 1994.
- [40] J.B. White III and S.W. Boya. Where's Overlap ? An Analysis of Popular MPI Implementations. In A. Skjeellum, P.V. Bangalore, and Y.S. Dandass, editors, *Proceedings of the Third MPI Developer's and User's Conference*, pages 1–6, Atlanda, Georgia, 1999. MPI Software Technology Press.

- [41] François Irigoin and Remy Triolet. Supernode partitioning. In *Proc. 15th Annual ACM Symp. Principles of Programming Languages*, pages 319–329, San Diego, CA, January 1988.
- [42] K.K. Kee and S. Hariri. Efficient Communication Algorithms for Pipeline Multicomputers. In *Proceedings of Supercomputing'94*, pages 468–477, Washington, D.C., Nov. 1994. IEEE Comp. Soc. Press.
- [43] C.-T. King, W.-W. Chou, and L.M. Ni. Pipelined Data-Parallel Algorithms : Part I – Concept and Modeling. *IEEE Trans. on Parallel and Distributed Systems*, 1(4) :470–485, 1990.
- [44] C.-T. King, W.-W. Chou, and L.M. Ni. Pipelined Data-Parallel Algorithms : Part II – Design. *IEEE Trans. on Parallel and Distributed Systems*, 1(4) :486–499, 1990.
- [45] C.-T. King, W.H. Chou, and L.M. Ni. Pipelined Data Parallel Algorithms - Concept and Modeling. In *International Conference on Supercomputing*, pages 385–395, July 1988.
- [46] A. Lain and P. Banerjee. Techniques to Overlap Computation and Communication in Irregular Iterative Applications. In *Supercomputing '94 : International conference*, pages 236–245, Manchester, UK, 1994. Univ of Manchester.
- [47] L.Colombet, P.Michallon, and D.Trystram. Parallel Matrix-Vector Product on Rings with a Minimum of Communications. In *Parallel Computing*, number 22 in Elsevier Science, pages 289–310, 1996.
- [48] B.-H. Lim and R. Bianchini. Limits on the Performance Benefits of Multithreading and Prefetching. Technical Report RC 20238, IBM T. J. Watson Research Center, 1995.
- [49] David K. Lowenthal. Accurately Selecting Block Size at Run Time in Pipelined Parallel Programs. Submitted to the International Journal of Parallel Programming.
- [50] David K. Lowenthal and Michael James. Run-Time Selection of Block Size in Pipelined Parallel Programs. In *Proceedings of the 2nd Merged IPPS/SPDP Conference*, pages 82–87, April 1999.
- [51] B. Tourancheau M. Pourzandi. Recouvrement Calcul/Communication dans l'Elimination de Gauss sur un iPSC/860. Technical report, Ecole Normale Supérieure de Lyon, 1992.
- [52] Kevin McManus, Steve Johnson, and Mark Cross. Communication Latency Hiding in a Parallel Conjugate Gradient Method. In *Proc. Domain Decomposition 11*, Greenwich, July 1998.

- [53] E. Montagne, M. Ruloz, R. Suros, and F. Breant. Modeling Optimal Granularity when Adapting Systolic Algorithms to Transputer Based Supercomputers. *Parallel Computing*, 20 :807–814, 1994.
- [54] Tung Nguyen, Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. Asynchronous Dynamic Load Balancing of Tiles. In *Proc. of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, March 1999.
- [55] C.N. Ojeda-Guerra and A. Suárez. Solving Linear Systems of Equations Overlapping Communications and Computations in Torus Networks. In *Fifth Euro-Micro Workshop on Parallel and Distributed Processing (PDP'97)*, pages 453–460. IEEE Computer Society Press, January 1997.
- [56] Daniel J. Palermo and Prithviraj Banerjee. Automatic selection of Dynamic Data Partitioning Schemes for Distributed-Memory Multicomputers. In *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing*, Columbus, OH., Aug. 1995.
- [57] Daniel J. Palermo and Prithviraj Banerjee. Automatic selection of Dynamic Data Partitioning Schemes for Distributed-Memory Multicomputers. Technical Report CRHC-95-09, University of Illinois at Urbana-Champaign, April 1995.
- [58] D.J. Palermo. *Compiler Techniques for Optimizing Communications and Data-distribution for Distributed Memory Multicomputers*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.
- [59] D. Park. Adaptive Granularity : Transparent Integration of Fine-Grain and Coarse Grain Communications. Technical report, Computer Science Departement, University of Southern California, 1996.
- [60] Loic Prylli, Bernard Tourancheau, and Roland Westrelin. Modeling of a high speed network to maximize throughput performance : the experience of BIP over Myrinet. In H.R. Arabnia, editor, *Parallel and Distributed Processing Techniques and Applications (PDPTA '98)*, volume II, pages 341–349, Las Vegas, USA, 1998. CSREA Press.
- [61] Michael J. Quinn and Philip J. Hatcher. On the Utility of Communication-Computation Overlap in Data-Parallel Programs. *Journal of Parallel and Distributed Computing*, 33(02) :197, 1996.
- [62] J. Ramanujam and P. Sadayappan. Tiling of Iteration Spaces for Multicomputers. In *Proc. Internal Conference on Parallel Processing*, volume 2, pages 179–186, August 1990.
- [63] Pierre Ramet. Calcul de la suite optimale de taille de paquets pour la factorisation de Cholesky. In *ACTES RenPar'9*, pages 111–114, 1997.

- [64] D.F. Robinson, X.-He Sun, and R.J. Enbody. A Pipelined Parallel Approach to Solving Dense Linear Systems. In *Proc. of the fourth Conf. on Hypercubes, Concurrent Computers and Applications*, pages 441–444, Monterey, CA, 1989.
- [65] Y. Saad. Communication Complexity of the Gaussian Elimination Algorithm on Multiprocessors. *Linear Algebra and Applications*, 77 :315–340, 1986.
- [66] B. Siegell and P. Steenkiste. Automatic selection of load balancing parameters using compiletime and run-time information. *Concurrency - Practice and Experience*, 9(3) :275–317, 1996.
- [67] Bruce S. Siegell. *Automatic generation of parallel programs with dynamic load balancing for a network of workstations*. PhD thesis, Carnegie Mellon Univ. - Sch. of Computer Science, May 1995.
- [68] A. Sohn and R. Biswas. Communication Studies of DMP and SMP Machines. Technical Report NAS-97-004, NAS, 1997.
- [69] A. Sohn, J. Ku, Y. Kodama, M. Sato, H. Sakane, H. Yamana, S. Sakai, and Y. Yamaguchi. Identifying the Capability of Overlapping Computation with Communication. In *Proc. of PACT'96*, pages 133–138, Boston, Mass., Oct 1996. North-Holland Pub. Co.
- [70] Y. Tanaka, M. Matsuda, K. Kubota, and M. Sato. Performance Improvement by Overlapping Computation and Communication on SMP Clusters. In *Proc. of Int'l Conf. PDPTA '98*, volume 1, pages 275–282, 1998.
- [71] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Rice University, January 1993.
- [72] R.F. van der Wijngaart, S.R. Sarukkai, and P. Mehra. The Effects of Interrupts on Software Pipeline Execution on Message-passing Architectures. In *Proc. of the 1996 Int. Conf. on Supercomputing (FCRC'96)*, pages 189–196, Philadelphia, Penn., May 1996. ACM SIGARCH.
- [73] A. Wakatani and M. Wolfe. A New Approach to Array Redistribution : Strip-Mining Redistribution. In *Proceedings of Parallel Architectures and Languages Europe (PARLE 94)*, pages 323–335, July 1994.
- [74] A. Wakatani and M. Wolfe. Effectiveness of Message Strip-Mining for Regular and Irregular Communication. In *Intl. Conference on Parallel and Distributed Computing Systems*, Oct. 1994.
- [75] D. W. Walker. Portable Programming within a Message-Passing Model : the FFT as an Example. In *The Third Conference On Hypercube Concurrent Computers and Applications*, volume II - Applications. Geoffrey Fox California Institute of Technology, 1988.

- [76] Randolph Y. Wang, Arvind Krishnamurthy, Rich P. Martin, Thomas E. Anderson, and David E. Culler. Towards a Theory of Optimal Communications Pipelines. Technical Report CSD-98-981, University of California, Berkeley, January 26, 1998.
- [77] Michael Wolfe. Iteration Space Tiling for Memory Hierarchies. In Gary Rodrigue, editor, *Proceedings of the 3rd Conference on Parallel Processing for Scientific Computing*, pages 357–361, Philadelphia, PA, USA, December 1989. SIAM Publishers.

## Deuxième partie

# Solveurs Parallèles Directs pour Matrices Symétriques Définies Positives Creuses



# Chapitre 5

## Introduction et positionnement de l'étude

La résolution de systèmes linéaires creux est une brique de base pour traiter numériquement de nombreux problèmes de calcul scientifique. Ces systèmes, qui apparaissent en particulier dans le cadre de la discréétisation d'équations aux dérivées partielles par éléments ou volumes finis, sont de très grande taille pour les applications en vraie grandeur. Le parallélisme est alors une technique incontournable pour résoudre ces très grands systèmes. Dans ce contexte, les *méthodes directes* [23, 31, 40] jouent un rôle très important car elles sont générales et surtout robustes numériquement ; il existe en effet de nombreux cas où une factorisation numérique s'avère être le seul recours pour une résolution raisonnable du système. On trouvera dans [23], et dans les références incluses, un état de l'art complet des problèmes rencontrés lors de la conception de solveurs parallèles directs performants. Cette introduction constitue une présentation rapide de ces problèmes.

Pour obtenir de bonnes performances globales, il est obligatoire d'utiliser systématiquement des *noyaux de calcul performants* implémentés au plus bas niveau pour exploiter au mieux les effets de caches. C'est le cas des primitives de calcul BLAS avec un maximum d'efficacité pour celles de niveau 3 dans le cadre de calculs matriciels denses par bloc. En effet, outre l'efficacité des noyaux de calcul pour l'utilisation des unités flottantes pipelinées des architectures processeurs super-scalaires, l'optimisation de la localité des accès mémoires est primordiale. Ainsi, si on considère le taux de ré-utilisabilité des données  $\tau = \#(\text{flops}) / \#(\text{accès mémoire})$  on a :

- $\tau = 2/3$  pour une opération BLAS1 (pour un SCAL de la forme  $y := y + \alpha \cdot x$ ),
- $\tau = 2$  pour une opération BLAS2 (pour un GEMV de la forme  $y := A \cdot x$ ),
- $\tau = n/2$  pour une opération BLAS3 (pour un GEMM de la forme  $C := A \cdot B$ ).

Il faut noter cependant qu'il est important de bien gérer la taille des blocs pour atteindre des vitesses de calcul maximales.

Cet état de fait a un impact considérable sur l'algorithme des solveurs [45, 46, 63, 73–75], sur la nature des données et le prétraitement à réaliser sur ces données, et enfin sur l'étude des problèmes liés à la gestion du creux des matrices car tout doit être pensé en termes de blocs : les structures de données seront par bloc et les calculs se feront par bloc.

Un des points essentiels propre aux méthodes directes sur les matrices creuses est le problème du *remplissage* [40]. En effet, la matrice factorisée de Cholesky  $L$  est plus pleine que la matrice initiale  $A$  de part la création de nouveaux termes non nuls au cours du processus de factorisation. Un outil fondamental pour étudier ce remplissage est le *modèle de graphe* [40] associé à l'élimination de Gauss. Celui-ci 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* [4, 40, 64, 65] des sommets du graphe associé à la matrice initiale qui minimise le remplissage et en même temps le nombre d'opérations à faire effectivement pour factoriser.

Le graphe non orienté  $G$  associé à une matrice  $A$   $n \times n$  symétrique ou à structure symétrique a  $n$  sommets et 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 correspondant aux termes de remplissage. L'*arbre d'élimination* [58]  $T$  associé à la matrice factorisée est un arbre, 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 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 *large et de faible hauteur*. Comme on le verra par la suite, les renumérotations performantes selon ces critères sont celles basées sur les techniques de “degré minimum” et de “dissections emboîtées” ; à noter que la numérotation classique de type Cuthill-McKee est à écarter car elle occasionne plus de remplissage et conduit en plus à des arbres d'élimination très longs avec donc une faible indépendance dans les calculs.

Une autre étape fondamentale est celle dite de *factorisation symbolique par bloc* [11] qui va permettre de calculer algorithmiquement la structure creuse par bloc de la matrice  $L$  à partir de celle de  $A$ . Le but est d'optimiser la *factorisation numérique* [6, 7, 9] en évitant 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 *bloc-colonne* de la matrice, le *graphe quotient*  $G^*/P$  à partir de  $G$  et cette partition  $P$ . L'intérêt de l'algorithme de factorisation symbolique par bloc 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 que l'on a l'égalité  $(G/P)^* = G^*/P$ . La complexité [11] 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 5.1) et dépend donc de la qualité de la numérotation vis à vis de la conservation du creux 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 bloc* et sa structure va décrire maintenant les dépendances dans les calculs par bloc du solveur.

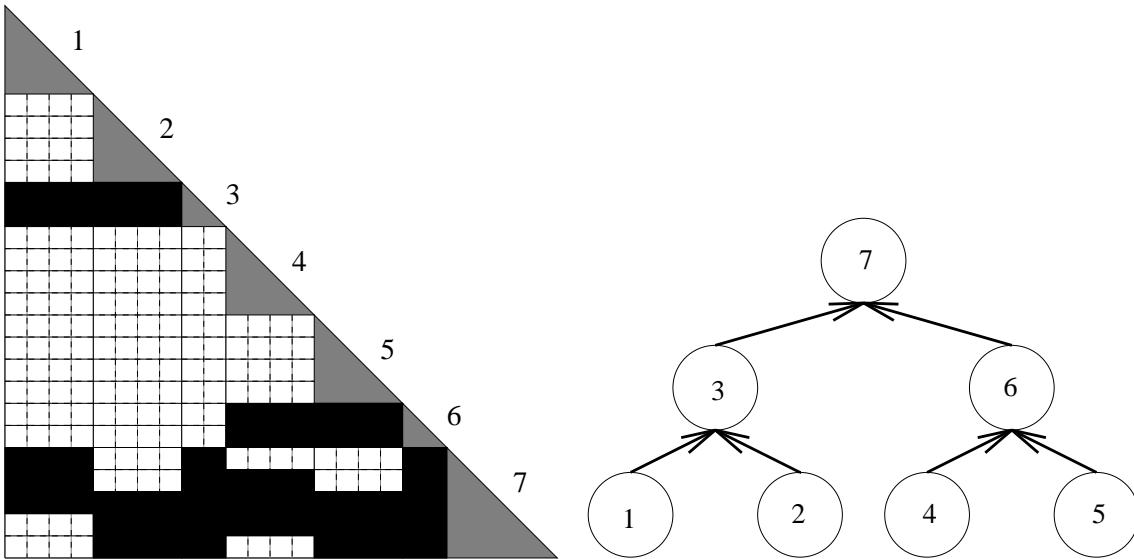


FIG. 5.1: Un exemple de matrice factorisée  $L$  structurée par bloc-colonne et d'arbre d'élimination par bloc. Chaque bloc-colonne est constitué d'un bloc diagonal et de plusieurs blocs extra-diagonaux denses.

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

Suivant la nature de la matrice  $A$ , on distingue classiquement plusieurs types de factorisation :

- pour une matrice  $A$  symétrique définie positive, on peut utiliser la factorisation de Cholesky ( $A = LL^t$ ) ou de Cholesky-Crout ( $A = LDL^t$ ), avec ou sans pivotage numérique symétrique ;
- pour une matrice  $A$  non symétrique, on utilisera la factorisation LU avec pivotage non symétrique.

Le cadre parallèle retenu pour l'instant dans ce travail a été celui des machines MIMD à mémoire distribuée, avec un modèle de programmation de type SPMD et utilisant une bibliothèque de communication de type MPI.

En fonction des propriétés de  $A$ , de la nature du solveur numérique à retenir (mais aussi du degré d'hétérogénéité de la plateforme d'exécution parallèle), on est amené à utiliser un ordonnancement statique (à précalculer aussi précisément que possible) et/ou dynamique des calculs par blocs suivant la capacité que l'on a à prédire à l'avance le comportement de ces calculs. Ce travail s'inscrit dans le cadre de la factorisation  $LDL^t$  de matrices symétriques définies positives *sans pivotage* implémentée sur des machines parallèles *homogènes*; on se place donc dans une situation où il est intéressant de rechercher un *ordonnancement statique performant* [36, 53, 67].

Une autre phase de prétraitement des données va donc précéder l'exécution parallèle du solveur numérique. Ce prétraitement a pour rôle de calculer algorithmiquement un bon partitionnement et une bonne distribution des données ; ceci se fera à partir de la factorisation symbolique et par bloc. Le prétraitement va donc, en plus de la phase de renumérotation, calculer de manière statique une *régulation équilibrée* pour le solveur (critère d'équilibrage de charge et prise en compte des contraintes de précédence) ; on utilisera pour cela la structure de l'arbre d'élimination par bloc. Le but sera d'exploiter les différents niveaux de parallélisme 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. On parlera aussi de parallélisme induit par le creux ;
2. le parallélisme à grain moyen, dû à la possibilité de raffiner la partition des inconnues en découplant un nœud 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 partie du parallélisme interne d'un processeur lors du calcul sur les blocs (utilisation optimale du pipeline du processeur). Ce dernier niveau de parallélisme est assuré, modulo une bonne taille de blocs, par l'utilisation des primitives BLAS3.

A noter que l'on peut aussi optimiser la localité des communications pour éviter des contentions dans le réseau du calculateur.

A partir de la structure de données par bloc calculée par la factorisation symbolique [11], la phase de prétraitement va donc calculer une bonne distribution des données sur les processeurs. On utilisera l'arbre d'élimination pour distribuer de manière équilibrée les blocs indépendants qui sont les plus bas dans l'arbre, puis on découpera les blocs situés dans les niveaux les plus hauts qui représentent de manière inhérente la plus grande masse de calcul. Les sous blocs seront alors distribués de manière bloc cyclique pour utiliser le parallélisme contenu dans les calculs denses. On construit ainsi de manière statique une bonne régulation des calculs [30, 36, 49, 71].

Cette technique a été utilisée selon un schéma de distribution 1D où l'on distribue de manière élémentaire des blocs-colonnes entiers. Schreiber [76] a montré que ce type de schéma limite la scalabilité des solveurs et qu'il faut alors avoir un schéma de distribution 2D où l'on distribue de manière élémentaire les blocs diagonaux et extra-diagonaux. C'est dans ce dernier cadre que l'on a atteint les meilleures performances connues à ce jour.

Concernant l'algorithme des solveurs, il y a essentiellement deux types d'approche qui sont l'approche *supernodale*, qui provient directement des algorithmes d'élimination par colonne réécrits par blocs et avec les variantes que l'on appelle *fan-in* (voir figure 5.3) ou *fan-out* (voir figure 5.2) [6–9, 48, 63], et l'approche *multifrontale* (voir figure 5.4) [5, 16, 26, 27, 45, 48]. En ce qui concerne les méthodes supernodales qui constituent les méthodes retenues dans ce travail, l'approche fan-in ou “left-looking” consiste à répercuter sur le bloc-colonne courant les modifications des blocs-colonnes de gauche concernés (l'opération  $cmod(j, k)$  répercute les modifications du bloc-colonne  $j$  par le bloc-colonne  $k$ ), puis à calculer la valeur finale du bloc-colonne courant (l'opération  $cdiv(j)$  calcule la mise à jour du bloc-colonne  $j$ ) ; de manière symétrique, l'approche fan-out ou “right-looking” consiste à calculer définitivement le bloc-colonne courant et à répercuter ses contributions sur les blocs-colonnes de droite concernés. Dans un cadre parallèle, l'approche fan-in est plus performante car elle permet de limiter considérablement la masse de communication.

Dans les deux chapitres suivants de cette thèse, nous présentons notre stratégie d'ordonnancement statique pour un solveur parallèle supernodal de type fan-in, d'abord pour une distribution 1D [53] (chapitre 6) puis pour une distribution 2D [54] (chapitre 7), et nous donnons des résultats expérimentaux sur SP2 ; ces résultats montrent que le logiciel qui résulte de ces travaux est tout à fait compétitif avec la référence actuelle PSPASES développé par G. Karypis, V. Kumar et A. Gupta [55], et qui se place dans un cadre très proche des solutions que nous avons retenues.

Un des objectifs de cette étude est d'intégrer ces techniques et de les valider dans un cadre industriel (voir annexe B) ; pour cela nous avons utilisé un jeu de 33 problèmes tests, en grande partie issus d'applications industrielles. Ces travaux ont été menés en collaboration avec Pascal Hénon.

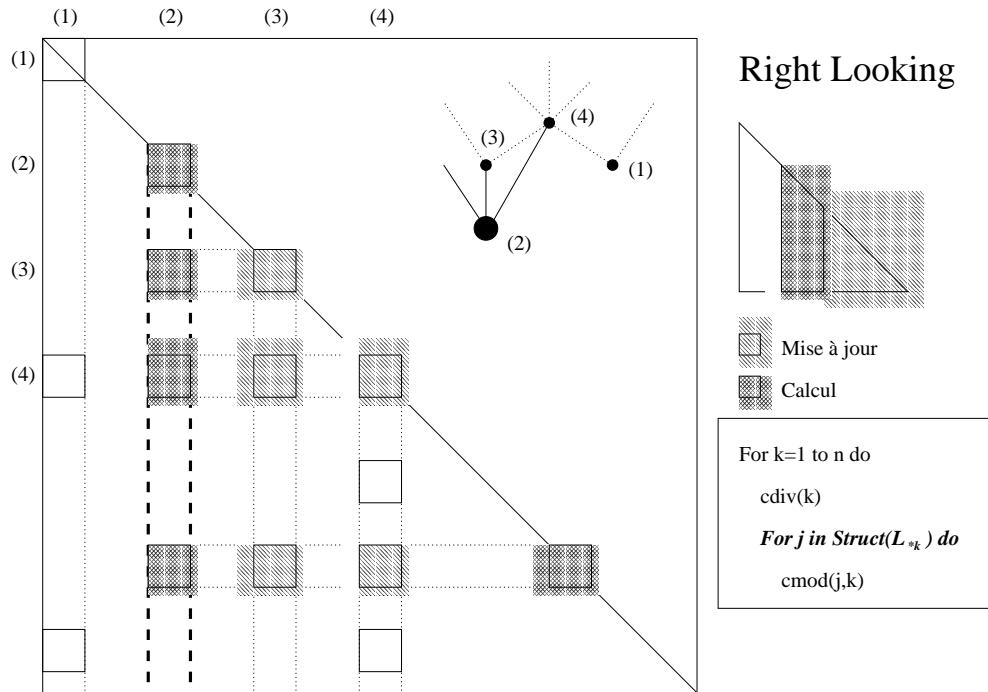


FIG. 5.2: Schéma Fan-Out :  $\text{Struct}(L_{*k}) := \{j > k \mid L_{jk} \neq 0\}$ .

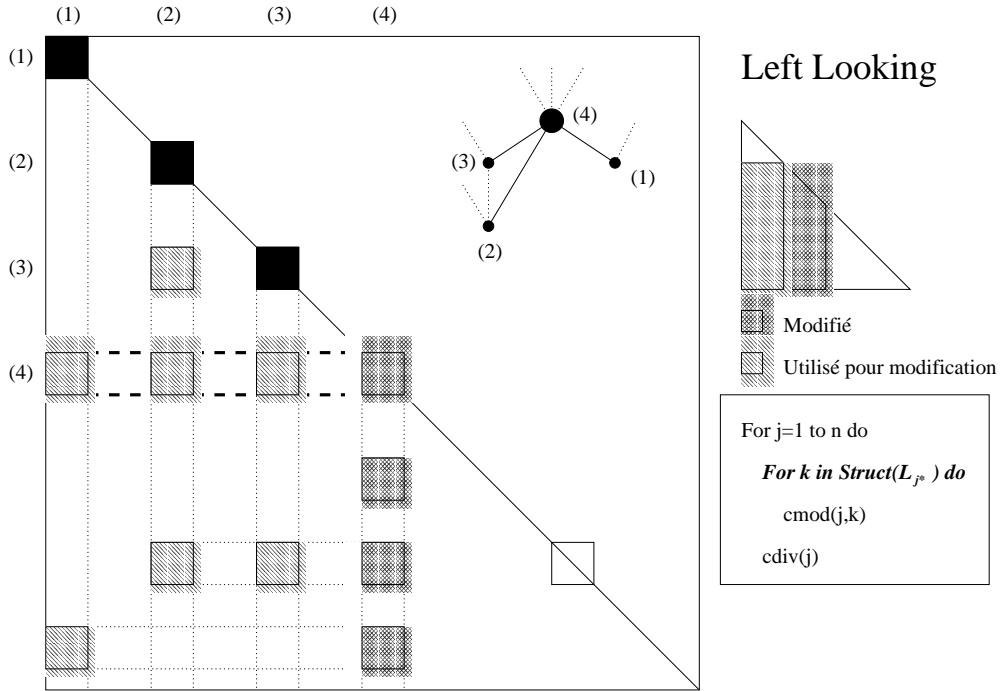


FIG. 5.3: Schéma Fan-In :  $Struct(L_{j*}) := \{k < j \mid L_{jk} \neq 0\}$ .

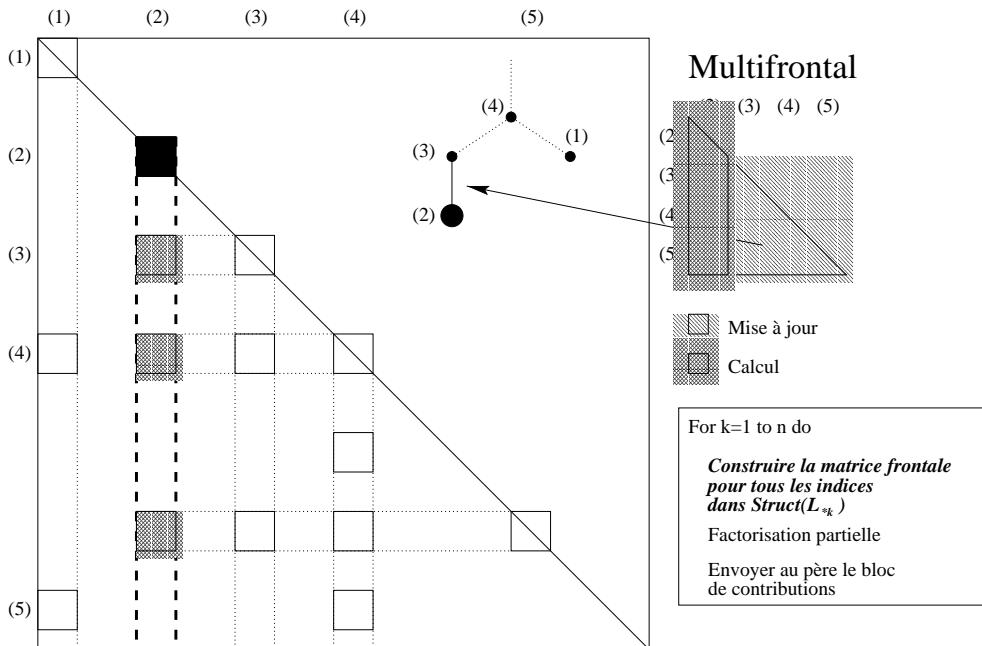


FIG. 5.4: Schéma Multifrontal :  $Struct(L_{*k}) := \{j > k \mid L_{jk} \neq 0\}$ .



# Chapitre 6

## Solveur $LDL^t$ avec distribution 1D

### 6.1 Introduction

Dans ce chapitre nous nous intéressons au problème du partitionnement et de la distribution par blocs pour l'algorithme de la factorisation  $LDL^t$  d'une matrice creuse sans pivotage. On se place dans le cadre d'une distribution statique pour une architecture parallèle de type MIMD à mémoire distribuée. Nous avons retenu la factorisation de type  $LDL^t$  afin de pouvoir traiter des problèmes en nombre complexes, mais les algorithmes présentés peuvent être étendus à d'autres types de factorisation. Nous allons présenter et analyser un algorithme général qui calcule un ordonnancement statique efficace [36, 67] des calculs par blocs pour un solveur parallèle basé sur une approche *supernodale fan-in* [7, 73–75]; ce solveur sera complètement piloté par cet ordonnancement. Pour cela nous aurons à prendre en compte de façon précise le coût de calcul des opérations de type BLAS 3, ainsi que le coût des communications et des agrégations locales imposées par la stratégie fan-in. Nous montrerons qu'avec cet algorithme de partitionnement et de distribution, et à partir d'une renommérotation des inconnues basée sur un couplage fin [65] entre les stratégies de type “dissections emboîtées” et de type “degré minimum approché”, notre solveur est à la fois efficace sur des problèmes réguliers comme les grilles d’éléments finis 2D ou 3D et sur des problèmes irréguliers.

Pour réaliser une factorisation parallèle efficace nous avons vu qu'il était nécessaire de considérer trois étapes de pré-traitement :

- L'étape de *renumérotation* qui calcule une permutation symétrique de la matrice initiale  $A$  pour que la factorisation exploite le maximum d'indépendance dans les calculs tout en minimisant le remplissage. Dans cette étude nous utilisons un

couplage fin entre les algorithmes de dissections emboîtées et de degré minimum approché [4, 64, 65] ; le partitionnement du graphe original est réalisé en fusionnant le partitionnement des séparateurs calculés par l'algorithme de dissections emboîtées et les super-variables amalgamées pour chaque sous-graphe renuméroté à l'aide du degré minimum approché en tenant compte des informations de *Halo*.

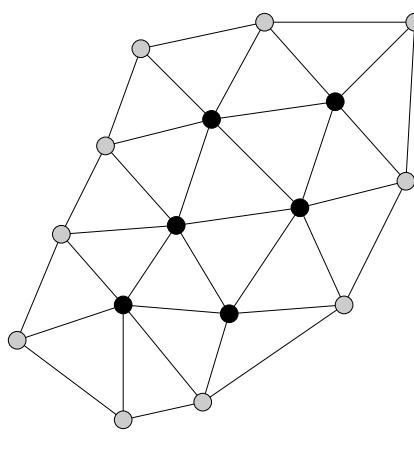


FIG. 6.1: Sans Halo.

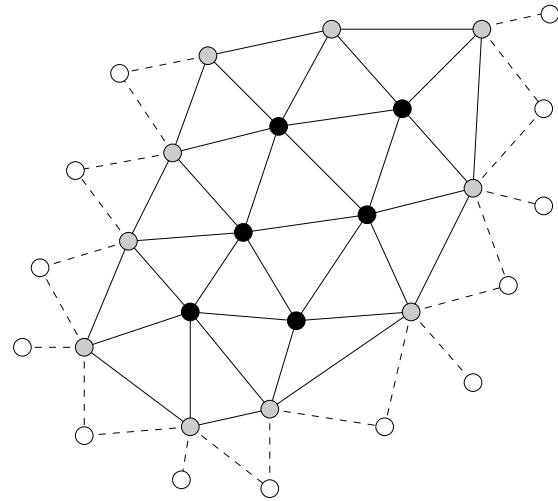


FIG. 6.2: Avec Halo.

Cette approche consiste à ajouter aux sous-graphes passés à l'algorithme de degré minimum une description topologique de leur voisinage immédiat (voir figures 6.1 et 6.2). Cette information supplémentaire, appelée *Halo*, est alors exploitée par un algorithme de degré minimum approché, modifié pour ne numérotter que les sommets appartenant au graphe.

- L'étape de *factorisation symbolique par blocs* qui détermine la structure de données par blocs de la matrice factorisée  $L$  résultant de la partition calculée à l'étape de renumérotation. Cette structure est constituée de  $N$  blocs-colonnes, contenant chacun un bloc diagonal symétrique plein et un ensemble de blocs extra-diagonaux rectangles pleins. Cette étape peut être réalisée avec une complexité quasi-linéaire en temps et en espace [11]. A partir de la structure par blocs de  $L$ , on peut construire le graphe quotient décrivant l'ensemble des dépendances entre blocs ainsi que l'arbre d'élimination supernodal.

- L'étape de *partitionnement et de distribution* qui raffine la partition issue de l'étape de factorisation symbolique en découplant et en distribuant les blocs suffisamment larges pour extraire du parallélisme induit par les calculs sur les blocs denses, puis qui affecte chaque bloc à un processeur dans l'architecture cible.

Dans la suite de ce chapitre nous présentons tout d'abord l'algorithme de factorisation parallèle d'une matrice creuse symétrique définie positive sur lequel nous nous appuyons pour décrire ensuite le partitionnement et la distribution par bloc. Dans ce chapitre nous considérons uniquement une distribution 1D des blocs-colonnes de la matrice, le chapitre suivant présentant l'extension à une distribution 2D. Nous présentons enfin nos expérimentations numériques sur un IBM SP2 pour une large classe de matrices creuses ainsi qu'une analyse des performances obtenues.

## 6.2 Factorisation parallèle et algorithme de distribution

On considère donc la structure de données par blocs de la matrice factorisée  $L$  calculée par la factorisation symbolique par blocs. Rappelons qu'un bloc-colonne est constitué d'un bloc diagonale plein et d'un ensemble de blocs extra-diagonaux pleins.

Nous définissons la fonction booléenne  $off\_diag(k, j)$ ,  $1 \leq k < j \leq N$  qui retourne *vrai* si et seulement s'il existe un bloc extra-diagonal dans le bloc-colonne  $k$  “en face” du bloc-colonne  $j$ . On peut alors construire les deux ensembles :

$$\begin{cases} BStruct(L_{k*}) & := \{i < k \mid off\_diag(i, k) = vrai\} \\ BStruct(L_{*k}) & := \{j > k \mid off\_diag(k, j) = vrai\}. \end{cases}$$

Ainsi,  $BStruct(L_{k*})$  est l'ensemble des blocs-colonnes qui modifient le bloc-colonne  $k$  et  $BStruct(L_{*k})$  est l'ensemble des blocs colonnes qui sont modifiés par le bloc-colonne  $k$  (voir figure 6.3).

On s'intéresse maintenant à l'algorithme supernodal de la factorisation parallèle creuse  $LDL^t$  où toutes les contributions distantes sont localement agrégées dans une structure bloc. Ce schéma est proche de l'algorithme fan-in [9] ; chaque processeur communique uniquement des blocs-colonnes de contributions agrégées. Ces structures, que nous noterons par la suite  $AUCB_k$  (*aggregate update column block*), peuvent être facilement construites grâce à l'étape de factorisation symbolique. Il est important de remarquer qu'un bloc-colonne  $k$  ne recevra que des contributions de la part des processeurs dans l'ensemble  $Procs(L_{k*}) := \{map(i) \mid i \in BStruct(L_{k*})\}$ , où l'opérateur  $map()$  correspond à la distribution des blocs-colonnes.

Le “pseudo-code” de la factorisation  $LDL^t$  peut-être exprimé en terme de calculs par blocs (voir figure 6.4) ; ces calculs sont réalisés, autant que possible, sur des ensembles de blocs compactés pour profiter au mieux des effets BLAS.

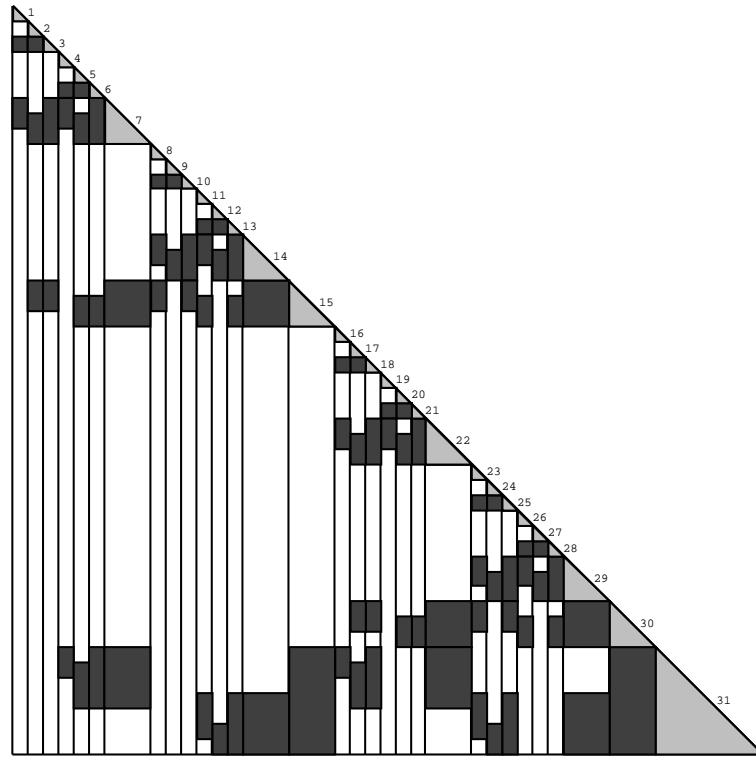


FIG. 6.3: Un exemple de matrice factorisée structurée par bloc-colonne. On a  $BStruct(L_{7*}) := \{1, 2, 3, 4, 5, 6\}$  et  $BStruct(L_{*7}) := \{15, 31\}$ .

Les synchronisations de l’algorithme sont assurées par l’utilisation des variables  $r_k$  qui comptent le nombre de contributions restant à prendre en compte pour le bloc-colonne  $k$ . Tant que  $r_k \neq 0$ , il reste des contributions à recevoir pour le bloc-colonne  $k$  (lignes 2–4).

Ensuite le bloc diagonal peut-être factorisé et les blocs extra-diagonaux sont mis à jour (lignes 6–8). Le calcul partiel ( $F$ ) est sauvegardé pour être réutilisé dans le calcul des contributions. Toutes les contributions associées au bloc-colonne  $k$  peuvent maintenant être calculées et ajoutées localement (ligne 12) ou agrégées (ligne 14).

Un bloc-colonne de contributions agrégées peut-être envoyé si toutes ses contributions locales ont été prises en compte (ligne 15). Pour limiter l’augmentation mémoire due au stockage des agrégations locales, ces structures sont allouées dynamiquement seulement et sont désallouées dès que leur émission est achevée. Dans le cas où la ressource mémoire est critique, il est également possible d’envoyer ces structures avec une agrégation partiellement achevée afin de libérer de l’espace mémoire, ce qui s’apparente à un schéma de type fan-both [6].

Remarquons que la complexité de chacune de ces étapes dépend uniquement de la taille des blocs et peut donc être évaluée précisément.

**Notations pour  $k$  fixé,  $1 \leq k \leq N$  :**

- $r_k$  : nombre de contributions restant à soustraire au bloc colonne  $k$ ,
- $r_k^o$  : valeur initiale de  $r_k$ ,
- $s_k$  : nombre de contributions restant à ajouter à  $AUCB_k$ ,
- $s_k^o$  : valeur initiale de  $s_k$ ,
- le symbole \* signifie  $\forall j \in BStruct(L_{*k})$ ,
- soit  $j \geq k$  ; la séquence  $[j]$  signifie  $\forall i \in BStruct(L_{*k}) \cup \{k\}$  avec  $i \geq j$  .

Pour le processeur p :

1. **For**  $k \in [1, N]$  tel que  $map(k) == p$  **Do**
2.     **While**  $r_k \neq 0$  **Do** % étape de réception/mise à jour
3.         Recevoir  $AUCB_j$ ;
4.          $A_{[j]j} = A_{[j]j} - AUCB_j$ ;    $r_j \leftarrow r_j - s_j^o$ ;
5.         % étape de calcul/émission
6.         Factoriser  $A_{kk}$  sous la forme  $L_{kk}D_kL_{kk}^t$ ;
7.         Résoudre  $L_{kk}F_*^t = A_{*k}^t$ ;
8.         Résoudre  $D_kL_{*k}^t = F_*^t$ ;
9.         **For**  $j \in BStruct(L_{*k})$  **Do**
10.             Calculer  $C = L_{[j]k}F_j^t$ ;
11.             **If**  $map(j) == p$  **Then**
12.                  $A_{[j]j} = A_{[j]j} - C$ ;    $r_j \leftarrow r_j - 1$ ;
13.             **Else** % étape d'agrégation
14.                  $AUCB_j = AUCB_j + C$ ;    $s_j \leftarrow s_j - 1$ ;
15.             **If**  $s_j = 0$  **Then** envoyer  $AUCB_j$  à  $map(j)$ ;

FIG. 6.4: Algorithme de la factorisation parallèle pour une distribution 1D.

Avant de pouvoir exécuter l'algorithme général présenté ci-dessus, il est nécessaire de réaliser une étape de repartitionnement et de distribution des blocs sur l'ensemble des processeurs. L'efficacité de la factorisation parallèle est complètement liée au choix d'un bon ordonnancement des calculs ; cette étape de repartitionnement et de distribution a donc pour but de calculer une régulation statique qui équilibre la charge sur les processeurs et qui respecte au mieux les contraintes de précédence dans les calculs imposées par l'algorithme de factorisation. Pour cela, on utilise l'arbre d'élimination construit lors de la factorisation symbolique par bloc, et on exploite les différents niveaux de parallélisme cités dans l'introduction au chapitre 5.

Les algorithmes de la littérature et qui sont de type “subtree to subcube” ou de type “proportional mapping” proposent un repartitionnement et une distribution qui optimisent l’équilibrage de charge en descendant récursivement l’arbre d’élimination. Étant données cette nouvelle partition et la distribution des blocs, les solveurs sont généralement responsables du bon ordonnancement des calculs et des communications. Ces approches existantes conduisent alors à deux types de problèmes :

- l’équilibrage de charge est approximatif car seul le nombre d’opérations est pris en compte. Nous avons vu que pour être efficace, l’algorithme du solveur est orienté par blocs afin d’utiliser les routines BLAS ; le problème connu est que le temps réel d’exécution des BLAS n’est pas simplement proportionnel au nombre d’opérations scalaires. De plus, les surcoûts dus aux mécanismes d’agrégation locale (propres aux versions fan-in) ne sont pas comptabilisés, de même que les temps de latence imposés par les communications.
- l’ordonnancement, à l’exécution, des étapes concurrentes de calcul et de communication est limité à des critères empiriques.

Pour résoudre ces problèmes nous avons choisi une approche de type inspecteur-exécuteur, l’étape de repartitionnement et de distribution jouant le rôle d’inspecteur et le solveur parallèle celui d’exécuteur. Ainsi, on génère statiquement un ordonnancement total des calculs de la factorisation, habituellement construit à l’exécution. Il est alors possible de déterminer :

- l’ordre de traitement des blocs-colonnes qui ont leur compteur de contribution  $r_k$  égal à zéro (lignes 1-2),
- l’ordre dans lequel il faut envoyer les blocs-colonnes de contributions agrégées (ligne 15),
- la façon de basculer entre les étapes de calcul/émission (lignes 5-15) et de réception/mise à jour (lignes 2-4).

Pour que cette approche soit réellement exploitable, il est nécessaire de savoir estimer de façon précise la charge de calcul et la latence des communications à l’aide d’un modèle. Ce modèle devra bien sûr être calibré sur l’architecture cible (voir annexe B.1).

Dans notre approche, et contrairement aux solutions couramment proposées, les phases de repartitionnement et de distribution sont dissociées. L’étape de repartitionnement découpe les blocs-colonnes associés aux super-variables les plus larges et construit pour chaque bloc-colonne résultant un ensemble de processeurs candidats qui seront utilisés par l’étape de distribution. Cette dernière affectera de façon efficace un bloc-colonne sur l’un de ces processeurs candidats.

L'algorithme de repartitionnement est basé sur une stratégie de descente récursive de l'arbre d'élimination comme celle présentée par Pothen et Sun dans [67] et dont une description est donnée au paragraphe 6.2.1.

Une fois l'étape de repartitionnement effectuée, on dispose pour chaque bloc-colonne d'un ensemble de processeurs candidats pour la distribution. L'idée principale, pour choisir le processeur qui détiendra finalement le bloc-colonne, repose sur une simulation de la factorisation parallèle en même temps que l'on effectue la distribution. Pour cela, on associe à chaque processeur une horloge correspondant au temps courant dans la factorisation. On définit aussi, pour chaque processeur, une liste (implémentée par un tas relativement à l'ordre donné par une horloge temporelle associée à chaque bloc-colonne) de blocs-colonnes prêts à être factorisés, et contenant à un instant donné tous les blocs colonnes non encore distribués ayant reçu leurs contributions et pour lesquels le processeur est candidat. Le processus est initialisé en commençant par la distribution des feuilles de l'arbre d'élimination pour lesquelles il existe un unique processeur candidat.

Les calculs par blocs sont ordonnés de façon à respecter le rang avec lequel le bloc-colonne à été distribué. Pour un processeur  $p$ , on associe donc le vecteur  $K_p$  contenant les  $N_p$  blocs-colonnes locaux et ordonnés par leur priorité. Mais il faut aussi, pour que le solveur respecte les prédictions de l'étape de distribution, ordonner de façon compatible les émissions des blocs-colonnes de contributions agrégées. Une description de l'algorithme est donnée au paragraphe 6.2.2.

L'algorithme de la factorisation parallèle devra donc être modifié pour prendre en compte ces extensions (voir figure 6.5). Les émissions sont réalisées à l'aide de l'ensemble  $AUCB_{ready}$  contenant l'ensemble des blocs-colonnes de contributions agrégées prêts à être envoyés au processeur  $map(j)$  et dont les priorités sont plus élevées que le premier bloc-colonne de contributions agrégées non prêt.

<p><u>Pour le processeur p :</u></p> <ol style="list-style-type: none"> <li>1. <b>For</b> <math>i = 1</math> <b>To</b> <math>N_p</math> <b>Do</b></li> <li style="padding-left: 20px;">2.       <math>k = K_p(i)</math></li> <li style="padding-left: 20px;">[...]</li> <li style="padding-left: 20px;">3.       recevoir <math>AUCB_k</math>;</li> <li style="padding-left: 20px;">4.       <math>A_{[k]k} = A_{[k]k} - AUCB_k</math>;   <math>r_k \leftarrow r_k - s_k^o</math>;</li> <li style="padding-left: 20px;">[...]</li> <li style="padding-left: 20px;">15.      <b>If</b> <math>s_j = 0</math> <b>Then</b> envoyer <math>AUCB_{ready}</math> à <math>map(j)</math>;</li> </ol>
--

FIG. 6.5: Modifications apportées à l'algorithme de factorisation 1D.

### 6.2.1 Etape de repartitionnement de la partition initiale

Cette étape de repartitionnement doit donc déterminer les nœuds de l'arbre d'élimination par bloc où le parallélisme de second niveau peut être exploité.

Nous utilisons pour découper un nœud un critère minimal en taille (nombre de colonnes) pour une bonne utilisation des primitives BLAS3. En effet, en dessous d'une certaine taille de bloc, les primitives BLAS3 perdent leur efficacité et ce qui pourrait être gagné en parallélisme par un découpage plus fin est perdu en puissance de calcul sur l'appel de ces primitives. La taille de découpage des blocs-colonnes est donc un compromis entre le parallélisme de second niveau et celui de troisième niveau.

Pour chaque nœud  $k$ , on doit estimer l'ensemble des processeurs qui seront susceptibles de participer de manière efficace au calcul de ce nœud  $k$ , ensemble que l'on note  $Cand(k)$ . Si cet ensemble n'a qu'un seul élément, il sera inutile de découper le nœud ; dans le cas contraire, le nœud  $k$  sera découpé et cet ensemble constituera les ensembles  $Cand(k')$  pour tous les nœuds  $k'$  obtenus lors du découpage. Cette technique de découpage est pilotée par un parcours récursif de l'arbre d'élimination.

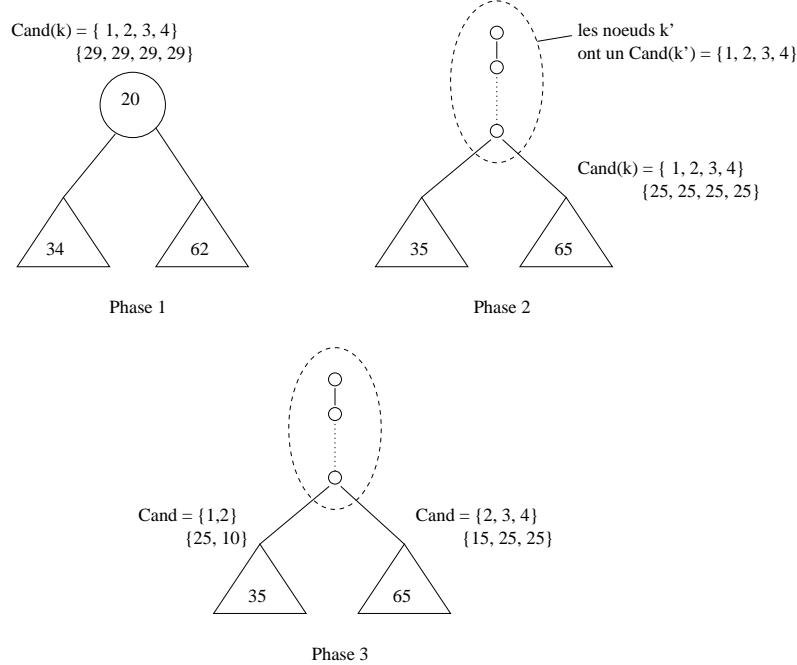


FIG. 6.6: Repartitionnement des nœuds de l'arbre d'élimination par bloc.

L'étape commence par découper la racine de l'arbre ; on recalcule ensuite les coûts des sous-arbres de la racine (qui ont été modifiés par le découpage) ainsi que

le coût que chaque processeur devra prendre parmi les sous-arbres. Enfin on crée pour chaque sous-arbre un nouveau groupe de processeurs dont la somme des coûts à prendre est égale à celle du sous-arbre (les processeurs ayant un coût nul ne peuvent plus faire partie de ces groupes). On réitère l'opération sur les sous-arbres, jusqu'à ce que tous les groupes candidats ne soient plus que des singletons ou que l'on arrive à des feuilles de l'arbre.

Par exemple, sur la figure 6.6, à la phase 1, on a un arbre dont le coût total est 116 ( $20+34+62$ ), et 4 processeurs candidats avec un coût de calcul de 29 chacun. Après découpage lors de la phase 2, on fait apparaître les noeuds  $k'$  pour lesquels les 4 processeurs sont candidats. Le coût total des sous arbres, après découpage, passe à 100, soit 25 pour chacun des 4 processeurs candidats. Dans la phase 3, on affecte 2 processeurs candidats au calcul du premier sous arbre, et 3 pour le second.

Remarquons qu'un processeur peut faire partie de 2 groupes de processeurs candidats comme c'est le cas du processeur 2 dans la figure 6.6. Cette possibilité de recouvrement évite les erreurs d'arrondis des méthodes de distribution par descentes, liées à l'arithmétique en nombre entier. Cela aura pour effet, lors de la phase de distribution qui simule une exécution des blocs-colonnes distribués, de permettre à ces processeurs de "recouvrement" de participer davantage dans le sous-arbre dont le calcul est le plus en retard, assurant ainsi des rééquilibrages intéressants.

### 6.2.2 Etape de distribution

La phase de distribution a pour rôle de déterminer pour chaque bloc-colonne, son processeur propriétaire parmi son ensemble de processeurs candidats. Pour cela, nous allons simuler leur exécution par le solveur, au cours de la distribution.

Nous allons détailler dans la suite de ce paragraphe les étapes importantes de l'algorithme. Tous d'abord nous décrirons certaines notions préalables à la description de l'algorithme :

- la simulation de l'exécution du calcul d'un bloc-colonne,
- le calcul de la date de réception des contributions d'un bloc-colonne,
- la gestion des files d'attentes des processeurs et leur ordonnancement.

Puis nous présenterons le processus de distribution avec en particulier :

- le choix du propriétaire d'un bloc-colonne.

Enfin nous expliquerons :

- le calcul de la priorité des blocs-colonnes et des blocs de contributions agrégées à envoyer, nécessaire au fonctionnement de l'algorithme de factorisation.

- A chaque processeur dont nous simulons l'exécution des tâches de calcul effectuées lors de la factorisation, nous associons une horloge. La simulation de l'exécution des tâches de calcul liées à un bloc-colonne sur ce processeur aura pour fonction de faire avancer cette horloge en fonction des coûts de calcul des tâches réalisées. Cette horloge servira, lorsque nous devrons décider sur quel processeur placer un bloc-colonne, à connaître l'état d'avancement dans le temps de tous les processeurs. Elle nous permet également, au cours de la simulation pour un bloc-colonne  $k$ , de dater la fin de calcul des contributions de  $k$  pour les blocs-colonnes  $j \in BStruct(L_{*k})$ .

La simulation de l'exécution d'un bloc-colonne  $k$  peut commencer dès que toutes les contributions des blocs-colonnes  $i \in BStruct(L_{k*})$  lui ont été ajoutées.

La simulation de l'exécution d'un bloc-colonne  $k$  permet donc de déterminer l'ensemble des blocs-colonnes  $j \in BStruct(L_{*k})$  dont toutes les contributions, issues de  $k$ , sont maintenant calculées. Nous pouvons connaître pour chaque bloc-colonne  $j$  de cet ensemble, la date, pour chaque processeur  $p \in Cand(j)$ , à laquelle il aura reçu toutes les contributions qui lui sont destinées et aura ajouté ses contributions locales, s'il est attribué sur ce processeur  $p$ .

Conceptuellement cette date ne correspond pas au moment où un bloc-colonne est prêt à être factorisé, mais à la date où le processeur possède en local toutes les données nécessaires pour mettre à jour puis factoriser ce bloc-colonne.

De plus, chaque processeur  $p$  dispose d'une file d'attente  $F_p$  des blocs-colonnes dont il possède, à un instant donné, tous les blocs de contributions agrégées qui lui sont destinés (mais non encore ajoutés) et dont ce processeur est candidat. Ces blocs-colonnes sont ordonnés, dans la file  $F_p$ , dans l'ordre chronologique. Un bloc-colonne  $k$  sera présent dans chacune des files d'attente des processeurs  $p \in Cand(k)$ .

- Le processus de distribution commence par placer les feuilles de l'arbre d'élimination, qui ont plus d'un processeur candidat, dans les files d'attente de ces processeurs candidats. On simule ensuite l'exécution des feuilles de l'arbre d'élimination qui n'ont qu'un seul processeur candidat.

Puis, tant qu'il y a des blocs-colonnes à distribuer, le choix du bloc-colonne suivant à distribuer est donné par le premier élément de la file d'attente du processeur qui possède l'horloge la plus en retard. Il faut déterminer lequel des processeurs candidats doit se voir affecté ce bloc-colonne. Pour cela, on détermine le processeur parmi les processeurs candidats sur lequel ce bloc-colonne sera prêt le plus tôt au sens de l'arbre d'élimination. Il ne s'agit pas nécessairement du processeur le plus en retard parmi les processeurs candidats. Ceci est illustré par la figure 6.7 : le processeur 1 est le plus en retard ; pourtant après ajout des blocs de contributions

agrégées, il est prêt pour la factorisation du bloc-colonne à une date ultérieure à celle du processeur 2 qui sera élu propriétaire. Il y a plusieurs raisons à ce phénomène :

- le recouvrement calcul-communication sur les blocs de contributions agrégées varie d'un candidat à l'autre (sur le processeur 4, on peut voir que les communications sont totalement recouvertes par du calcul alors que sur le processeur 1, il y a des périodes d'attente inactive en réception),
- la somme du coût d'ajout des blocs de contributions agrégés varie suivant le candidat.

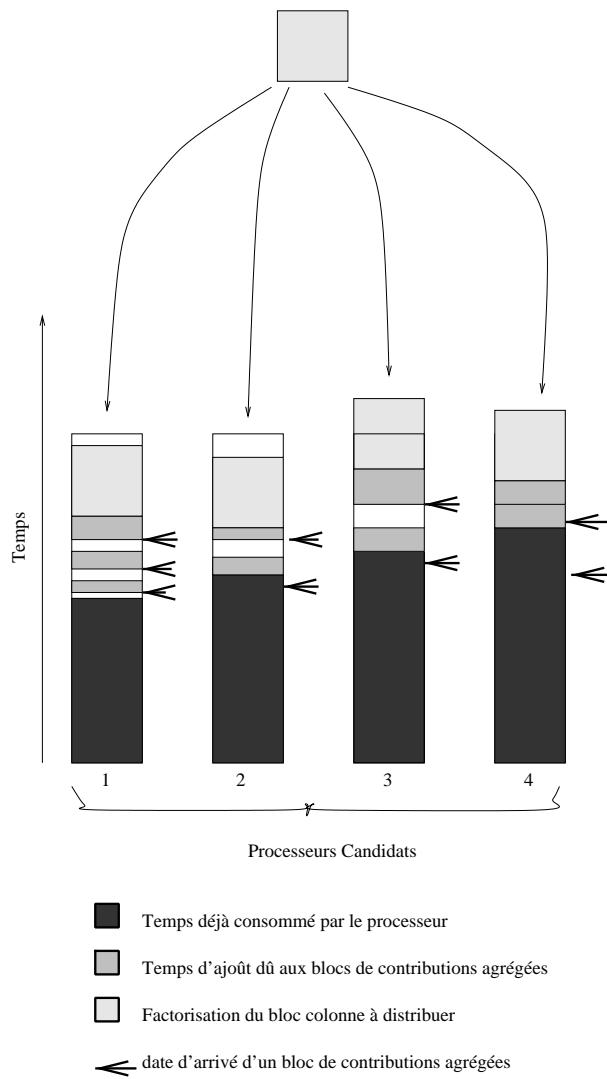


FIG. 6.7: Ajoût des contributions, affectation d'un processeur.

- Toutes les prédictions faites dans l'étape de distribution ne serviront à rien si le solveur ne traite pas rigoureusement les blocs prêts dans l'ordre prévu par le simulateur. Les prédictions seront aussi faussées si le solveur ajoute trop tôt des blocs de contributions agrégées. Pour éviter cela on construit un ordre de traitement imposé par des priorités :

- un bloc-colonne ne pourra pas être traité tant que tous les blocs-colonnes de plus faible priorité n'ont pas été traités,
- un bloc de contributions agrégées ne pourra être envoyé que lorsque tous les blocs de contributions agrégées de plus faible priorité auront été envoyés.

Le calcul de ces priorités est très simple puisque le simulateur distribue les blocs chronologiquement ; il suffira en effet d'attribuer les priorités d'un bloc-colonne et des blocs de contributions agrégées associés selon un compteur que l'on incrémentera à chaque fois qu'un bloc-colonne est simulé.

### 6.3 Expérimentations numériques

Dans cette section, nous présentons les expérimentations effectuées sur une collection de matrices creuses provenant de la collection Harwell-Boeing [25], du projet PARASOL ESPRIT IV LTR No. 20160 et du CEA/CESTA (Cologne 3D).

Tous les algorithmes présentés dans ce chapitre ont été intégrés dans la chaîne logicielle PASTIX [53], développée au LaBRI et basés sur la bibliothèque SCOTCH [64] version 3.4 dédiée au partitionnement de graphes et la renumérotation de matrices creuses (dissections emboîtées et degré minimum approché avec contrainte de Halo [65]). Le développement et la validation ont été effectués sur le calculateur IBM SP2 du LaBRI disposant de 16 nœuds. Les expérimentations parallèles ont été réalisées sur le calculateur IBM SP2 du CINES doté de 192 nœuds fins 120 MHz Power2SC (480 MFlops de puissance crête) disposant chacun de 256 MBytes de mémoire physique et d'un réseau d'interconnexion basé sur un switch TB3. Les expérimentations séquentielles des phases de pré-traitement ont également été effectuées sur un nœud de ce calculateur IBM SP2.

Les mesures donnés dans la table 6.1 ont été obtenus à l'aide d'une factorisation symbolique *scalaire* pour une renumérotation issue de SCOTCH mais sans amalgamation dans la partie degré minimum :  $NNZ_A$  est le nombre de termes extra-diagonaux dans la partie triangulaire inférieure de la matrice  $A$ ,  $NNZ_L$  est le nombre de termes extra-diagonaux dans la matrice factorisée  $L$  et  $OPC$  est le nombre d'opérations pour factoriser.

Nom	Taille	NNZ <sub>A</sub>	NNZ <sub>L</sub>	OPC	Description
<b>GRID511</b>	261121	1041420	1.202166e+07	2.565341e+09	Maillage 2D régulier
<b>GRID767</b>	588289	2348556	2.979676e+07	8.745496e+09	Maillage 2D régulier
<b>GRID1023</b>	1046529	4179980	5.615708e+07	2.083481e+10	Maillage 2D régulier
<b>CUBE31</b>	29791	361890	8.346406e+06	5.525167e+09	Maillage 3D régulier
<b>CUBE39</b>	59319	730778	2.210534e+07	2.240674e+10	Maillage 3D régulier
<b>CUBE47</b>	103823	1290898	4.828456e+07	6.963850e+10	Maillage 3D régulier
<b>144</b>	144649	1074393	4.696077e+07	5.659395e+10	Harwell-Boeing
<b>598A</b>	110971	741934	2.590574e+07	1.876964e+10	Harwell-Boeing
<b>BCSSTK30</b>	28924	1007284	4.309003e+06	1.188915e+09	Harwell-Boeing
<b>BCSSTK32</b>	44609	985046	5.239146e+06	1.162900e+09	Harwell-Boeing
<b>M14B</b>	214765	1679018	6.236747e+07	6.112540e+10	Harwell-Boeing
<b>OCEAN</b>	143437	409593	2.029997e+07	1.301477e+10	Harwell-Boeing
<b>BBMAT</b>	38744	1274141	1.716094e+07	1.250040e+10	Harwell-Boeing
<b>TOOTH</b>	78136	452591	1.031143e+07	6.267094e+09	Harwell-Boeing
<b>B5TUE</b>	162610	3873534	2.541937e+07	1.530774e+10	PARASOL
<b>OILPAN</b>	73752	1761718	8.912337e+06	2.984944e+09	PARASOL
<b>BMWCR41</b>	148770	5247616	6.597301e+07	5.701988e+10	PARASOL
<b>INVEXT</b>	30412	906915	7.256566e+06	3.766788e+09	PARASOL
<b>MIXTANK</b>	29957	982542	9.280247e+06	7.316933e+09	PARASOL
<b>MT1</b>	97578	4827996	3.114873e+07	2.109265e+10	PARASOL
<b>QUER</b>	59122	1403689	9.118592e+06	3.280680e+09	PARASOL
<b>SHIP001</b>	34920	2304655	1.427916e+07	9.033767e+09	PARASOL
<b>SHIP003</b>	121728	3982153	5.872912e+07	8.008089e+10	PARASOL
<b>SHIPSEC5</b>	179860	4966618	5.649801e+07	6.952086e+10	PARASOL
<b>SHIPSEC8</b>	114919	3269240	3.572761e+07	3.684269e+10	PARASOL
<b>THREAD</b>	29736	2220156	2.404333e+07	3.884020e+10	PARASOL
<b>X104</b>	108384	5029620	2.634047e+07	1.712902e+10	PARASOL
<b>BMW3</b>	227362	5530634	4.420244e+07	3.007981e+10	PARASOL
<b>CRANKSEG1</b>	52804	5280703	3.142730e+07	3.007141e+10	PARASOL
<b>CRANKSEG2</b>	63838	7042510	4.190437e+07	4.602878e+10	PARASOL
<b>COL15</b>	10428	701292	3.371109e+06	1.495460e+09	Cologne 3D
<b>COL30</b>	20373	1394292	7.849464e+06	4.359803e+09	Cologne 3D
<b>COL75</b>	50208	3473292	2.248613e+07	1.532035e+10	Cologne 3D

TAB. 6.1: Description de nos cas tests.

### 6.3.1 L'algorithme de repartitionnement et de distribution

La table 6.2 rassemble les temps d'exécution séquentiels en secondes de l'étape de repartitionnement et de distribution pour un nombre de processeurs variant entre 2 et 32, ainsi que le temps de la factorisation symbolique par bloc.

Nous pouvons remarquer que l'algorithme est peu coûteux en temps (à comparer avec les temps de factorisation fournis en table 6.3) et qu'il n'est pas un goulot d'étranglement pour la chaîne logicielle. On peut montrer que, dans le pire de cas, sa complexité varie comme :

$$O(M + N \cdot (\log(P) + \log(N_0)) + P \cdot H \cdot \log(N_0)) ,$$

où, dans l'arbre d'élimination,  $N_0$  est le nombre de feuilles et  $H$  la hauteur et où  $M$  est le nombre total de blocs dans la structure de  $L$ .

Name	Number of processors					SF_Time
	2	4	8	16	32	
<b>GRID511</b>	0.35	0.37	0.43	0.52	0.66	0.83
<b>GRID767</b>	0.73	0.76	0.87	1.06	1.32	1.77
<b>GRID1023</b>	1.43	1.50	1.75	2.20	2.64	3.31
<b>CUBE31</b>	0.09	0.13	0.17	0.27	0.38	0.27
<b>CUBE39</b>	0.17	0.26	0.36	0.71	0.95	0.58
<b>CUBE47</b>	0.51	0.71	0.92	1.80	2.46	1.35
<b>144</b>	2.10	2.92	3.71	4.55	5.99	1.63
<b>598A</b>	1.50	1.86	2.86	3.19	4.17	1.27
<b>BCSSTK30</b>	0.05	0.08	0.09	0.11	0.13	0.47
<b>BCSSTK32</b>	0.14	0.18	0.20	0.24	0.30	0.55
<b>M14B</b>	2.27	3.10	4.62	5.80	6.80	2.43
<b>OCEAN</b>	2.38	2.97	3.62	4.44	5.65	1.29
<b>BBMAT</b>	0.14	0.29	0.41	0.62	0.77	0.69
<b>TOOTH</b>	1.47	1.73	2.19	2.72	2.95	0.96
<b>B5TUE</b>	0.39	0.44	0.51	0.59	0.73	1.93
<b>OILPAN</b>	0.20	0.21	0.22	0.24	0.30	0.88
<b>BMWCR41</b>	0.19	0.27	0.52	0.82	0.97	2.59
<b>INVEXT</b>	0.41	0.62	0.76	0.92	1.35	0.62
<b>MIXTANK</b>	0.24	0.62	0.76	0.92	1.05	0.61
<b>MT1</b>	0.10	0.16	0.22	0.27	0.31	1.80
<b>QUER</b>	0.13	0.15	0.16	0.17	0.20	0.69
<b>SHIP001</b>	0.05	0.06	0.09	0.14	0.17	0.88
<b>SHIP003</b>	0.41	0.67	1.04	1.50	1.86	2.05
<b>SHIPSEC5</b>	0.74	0.84	1.19	1.46	1.87	2.43
<b>SHIPSEC8</b>	0.41	0.64	0.87	1.16	1.31	1.62
<b>THREAD</b>	0.06	0.34	0.37	0.42	0.46	0.87
<b>X104</b>	0.17	0.22	0.26	0.30	0.35	1.95
<b>BMW3</b>	0.54	0.84	0.91	1.08	1.24	3.12
<b>CRANKSEG1</b>	0.05	0.09	0.22	0.34	0.44	1.99
<b>CRANKSEG2</b>	0.05	0.12	0.25	0.43	0.50	2.55
<b>COL15</b>	0.02	0.03	0.04	0.05	0.06	0.05
<b>COL30</b>	0.02	0.05	0.07	0.10	0.13	0.10
<b>COL75</b>	0.07	0.11	0.15	0.21	0.25	0.24

TAB. 6.2: Temps d'exécution des étapes de partitionnement/distribution et factorisation symbolique pour une distribution 1D.

### 6.3.2 La factorisation parallèle

La table 6.3 présente les performances de la factorisation parallèle par blocs pour notre distribution 1D (le **temps** est donné en secondes et la puissance en gigaflops). Les tirets correspondent à des exécutions faussées par des phénomènes de “swapping” ou d'allocation mémoire impossible.

Nom	Nombre de processeurs					
	2	4	8	16	32	64
GRID511	<b>8.06</b> (0.32)	<b>4.17</b> (0.62)	<b>2.36</b> (1.09)	<b>1.47</b> (1.75)	<b>1.16</b> (2.21)	<b>0.97</b> (2.64)
GRID767	-	<b>11.94</b> (0.73)	<b>6.49</b> (1.35)	<b>3.95</b> (2.21)	<b>2.79</b> (3.14)	<b>2.45</b> (3.56)
GRID1023	-	-	<b>18.23</b> (1.14)	<b>8.22</b> (2.53)	<b>5.36</b> (3.88)	<b>5.42</b> (3.84)
CUBE31	<b>11.68</b> (0.47)	<b>6.38</b> (0.87)	<b>3.77</b> (1.47)	<b>2.72</b> (2.03)	<b>2.42</b> (2.28)	<b>2.43</b> (2.28)
CUBE39	<b>42.54</b> (0.53)	<b>23.24</b> (0.96)	<b>13.17</b> (1.70)	<b>8.13</b> (2.76)	<b>6.83</b> (3.28)	<b>5.32</b> (4.21)
CUBE47	-	<b>78.02</b> (0.89)	<b>37.54</b> (1.85)	<b>22.90</b> (3.04)	<b>16.60</b> (4.20)	<b>14.78</b> (4.71)
144	-	<b>70.64</b> (0.80)	<b>38.73</b> (1.46)	<b>22.45</b> (2.52)	<b>13.48</b> (4.20)	<b>11.42</b> (4.95)
598A	<b>54.90</b> (0.34)	<b>26.55</b> (0.71)	<b>14.42</b> (1.30)	<b>8.19</b> (2.29)	<b>5.26</b> (3.57)	<b>4.20</b> (4.47)
BCSSTK30	<b>3.44</b> (0.35)	<b>1.92</b> (0.62)	<b>1.24</b> (0.96)	<b>0.92</b> (1.29)	<b>0.90</b> (1.32)	<b>1.19</b> (1.00)
BCSSTK32	<b>4.08</b> (0.29)	<b>2.54</b> (0.46)	<b>1.19</b> (0.99)	<b>0.85</b> (1.37)	<b>0.68</b> (1.71)	<b>1.53</b> (0.77)
M14B	-	-	<b>51.11</b> (1.20)	<b>24.37</b> (2.51)	<b>13.54</b> (4.51)	<b>9.94</b> (6.15)
OCEAN	<b>48.05</b> (0.27)	<b>25.22</b> (0.52)	<b>13.46</b> (0.97)	<b>7.98</b> (1.63)	<b>4.78</b> (2.72)	<b>3.65</b> (3.56)
BBMAT	<b>33.71</b> (0.37)	<b>16.30</b> (0.77)	<b>9.10</b> (1.37)	<b>5.56</b> (2.25)	<b>3.51</b> (3.56)	<b>3.39</b> (3.69)
TOOTH	<b>20.84</b> (0.30)	<b>11.67</b> (0.54)	<b>6.18</b> (1.01)	<b>4.01</b> (1.56)	<b>3.03</b> (2.07)	<b>2.75</b> (2.28)
B5TUER	<b>32.47</b> (0.47)	<b>17.27</b> (0.89)	<b>9.86</b> (1.55)	<b>5.73</b> (2.67)	<b>4.10</b> (3.73)	<b>3.54</b> (4.33)
OILPAN	<b>7.71</b> (0.39)	<b>4.12</b> (0.73)	<b>2.39</b> (1.25)	<b>1.59</b> (1.88)	<b>1.34</b> (2.23)	<b>1.59</b> (1.88)
BMWCR41	-	-	<b>31.54</b> (1.81)	<b>17.29</b> (3.30)	<b>9.40</b> (6.07)	<b>7.21</b> (8.03)
INVEXT	<b>11.43</b> (0.33)	<b>6.11</b> (0.62)	<b>3.30</b> (1.14)	<b>2.25</b> (1.67)	<b>1.87</b> (2.02)	<b>1.65</b> (2.29)
MIXTANK	<b>16.81</b> (0.44)	<b>9.86</b> (0.74)	<b>5.46</b> (1.34)	<b>3.61</b> (2.03)	<b>3.04</b> (2.41)	<b>3.09</b> (2.37)
MT1	-	<b>24.28</b> (0.87)	<b>13.20</b> (1.60)	<b>7.68</b> (2.75)	<b>5.35</b> (3.94)	<b>4.65</b> (4.54)
QUER	<b>8.07</b> (0.41)	<b>4.31</b> (0.76)	<b>2.48</b> (1.32)	<b>1.76</b> (1.86)	<b>1.57</b> (2.09)	<b>1.47</b> (2.23)
SHIP001	<b>21.03</b> (0.43)	<b>10.72</b> (0.84)	<b>5.68</b> (1.59)	<b>4.03</b> (2.24)	<b>2.31</b> (3.91)	<b>2.14</b> (4.23)
SHIP003	-	<b>117.92</b> (0.68)	<b>45.58</b> (1.76)	<b>25.84</b> (3.10)	<b>16.05</b> (4.99)	<b>12.94</b> (6.19)
SHIPSEC5	-	<b>113.07</b> (0.62)	<b>38.74</b> (1.80)	<b>22.37</b> (3.11)	<b>14.49</b> (4.80)	<b>11.65</b> (5.97)
SHIPSEC8	-	<b>40.28</b> (0.91)	<b>22.69</b> (1.62)	<b>13.22</b> (2.79)	<b>8.69</b> (4.24)	<b>8.04</b> (4.59)
THREAD	<b>102.14</b> (0.38)	<b>41.97</b> (0.93)	<b>23.30</b> (1.67)	<b>14.01</b> (2.77)	<b>12.07</b> (3.22)	<b>9.26</b> (4.20)
X104	<b>37.80</b> (0.45)	<b>20.43</b> (0.84)	<b>11.36</b> (1.51)	<b>6.97</b> (2.46)	<b>5.96</b> (2.87)	<b>5.57</b> (3.07)
BMW3	-	<b>34.65</b> (0.87)	<b>18.70</b> (1.61)	<b>10.78</b> (2.79)	<b>6.98</b> (4.31)	<b>5.91</b> (5.09)
CRANKSEG1	-	<b>31.55</b> (0.95)	<b>17.27</b> (1.74)	<b>10.27</b> (2.93)	<b>6.44</b> (4.67)	<b>5.06</b> (5.95)
CRANKSEG2	-	<b>47.58</b> (0.97)	<b>25.87</b> (1.78)	<b>14.77</b> (3.12)	<b>8.48</b> (5.43)	<b>6.34</b> (7.26)
COL15	<b>3.68</b> (0.41)	<b>1.97</b> (0.76)	<b>1.26</b> (1.18)	<b>0.94</b> (1.60)	<b>0.85</b> (1.75)	<b>0.95</b> (1.57)
COL30	<b>9.83</b> (0.44)	<b>5.23</b> (0.83)	<b>3.02</b> (1.44)	<b>1.99</b> (2.19)	<b>1.62</b> (2.69)	<b>1.67</b> (2.60)
COL75	<b>37.97</b> (0.40)	<b>19.43</b> (0.79)	<b>9.22</b> (1.66)	<b>5.32</b> (2.88)	<b>3.95</b> (3.87)	<b>3.41</b> (4.49)

TAB. 6.3: Temps d'exécution (vitesse en Gflops) de la factorisation parallèle pour une distribution 1D.

Une analyse détaillée d'une exécution confirme que les routines de calcul BLAS 3 (*GEMM* and *TRSM*) représentent plus de 75% du temps total. Une régression, à l'aide d'un polynôme à plusieurs variables, a été utilisée pour obtenir un modèle analytique du coût de calcul de ces routines. Ce modèle, ainsi que les mesures expérimentales du temps de latence et du débit des communications, sont utilisés par l'algorithme de repartitionnement et de distribution. Remarquons enfin que le nombre d'opérations réellement effectuées lors de la factorisation est toujours plus élevé que la valeur de *OPC* de la table 6.1 à cause de la formulation par bloc de l'algorithme de factorisation, et de l'amalgamation réalisée par l'étape de renumérotation.

Une assez bonne scalabilité est obtenue pour l'ensemble des cas tests, au moins pour un nombre moyens de processeurs et ceci malgré une distribution 1D (voir figure 6.8). Pour des tailles moyennes ou grandes de grilles d'éléments finis ainsi que pour les problèmes irréguliers, la puissance mesurée varie entre 1.29 et 3.35 gigaflops sur 16 nœuds et entre 1.32 et 6.16 gigaflops sur 32 nœuds. On obtient en moyenne environ 30% de la puissance crête théorique, pour des calculs réalisés en double précision.

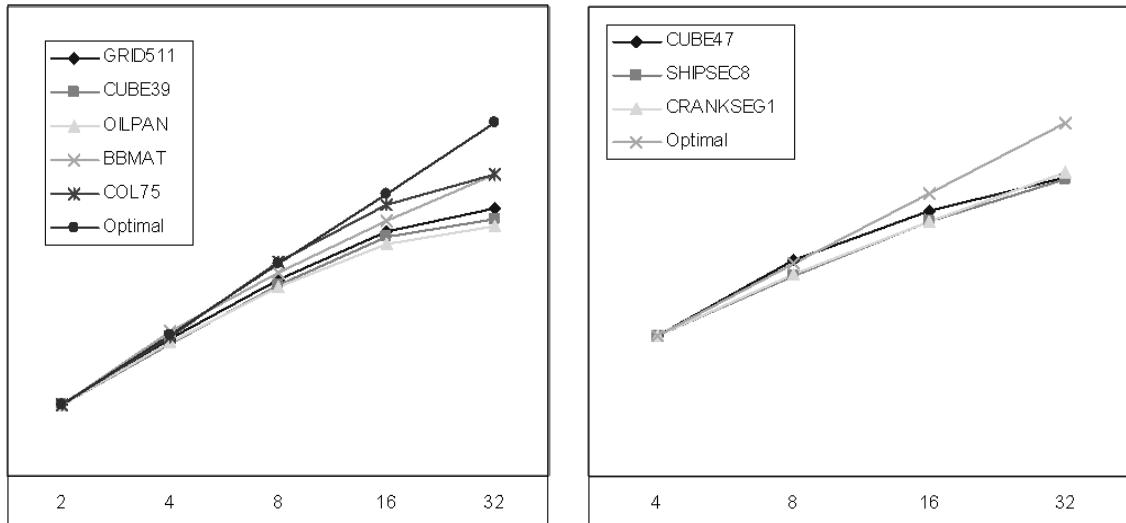


FIG. 6.8: Efficacité relative pour une distribution 1D.

Nous avons également comparé les traces d'exécution du simulateur de l'étape de repartitionnement et de distribution avec les traces de l'exécution réelle de la factorisation numérique parallèle. On voit que l'ordonnancement prédict conduit à une exécution parallèle efficace et maximise l'activité des processeurs. Les traces prédictes et réelles sont très semblables (voir figures 6.9, 6.10, 6.11).

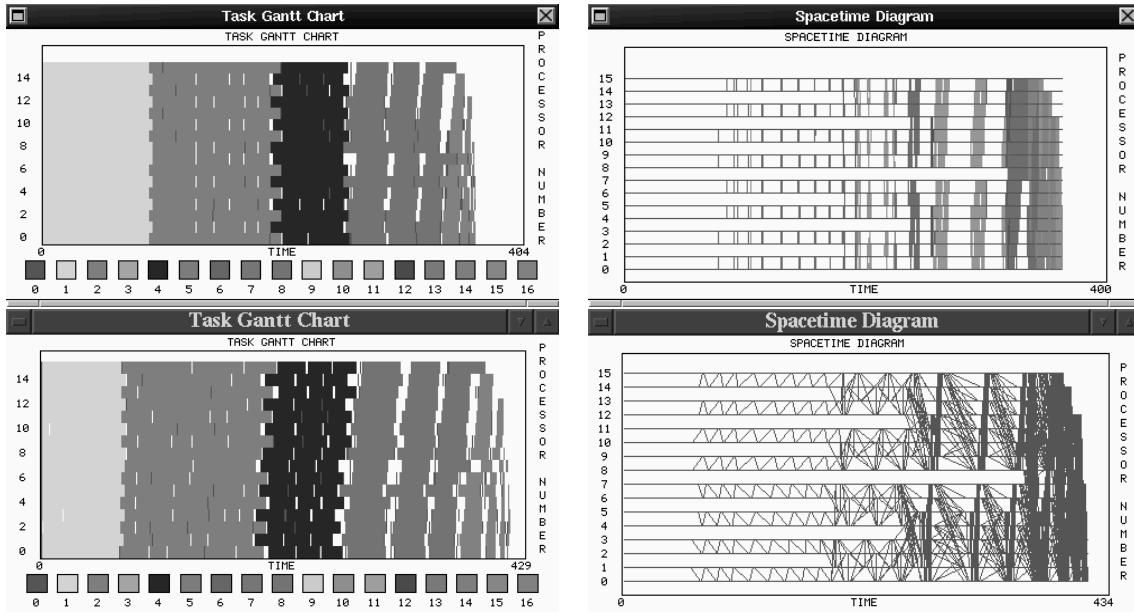


FIG. 6.9: CUBE39 : diagramme de Gantt et de communication pour les traces pré-dites (en haut) et réelles (en bas).

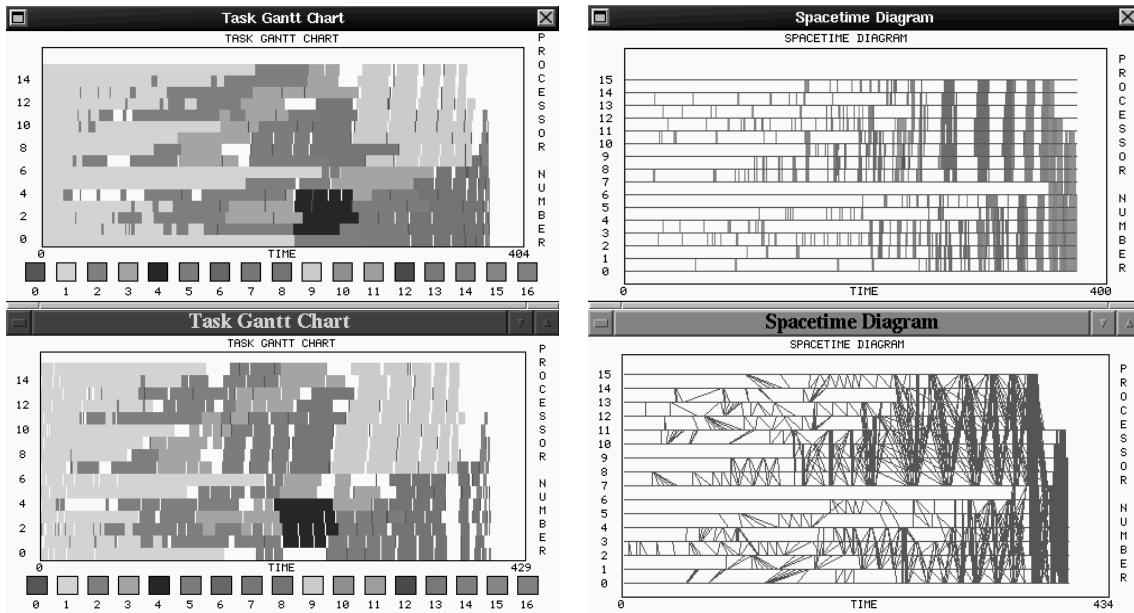


FIG. 6.10: BMW3 : diagramme de Gantt et de communication pour les traces pré-dites (en haut) et réelles (en bas).

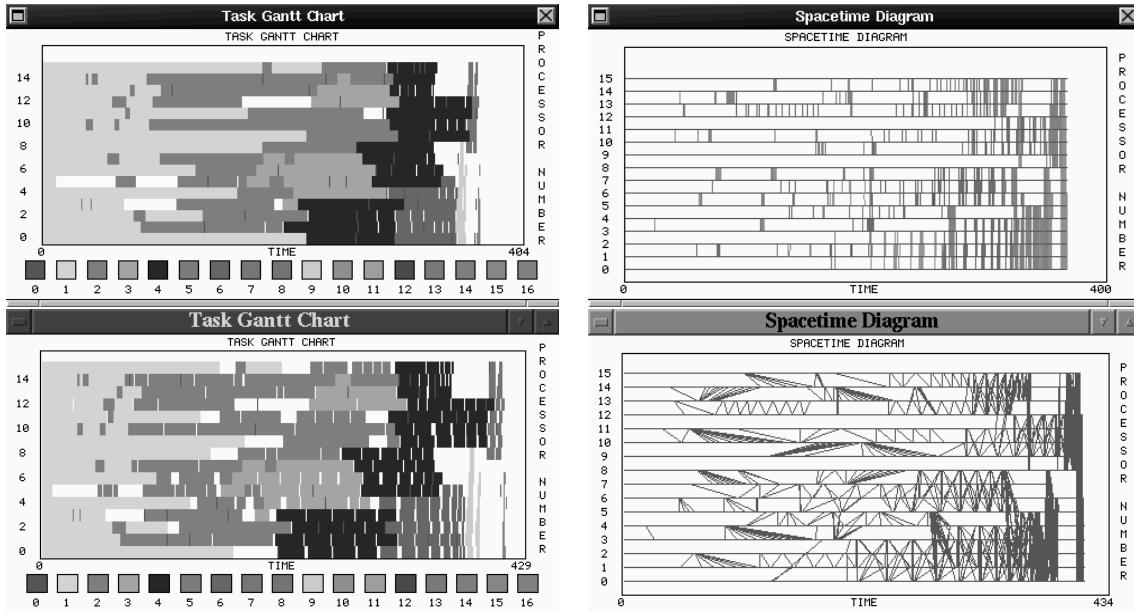


FIG. 6.11: CRANKSEG1 : diagramme de Gantt et de communication pour les traces prédictes (en haut) et réelles (en bas).

### 6.3.3 La résolution du système

L'étape de résolution est composée de 2 phases ; la phase de *descente* qui suit exactement le même schéma d'ordonnancement des calculs que celui de la factorisation, et la phase de *remontée* que nous avons implémentée pour suivre l'ordonnancement inverse de la factorisation. En toute rigueur, on pourrait utiliser notre algorithme de distribution adapté aux calculs et aux contraintes de la résolution mais cela nécessiterait en général une redistribution (coûteuse) après la factorisation. Nous choisissons donc pour l'instant de conserver la même distribution que pour la factorisation.

La table 6.4 présente les performances de la résolution parallèle par blocs pour notre distribution 1D (le **temps** est donné en secondes et la puissance en gigaflops). Il s'agit des temps obtenus pour réaliser une étape de descente-remontée pour un seul vecteur second membre ; on utilise donc des routines de calcul de type BLAS 2.

Nom	Nombre de processeurs				
	2	4	8	16	32
GRID511	<b>0.93</b> (0.05)	<b>0.49</b> (0.10)	<b>0.27</b> (0.18)	<b>0.22</b> (0.22)	<b>0.19</b> (0.26)
GRID767	-	<b>1.04</b> (0.12)	<b>0.60</b> (0.20)	<b>0.36</b> (0.33)	<b>0.27</b> (0.45)
GRID1023	-	-	<b>1.07</b> (0.21)	<b>0.72</b> (0.32)	<b>0.50</b> (0.45)
CUBE31	<b>0.33</b> (0.10)	<b>0.24</b> (0.14)	<b>0.17</b> (0.19)	<b>0.12</b> (0.28)	<b>0.16</b> (0.20)
CUBE39	<b>0.92</b> (0.10)	<b>0.58</b> (0.15)	<b>0.34</b> (0.26)	<b>0.22</b> (0.40)	<b>0.23</b> (0.39)
CUBE47	-	<b>1.11</b> (0.17)	<b>0.64</b> (0.30)	<b>0.42</b> (0.46)	<b>0.40</b> (0.49)
144	-	<b>1.78</b> (0.11)	<b>0.94</b> (0.20)	<b>0.57</b> (0.33)	<b>0.46</b> (0.41)
598A	<b>1.82</b> (0.06)	<b>1.03</b> (0.10)	<b>0.65</b> (0.16)	<b>0.39</b> (0.26)	<b>0.27</b> (0.39)
BCSSTK30	<b>0.19</b> (0.09)	<b>0.24</b> (0.07)	<b>0.11</b> (0.16)	<b>0.06</b> (0.31)	<b>0.05</b> (0.36)
BCSSTK32	<b>1.22</b> (0.02)	<b>0.16</b> (0.13)	<b>0.11</b> (0.18)	<b>0.08</b> (0.28)	<b>0.07</b> (0.31)
M14B	-	-	<b>1.43</b> (0.17)	<b>0.82</b> (0.30)	<b>0.59</b> (0.42)
OCEAN	-	<b>1.51</b> (0.05)	<b>0.75</b> (0.11)	<b>0.41</b> (0.20)	<b>0.35</b> (0.23)
BBMAT	<b>0.64</b> (0.11)	<b>0.37</b> (0.19)	<b>0.24</b> (0.28)	<b>0.18</b> (0.38)	<b>0.14</b> (0.47)
TOOTH	<b>1.34</b> (0.03)	<b>0.71</b> (0.06)	<b>0.46</b> (0.09)	<b>0.27</b> (0.15)	<b>0.29</b> (0.14)
B5TUEER	<b>1.03</b> (0.10)	<b>0.61</b> (0.17)	<b>0.41</b> (0.25)	<b>0.25</b> (0.40)	<b>0.21</b> (0.50)
OILPAN	<b>0.43</b> (0.08)	<b>0.25</b> (0.14)	<b>0.16</b> (0.22)	<b>0.13</b> (0.28)	<b>0.09</b> (0.38)
BMWCRA1	-	-	<b>1.28</b> (0.21)	<b>0.42</b> (0.63)	<b>0.49</b> (0.54)
INVEXT	<b>0.46</b> (0.06)	<b>0.33</b> (0.09)	<b>0.19</b> (0.15)	<b>0.12</b> (0.24)	<b>0.12</b> (0.25)
MIXTANK	<b>0.44</b> (0.08)	<b>0.29</b> (0.13)	<b>0.18</b> (0.20)	<b>0.16</b> (0.24)	<b>0.31</b> (0.12)
MT1	<b>1.00</b> (0.12)	<b>0.56</b> (0.22)	<b>0.34</b> (0.36)	<b>0.25</b> (0.51)	<b>0.23</b> (0.53)
QUER	<b>0.39</b> (0.09)	<b>0.23</b> (0.16)	<b>0.15</b> (0.24)	<b>0.10</b> (0.37)	<b>0.08</b> (0.44)
SHIP001	<b>0.45</b> (0.13)	<b>0.26</b> (0.22)	<b>0.22</b> (0.26)	<b>0.12</b> (0.46)	<b>0.10</b> (0.55)
SHIP003	-	-	<b>0.87</b> (0.27)	<b>0.48</b> (0.49)	<b>0.43</b> (0.55)
SHIPSEC5	-	-	<b>0.80</b> (0.28)	<b>0.48</b> (0.48)	<b>0.45</b> (0.50)
SHIPSEC8	-	<b>0.88</b> (0.16)	<b>0.51</b> (0.28)	<b>0.38</b> (0.38)	<b>0.42</b> (0.34)
THREAD	-	<b>0.59</b> (0.16)	<b>0.39</b> (0.25)	<b>0.27</b> (0.35)	<b>0.26</b> (0.37)
X104	<b>1.28</b> (0.08)	<b>0.60</b> (0.18)	<b>0.36</b> (0.29)	<b>0.26</b> (0.41)	<b>0.24</b> (0.45)
BMW3	-	<b>4.55</b> (0.04)	<b>0.72</b> (0.24)	<b>0.47</b> (0.38)	<b>0.38</b> (0.47)
CRANKSEG1	-	<b>0.60</b> (0.21)	<b>0.39</b> (0.32)	<b>0.27</b> (0.46)	<b>0.21</b> (0.61)
CRANKSEG2	-	<b>0.78</b> (0.21)	<b>0.50</b> (0.34)	<b>0.31</b> (0.54)	<b>0.27</b> (0.62)
COL15	<b>0.12</b> (0.11)	<b>0.10</b> (0.14)	<b>0.13</b> (0.11)	<b>0.05</b> (0.29)	<b>0.05</b> (0.29)
COL30	<b>0.25</b> (0.12)	<b>0.18</b> (0.18)	<b>0.12</b> (0.26)	<b>0.07</b> (0.44)	<b>0.09</b> (0.34)
COL75	<b>0.69</b> (0.13)	<b>0.42</b> (0.21)	<b>0.26</b> (0.35)	<b>0.17</b> (0.52)	<b>0.13</b> (0.69)

TAB. 6.4: Temps d'exécution (vitesse en Gflops) de la résolution parallèle pour une distribution 1D.

Les expérimentations effectuées montrent encore une assez bonne scalabilité et le temps d'une résolution reste faible par rapport à celui de la factorisation.

Précisons que comme pour la factorisation, les contributions distantes sont agrégées localement et qu'il est bien sûr possible de traiter plusieurs seconds membres à la fois en une seule étape de résolution permettant ainsi d'utilisation des routines BLAS 3. Enfin la précision obtenue, en calcul double précision, est en moyenne de l'ordre de  $10^{-12}$  par rapport à une solution connue à l'avance.

## 6.4 Conclusion

Dans ce chapitre nous avons présenté un ordonnancement statique efficace des calculs par blocs pour un solveur creux direct parallèle. Dans le chapitre suivant nous allons présenter une extension de la méthode pour prendre en compte une distribution 2D des blocs dans la partie haute de l'arbre d'élimination, et améliorer ainsi la scalabilité du solveur pour pouvoir utiliser efficacement davantage de processeurs.

# Chapitre 7

## Solveur $LDL^t$ avec distribution 2D

### 7.1 Introduction

Dans ce chapitre nous allons présenter une extension des algorithmes vus précédemment pour prendre en compte une distribution bloc 2D dont on sait qu'elle engendre en théorie une meilleur scalabilité [76]. Cette approche apporte une granularité plus fine, offrant plus de liberté dans l'enchaînement des calculs et donc plus de parallélisme. Elle a par contre l'inconvénient de générer plus de communications.

Nous présentons tout d'abord l'algorithme de la factorisation parallèle d'une matrice creuse symétrique définie positive sur lequel nous nous appuyons pour décrire le partitionnement et la distribution 2D par bloc ; plus précisement, nous présentons une extention des travaux décrits dans le chapitre précédent, ce qui conduit au calcul d'une régulation statique basée sur une combinaison de distributions par bloc 1D et 2D et au solveur parallèle associé. Nous décrivons ensuite nos expérimentations numériques toujours sur un IBM SP2 pour une large classe de matrices creuses ainsi qu'une analyse des performances obtenues. Puis nous comparons ces performances avec celles obtenues par le solveur du logiciel de référence PSPASES développé à l'Université du Minnesota à Minneapolis aux USA. Enfin, avant de conclure et d'exposer les perspectives de ces travaux, nous analyserons la scalabilité mémoire de notre solveur parallèle ainsi que l'application de notre ordonnancement statique pour la factorisation d'une matrice pleine.

### 7.2 Factorisation parallèle et algorithme de distribution

On considère toujours la structure de données par blocs de la matrice factorisée  $L$  calculée par la factorisation symbolique par blocs. Rappelons qu'un bloc-colonne est constitué d'un bloc diagonal plein et d'un ensemble de blocs extra-diagonaux pleins. On considère toujours les deux ensembles :

$$\begin{cases} BStruct(L_{k*}) & := \{i < k \mid off\_diag(i, k) = true\} \\ BStruct(L_{*k}) & := \{j > k \mid off\_diag(k, j) = true\} \end{cases}$$

avec  $off\_diag(k, j)$ ,  $1 \leq k < j \leq N$  qui retourne *vrai* si et seulement s'il existe un bloc extra-diagonal dans le bloc-colonne  $k$  “en face” du bloc-colonne  $j$ . Contrairement à une distribution 1D comme vue précédemment, les blocs extra-diagonaux d'un bloc-colonne ne seront pas forcément tous distribués sur le même processeur pour une distribution 2D.

On s'intéresse toujours à l'algorithme supernodal de la factorisation parallèle creuse  $LDL^t$  où toutes les contributions distantes sont localement agrégées dans une structure bloc. Un bloc  $j$  dans un bloc-colonne  $k$  recevra donc un bloc de contributions agrégées, noté  $AUB_{jk}$  par la suite, seulement de la part des processeurs dans l'ensemble  $Procs(L_{jk}) := \{map(j, i) \mid i \in BStruct(L_{k*}) \text{ and } j \in BStruct(L_{*i})\}$ , où l'opérateur  $map(, )$  correspond à la distribution 2D des blocs.

Le “pseudo-code” de la factorisation  $LDL^t$  peut-être exprimé en terme de calculs par blocs (voir figure 7.1) ; les calculs effectifs sont réalisés, autant que possible, sur des ensembles de blocs compactés pour profiter au mieux des effets BLAS.

L'algorithme proposé permet de prendre en compte à la fois une distribution 1D et 2D ; un bloc-colonne distribué en 1D est vu comme s'il était distribué en 2D avec tous les blocs extra-diagonaux placés sur le même processeur ; cependant, ce cas particulier est traité spécifiquement au niveau du code pour effectuer les calculs sur l'ensemble des blocs compactés. Les calculs par blocs, encore appelée tâches, sont totalement ordonnés lors de l'étape de distribution. Les émissions des blocs de contributions agrégées sont également ordonnées de façon compatible. On peut alors définir quatre types de tâche :

- **COMP1D(k)** : mise à jour et calcul des contributions pour le bloc colonne  $k$  dans le cas d'une distribution 1D,
- **FACTOR(k)** : factorisation du bloc diagonal  $k$ ,
- **BDIV(j,k)** : mise à jour du bloc extra-diagonal  $j$  dans le bloc-colonne  $k$  avec  $k < j \leq N$ ,
- **BMOD(i,j,k)** : calcul de la contribution du bloc  $i$  dans le bloc-colonne  $k$  “en face” du bloc-colonne  $j$  avec  $k < j \leq i \leq N$ .

**Notations pour  $k$  fixé,  $1 \leq k \leq N$  :**

- $N_p$  : nombre total de tâches pour le processeur  $p$ ,
- $K_p[n]$  :  $n^{\text{ième}}$  tâche locale pour le processeur  $p$ ,
- pour un bloc-colonne  $k$ , le symbole  $\star$  signifie  $\forall j \in BStruct(L_{*k})$ ,
- soit  $j \geq k$  ; la séquence  $[j]$  signifie  $\forall i \in BStruct(L_{*k}) \cup \{k\}$  avec  $i \geq j$ ,
- $map([j], j)$  représente, par extension,  $map(i, j)$  pour chaque valeur de  $i \in [j]$ .

Pour le processeur p :

1. **For**  $n = 1$  to  $N_p$  **Do**
2. **Switch** ( Type( $K_p[n]$ ) ) **Do**
3. COMP1D(k) : Recevoir tous les  $AUB_{[k]k}$  pour  $A_{[k]k}$
4. Factoriser  $A_{kk}$  sous la forme  $L_{kk}D_kL_{kk}^t$
5. Résoudre  $L_{kk}F^t = A_{*k}^t$  et  $D_kL_{*k}^t = F^t$
6. **For**  $j \in BStruct(L_{*k})$  **Do**
7. Calculer  $C_{[j]} = L_{[j]k}F_j^t$
8. **If**  $map([j], j) == p$  **Then**  $A_{[j]j} = A_{[j]j} - C_{[j]}$
9. **Else**  $AUB_{[j]j} = AUB_{[j]j} + C_{[j]}$ , si prêt envoyer  $AUB_{[j]j}$  à  $map([j], j)$
10. FACTOR(k) : Recevoir tous les  $AUB_{kk}$  pour  $A_{kk}$
11. Factoriser  $A_{kk}$  sous la forme  $L_{kk}D_kL_{kk}^t$  et envoyer  $L_{kk}D_k$  à  $map([k], k)$
12. BDIV(j, k) : Recevoir  $L_{kk}D_k$  et tous les  $AUB_{jk}$  pour  $A_{jk}$
13. Résoudre  $L_{kk}F_j^t = A_{jk}^t$  et  $D_kL_{jk}^t = F_j^t$  et envoyer  $F_j^t$  à  $map([j], k)$
14. BMOD(i, j, k) : Recevoir  $F_j^t$
15. Calculer  $C_i = L_{ik}F_j^t$
16. **If**  $map(i, j) == p$  **Then**  $A_{ij} = A_{ij} - C_i$
17. **Else**  $AUB_{ij} = AUB_{ij} + C_i$ , si prêt envoyer  $AUB_{ij}$  à  $map(i, j)$

FIG. 7.1: Algorithme de la factorisation parallèle pour une distribution mixte 1D/2D.

De même qu'au chapitre précédent pour une distribution 1D, avant de pouvoir exécuter l'algorithme général présenté ci-dessus, il est nécessaire de réaliser l'étape de repartitionnement et de distribution des blocs, issus de la factorisation symbolique, sur l'ensemble des processeurs.

Nous utilisons toujours une approche de type inspecteur-exécuteur ; l'étape de repartitionnement et de distribution jouant le rôle d'inspecteur et la factorisation parallèle celui d'exécuteur. Ainsi, on générera un ordonnancement total des calculs de la factorisation, habituellement construit à l'exécution. Rappelons que les phases de repartitionnement et de distribution sont dissociées. La principale différence réside ici dans le fait qu'on ne distribue plus des blocs-colonnes, mais des tâches de calculs.

Comme l'algorithme traite à la fois des distributions 1D ou 2D, l'algorithme de repartitionnement a été modifié pour que le choix du type de distribution soit réalisé lors de la descente récursive de l'arbre d'élimination par bloc. Ainsi, les coûts affectés à chacun des sous-arbres peuvent prendre en compte le type de distribution. En effet la somme des coûts de calcul pour une distribution 2D sera (légèrement) supérieure à celle obtenue pour une distribution 1D à cause de la perte d'efficacité des routines BLAS, les calculs n'étant plus effectués de façon compactée. Le critère du changement du type de distribution est actuellement un critère de niveau (ou de seuil) dans l'arbre d'élimination. Pour les nœuds compris entre la racine de l'arbre et ce niveau, on utilise une distribution 2D, car il s'agit de blocs suffisamment importants pour pouvoir exploiter du parallélisme de niveau 2. Pour les nœuds plus bas dans l'arbre d'élimination, on utilise une distribution 1D. Dans tous les cas, si le nombre de processeurs candidats, calculés par l'étape de repartitionnement, est inférieur ou égal à 2 alors on force l'utilisation d'une distribution 1D. Des travaux sont en cours pour améliorer ce choix, et en particulier pour exprimer ce critère en fonction de la taille des blocs-colonnes et du nombre de processeurs.

Une fois l'étape de repartitionnement effectuée, on dispose pour chaque bloc-colonne d'un ensemble de processeurs candidats ainsi que du type de distribution retenue. La distribution des tâches s'effectue alors, comme dans le chapitre précédent, en même temps que l'on simule la factorisation parallèle. Pour un bloc-colonne distribué en 2D, on simule indépendamment chacune des tâches 2D qui lui sont associées en respectant bien sûr les contraintes de précédence.

Les calculs par blocs sont ordonnés de façon à respecter le rang selon lequel les tâches associées ont été distribuées. Pour un processeur  $p$ , on associe donc le vecteur  $K_p$  contenant les  $N_p$  tâches locales et ordonnées par leur priorité (voir figure 7.1).

## 7.3 Expérimentations numériques

### 7.3.1 Performances de la factorisation parallèle

Nous allons présenter maintenant les performances obtenues par notre solveur avec notre distribution mixte 1D/2D pour les mêmes jeux de tests qu'au chapitre précédent (voir table 6.1). On se place dans les mêmes conditions d'expérimentation qu'au paragraphe 6.3.

La table 7.1 présente les performances obtenues pour la factorisation avec la nouvelle approche ; à chaque fois, on a pris le meilleur temps quand on fait varier le seuil dans l'arbre d'élimination pour le changement entre distributions 1D et 2D (le **temps** est donné en secondes et la puissance en gflops). Notons enfin que l'implémentation de la nouvelle approche est légèrement moins efficace avec une distribution uniquement 1D ; pour quelques matrices, les temps présentés à la table 6.3 restent meilleurs. Les routines BLAS 3 représentent toujours plus de 75% du temps total.

Nom	Nombre de processeurs					
	2	4	8	16	32	64
<b>GRID511</b>	<b>8.25</b> (0.31)	<b>4.28</b> (0.60)	<b>2.46</b> (1.04)	<b>1.56</b> (1.65)	<b>1.01</b> (2.55)	<b>0.79</b> (3.26)
<b>GRID767</b>	-	<b>12.29</b> (0.71)	<b>6.63</b> (1.32)	<b>4.10</b> (2.13)	<b>2.66</b> (3.29)	<b>2.01</b> (4.36)
<b>GRID1023</b>	-	-	<b>14.57</b> (1.43)	<b>8.61</b> (2.42)	<b>5.66</b> (3.68)	<b>3.75</b> (5.56)
<b>CUBE31</b>	<b>11.81</b> (0.47)	<b>6.52</b> (0.85)	<b>3.89</b> (1.42)	<b>2.77</b> (2.00)	<b>2.17</b> (2.55)	<b>1.88</b> (2.94)
<b>CUBE39</b>	<b>43.58</b> (0.51)	<b>24.20</b> (0.93)	<b>13.58</b> (1.65)	<b>8.45</b> (2.65)	<b>6.50</b> (3.45)	<b>5.03</b> (4.46)
<b>CUBE47</b>	-	<b>73.31</b> (0.95)	<b>38.23</b> (1.82)	<b>23.88</b> (2.92)	<b>16.31</b> (4.27)	<b>12.45</b> (5.59)
<b>144</b>	-	-	<b>36.06</b> (1.57)	<b>21.33</b> (2.65)	<b>15.40</b> (3.67)	<b>11.83</b> (4.78)
<b>598A</b>	<b>78.39</b> (0.24)	<b>26.34</b> (0.71)	<b>14.40</b> (1.30)	<b>8.56</b> (2.19)	<b>5.53</b> (3.39)	<b>4.45</b> (4.22)
<b>BCSSTK30</b>	<b>3.49</b> (0.34)	<b>2.00</b> (0.60)	<b>1.13</b> (1.05)	<b>0.88</b> (1.36)	<b>0.77</b> (1.55)	<b>0.76</b> (1.55)
<b>BCSSTK32</b>	<b>4.03</b> (0.29)	<b>2.22</b> (0.52)	<b>1.37</b> (0.85)	<b>0.98</b> (1.19)	<b>0.78</b> (1.49)	<b>0.71</b> (1.63)
<b>M14B</b>	-	-	<b>46.95</b> (1.30)	<b>22.33</b> (2.74)	<b>14.06</b> (4.35)	<b>9.79</b> (6.25)
<b>OCEAN</b>	<b>48.37</b> (0.27)	<b>18.39</b> (0.71)	<b>9.85</b> (1.32)	<b>5.83</b> (2.23)	<b>3.82</b> (3.41)	<b>3.31</b> (3.93)
<b>BBMAT</b>	<b>28.17</b> (0.44)	<b>14.73</b> (0.85)	<b>8.16</b> (1.53)	<b>5.05</b> (2.47)	<b>3.64</b> (3.44)	<b>2.75</b> (4.55)
<b>TOOTH</b>	<b>20.01</b> (0.31)	<b>10.70</b> (0.59)	<b>6.24</b> (1.00)	<b>4.47</b> (1.40)	<b>2.93</b> (2.14)	<b>2.70</b> (2.32)
<b>B5TUEER</b>	<b>29.54</b> (0.52)	<b>15.52</b> (0.99)	<b>8.86</b> (1.73)	<b>5.25</b> (2.91)	<b>3.96</b> (3.87)	<b>2.91</b> (5.25)
<b>OILPAN</b>	<b>7.28</b> (0.41)	<b>3.81</b> (0.78)	<b>2.15</b> (1.39)	<b>1.39</b> (2.14)	<b>1.00</b> (3.00)	<b>0.87</b> (3.42)
<b>BMWCR41</b>	-	-	<b>30.97</b> (1.84)	<b>17.28</b> (3.30)	<b>9.89</b> (5.76)	<b>6.94</b> (8.21)
<b>INVEXT</b>	<b>11.59</b> (0.32)	<b>6.31</b> (0.60)	<b>3.47</b> (1.09)	<b>2.57</b> (1.47)	<b>1.78</b> (2.12)	<b>1.48</b> (2.55)
<b>MIXTANK</b>	<b>17.32</b> (0.42)	<b>10.02</b> (0.73)	<b>5.91</b> (1.24)	<b>3.90</b> (1.87)	<b>3.23</b> (2.27)	<b>3.17</b> (2.31)
<b>MT1</b>	<b>37.92</b> (0.56)	<b>20.35</b> (1.04)	<b>11.29</b> (1.87)	<b>6.65</b> (3.17)	<b>4.33</b> (4.87)	<b>3.51</b> (6.01)
<b>QUER</b>	<b>8.35</b> (0.39)	<b>4.46</b> (0.74)	<b>2.57</b> (1.28)	<b>1.67</b> (1.96)	<b>1.16</b> (2.83)	<b>0.93</b> (3.53)
<b>SHIP001</b>	<b>20.98</b> (0.43)	<b>10.91</b> (0.83)	<b>6.07</b> (1.49)	<b>3.63</b> (2.49)	<b>2.43</b> (3.72)	<b>1.96</b> (4.60)
<b>SHIP003</b>	-	<b>109.78</b> (0.73)	<b>45.83</b> (1.75)	<b>25.75</b> (3.11)	<b>16.60</b> (4.83)	<b>12.03</b> (6.66)
<b>SHIPSEC5</b>	-	<b>79.12</b> (0.88)	<b>35.68</b> (1.95)	<b>20.51</b> (3.39)	<b>13.99</b> (4.97)	<b>11.58</b> (6.00)
<b>SHIPSEC8</b>	-	<b>43.40</b> (0.85)	<b>24.83</b> (1.48)	<b>15.40</b> (2.39)	<b>11.39</b> (3.23)	<b>9.52</b> (3.87)
<b>THREAD</b>	<b>78.04</b> (0.50)	<b>41.14</b> (0.94)	<b>22.85</b> (1.70)	<b>13.50</b> (2.88)	<b>10.42</b> (3.73)	<b>6.70</b> (5.80)
<b>X104</b>	<b>31.53</b> (0.54)	<b>19.69</b> (0.87)	<b>9.98</b> (1.72)	<b>7.49</b> (2.29)	<b>5.09</b> (3.37)	<b>3.85</b> (4.44)
<b>BMW3</b>	-	<b>34.02</b> (0.88)	<b>18.27</b> (1.65)	<b>10.90</b> (2.76)	<b>7.69</b> (3.91)	<b>6.07</b> (4.96)
<b>CRANKSEG1</b>	-	<b>33.54</b> (0.90)	<b>18.73</b> (1.61)	<b>10.54</b> (2.85)	<b>6.74</b> (4.46)	<b>5.82</b> (5.17)
<b>CRANKSEG2</b>	-	<b>47.99</b> (0.96)	<b>25.33</b> (1.82)	<b>14.15</b> (3.25)	<b>8.92</b> (5.16)	<b>6.20</b> (7.42)

TAB. 7.1: Temps d'exécution (vitesse en Gflops) de la factorisation parallèle pour une distribution 2D.

Une bonne scalabilité réelle est obtenue pour l'ensemble des cas tests jusqu'à 64 processeurs. Pour des tailles moyennes ou grandes de grilles d'éléments finis ainsi que pour les problèmes irréguliers, la puissance mesurée varie entre 1.49 et 5.76 gflops sur 32 nœuds et entre 1.57 et 8.21 gflops sur 64 nœuds. On obtient en moyenne environ 30% de la puissance crête théorique, pour des calculs réalisés en double précision.

Les expérimentations que nous avons menées ensuite montrent que le seuil (niveau dans l'arbre d'élimination) qui contrôle le passage d'une distribution 1D vers une distribution 2D, augmente avec le nombre de processeurs (voir table 7.2). Plus le nombre de processeurs augmente, plus il faut descendre dans l'arbre avec une distribution 2D. Ce résultat n'est pas trivial car le critère doit dépendre aussi de la taille du bloc-colonne. On trouve cependant que pour la majorité des cas tests, il faut utiliser uniquement une distribution 1D jusqu'à 8 processeurs. Entre 16 et 32 processeurs, le seuil donnant les meilleures performances varie entre 1 et 5. Pour 64 processeurs il varie entre 3 et 9. Plus la taille du problème est importante (la hauteur de l'arbre d'élimination est alors plus grande), plus il faut augmenter le seuil. Remarquons enfin, que pour les faibles nombres de processeurs, l'espace mémoire est parfois limite et on peut observer une dégradation des performances liée à la pagination ; comme la distribution mixte conduit à une meilleure scalabilité mémoire (voir paragraphe 7.4) une meilleure performance d'ensemble est alors obtenue.

Nom	Nombre de processeurs					
	2	4	8	16	32	64
OILPAN 1D pure	<b>7.28</b>	<b>3.81</b>	2.19	1.56	1.21	1.14
OILPAN seuil 1	7.31	3.84	2.18	1.53	1.21	1.13
OILPAN seuil 2	7.85	3.83	<b>2.15</b>	1.41	1.21	1.02
OILPAN seuil 3	7.46	3.82	2.34	<b>1.39</b>	1.06	0.95
OILPAN seuil 4	7.33	3.86	2.31	1.48	1.14	0.98
OILPAN seuil 5	7.39	3.82	2.27	1.44	1.10	0.98
OILPAN seuil 6	7.35	3.93	2.28	1.51	1.04	0.88
OILPAN seuil 7	7.38	3.83	2.30	1.51	<b>1.00</b>	0.92
OILPAN seuil 8	7.40	4.08	2.29	1.51	1.16	0.94
OILPAN seuil 9	7.42	3.83	2.30	1.76	1.04	0.96
OILPAN seuil 10	7.41	3.82	2.30	1.54	1.12	0.91
OILPAN 2D pure	8.49	3.95	2.32	1.55	1.10	<b>0.87</b>
B5TUE 1D pure	30.27	<b>15.52</b>	8.91	5.30	4.15	3.84
B5TUE seuil 1	30.53	15.64	<b>8.86</b>	5.49	4.02	3.67
B5TUE seuil 2	32.21	15.79	9.18	<b>5.25</b>	4.30	3.44
B5TUE seuil 3	31.62	16.78	9.57	6.07	4.24	3.22
B5TUE seuil 4	32.75	16.27	10.09	6.67	4.28	3.33
B5TUE seuil 5	29.84	16.37	10.16	6.39	4.59	3.15
B5TUE seuil 6	29.60	16.24	10.12	6.40	4.22	3.30
B5TUE seuil 7	30.03	16.29	10.17	6.37	4.25	2.93
B5TUE seuil 8	<b>29.54</b>	16.32	10.13	6.62	4.08	2.97
B5TUE seuil 9	30.12	16.38	10.17	6.38	4.28	3.09
B5TUE seuil 10	29.80	16.51	10.14	6.36	<b>3.96</b>	<b>2.91</b>
B5TUE 2D pure	33.78	16.61	10.16	6.33	4.42	3.13

TAB. 7.2: Influence du seuil de changement de distribution sur le temps de factorisation pour les cas tests OILPAN et B5TUE.

La figure 7.2 illustre la bonne occupation des 64 processeurs pour un des plus gros cas tests. La partie gauche de la trace d'exécution correspond à une distribution 1D alors que la partie droite correspond à une distribution 2D pour les blocs-colonnes correspondant aux plus gros séparateurs générés dans la phase de la renumérotation de type dissections emboîtées.

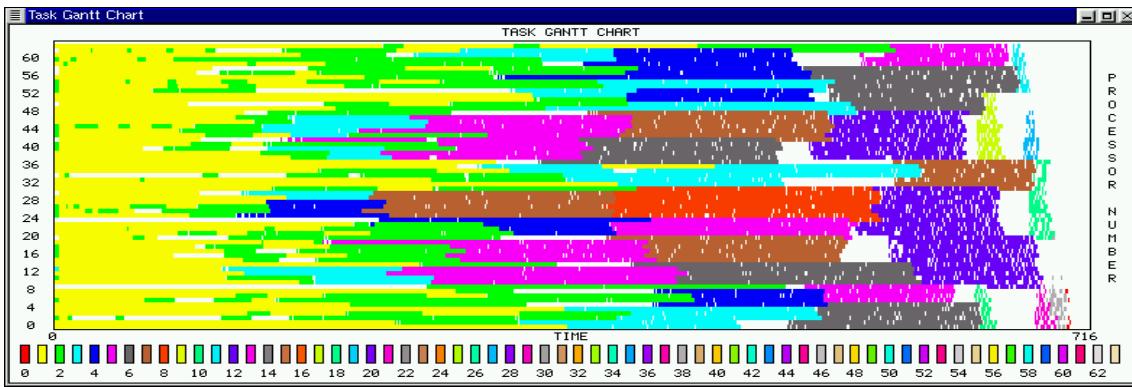


FIG. 7.2: Diagramme de Gantt pour BMWCRA1 sur 64 processeurs.

### 7.3.2 Comparaisons avec PSPASES

Dans cette section, nous présentons une comparaison des performances obtenues entre notre chaîne logicielle PASTIX<sup>1</sup> et la bibliothèque PSPASES<sup>2</sup> version 1.0.3 basée sur une approche multifrontale [55]. PASTIX utilise la version 3.4 de SCOTCH ; PSPASES est configuré pour utiliser la renumérotation de METIS 4.0 [56]. La taille critique des blocs est fixée à 64 et la bibliothèque IBM ESSL est utilisée dans les deux cas.

Les jeux de tests utilisés sont ceux pour lesquels on disposait d'une description au format RSA [25], le format commun accepté par les deux chaînes logicielles. Les mesures données dans la table 7.3 ont été obtenues à l'aide d'une factorisation symbolique *scalaire* pour une renumérotation issue de SCOTCH mais sans amalgamation dans la partie degré minimum : NNZ<sub>A</sub> est le nombre de termes extra-diagonaux dans la partie triangulaire inférieure de la matrice *A*, NNZ<sub>L</sub> est le nombre de termes extra-diagonaux dans la matrice factorisée *L* et OPC est le nombre d'opérations à réaliser.

Une remarque importante pour aider à la comparaison est que PSPASES réalise une factorisation de type Cholesky ( $LL^t$ ), intrinsèquement plus efficace au niveau

<sup>1</sup>Parallel Sparse matriX package.

<sup>2</sup>Parallel SPArse Symmetric dirEct Solver.

des effets BLAS et des effets de cache que la factorisation  $LDL^t$  utilisée par PASTIX. Par exemple, pour une matrice pleine 1024x1024, sur un nœud Power2SC, la routine ESSL  $LL^t$  effectue la factorisation en 1.07s alors que la routine ESSL  $LDL^t$  est réalisée en seulement 1.27s.

Nom	Taille	NNZ <sub>A</sub>	NNZ <sub>L(Scotch)</sub>	OPC <sub>(Scotch)</sub>	NNZ <sub>L(METIS)</sub>	OPC <sub>(METIS)</sub>
<b>B5TUE</b> R	162610	3873534	2.542e+07	1.531e+10	2.404e+07	1.237e+10
<b>BMWCR</b> A1	148770	5247616	6.597e+07	5.702e+10	6.981e+07	6.124e+10
<b>MT</b> 1	97578	4827996	3.115e+07	2.109e+10	3.455e+07	2.269e+10
<b>OILPAN</b>	73752	1761718	8.912e+06	2.985e+09	9.065e+06	2.751e+09
<b>QUER</b>	59122	1403689	9.119e+06	3.281e+09	9.586e+06	3.448e+09
<b>SHIP001</b>	34920	2304655	1.428e+07	9.034e+09	1.481e+07	9.462e+09
<b>SHIP003</b>	121728	3982153	5.873e+07	8.008e+10	5.910e+07	7.587e+10
<b>SHIPSEC5</b>	179860	4966618	5.650e+07	6.952e+10	5.256e+07	5.509e+10
<b>THREAD</b>	29736	2220156	2.404e+07	3.884e+10	2.430e+07	3.583e+10
<b>X104</b>	108384	5029620	2.634e+07	1.713e+10	2.728e+07	1.412e+10

TAB. 7.3: Description des cas tests de comparaison entre PASTIX et PSPASES.

La table 7.4 présente les performances obtenues pour la factorisation parallèle (le temps est donné en secondes et la puissance en gigaflops). Pour chaque matrice, la première ligne donne le temps de PASTIX et la seconde celui de PSPASES. Les tirets correspondent à des exécutions faussées par des phénomènes de “swapping” ou d’allocation mémoire impossible.

Nom	Nombre de processeurs					
	2	4	8	16	32	64
<b>B5TUE</b> R	<b>29.54</b> (0.52)	<b>15.52</b> (0.99)	<b>8.86</b> (1.73)	<b>5.25</b> (2.91)	<b>3.96</b> (3.87)	<b>2.91</b> (5.25)
	-	-	14.04 (0.88)	7.32 (1.69)	3.91 (3.16)	2.58 (4.79)
<b>BMWCR</b> A1	-	-	<b>30.97</b> (1.84)	<b>17.28</b> (3.30)	<b>9.89</b> (5.76)	<b>6.94</b> (8.21)
	-	-	-	-	24.87 (2.49)	13.48 (4.61)
<b>MT</b> 1	<b>37.92</b> (0.56)	<b>20.35</b> (1.04)	<b>11.29</b> (1.87)	<b>6.65</b> (3.17)	<b>4.33</b> (4.87)	<b>3.51</b> (6.01)
	-	-	-	10.71 (2.12)	5.70 (3.98)	3.59 (6.32)
<b>OILPAN</b>	<b>7.28</b> (0.41)	<b>3.81</b> (0.78)	<b>2.15</b> (1.39)	<b>1.39</b> (2.14)	<b>1.00</b> (3.00)	<b>0.87</b> (3.42)
	-	5.23 (0.53)	2.79 (0.99)	1.73 (1.59)	1.25 (2.20)	0.93 (2.96)
<b>QUER</b>	<b>8.35</b> (0.39)	<b>4.46</b> (0.74)	<b>2.57</b> (1.28)	<b>1.67</b> (1.96)	<b>1.16</b> (2.83)	<b>0.93</b> (3.53)
	23.80 (0.14)	13.11 (0.26)	3.22 (1.07)	2.01 (1.72)	1.30 (2.65)	0.96 (3.59)
<b>SHIP001</b>	<b>20.98</b> (0.43)	<b>10.91</b> (0.83)	<b>6.07</b> (1.49)	<b>3.63</b> (2.49)	<b>2.43</b> (3.72)	<b>1.96</b> (4.60)
	-	15.32 (0.62)	8.11 (1.17)	4.48 (2.11)	2.98 (3.17)	2.14 (4.42)
<b>SHIP003</b>	-	<b>109.78</b> (0.73)	<b>45.83</b> (1.75)	<b>25.75</b> (3.11)	<b>16.60</b> (4.83)	<b>12.03</b> (6.66)
	-	-	-	-	21.28 (3.57)	14.08 (5.39)
<b>SHIPSEC5</b>	-	<b>79.12</b> (0.88)	<b>35.68</b> (1.95)	<b>20.51</b> (3.39)	<b>13.99</b> (4.97)	<b>11.58</b> (6.00)
	-	-	-	-	21.80 (2.52)	11.81 (4.66)
<b>THREAD</b>	<b>78.04</b> (0.50)	<b>41.14</b> (0.94)	<b>22.85</b> (1.70)	<b>13.50</b> (2.88)	<b>10.42</b> (3.73)	<b>6.70</b> (5.80)
	-	-	-	21.02 (1.70)	11.24 (3.19)	6.41 (5.59)
<b>X104</b>	<b>31.53</b> (0.54)	<b>19.69</b> (0.87)	<b>9.98</b> (1.72)	<b>7.49</b> (2.29)	<b>5.09</b> (3.37)	<b>3.85</b> (4.44)
	-	-	-	8.32 (1.70)	4.92 (2.87)	3.18 (4.44)

TAB. 7.4: Comparaison des performances entre PASTIX et PSPASES.

Les expérimentations effectuées montrent que PASTIX se compare très favorablement à PSPASES et obtient de meilleurs temps de factorisation dans la plupart des cas jusqu'à 32 processeurs. Pour les plus gros cas tests, la comparaison est favorable à PASTIX jusqu'à 64 processeurs. Pour les cas plus petits, les performances sur 64 processeurs sont sensiblement équivalentes alors que l'on atteint les limites acceptables de la granularité des calculs.

## 7.4 Étude de la scalabilité mémoire

Les problèmes de scalabilité mémoire sont souvent cruciaux pour les applications utilisant les solveurs directs et souvent passés sous silence. Pour des architectures à mémoire distribuée, il est important de pouvoir contrôler l'espace mémoire utilisé par chaque processeur. Il est d'ailleurs intéressant de remarquer qu'à la vue des expérimentations que nous avons menées, on est le plus souvent limité par l'espace mémoire nécessaire pour stocker la matrice plutôt que par le temps mis pour la factoriser. Augmenter le nombre de processeurs n'est alors envisageable qu'à condition d'avoir une relativement bonne scalabilité mémoire.

Comme notre technique de renumérotation (SCOTCH) est basée sur une méthode de type dissection emboîtée qui a tendance à produire des arbres d'élimination équilibrés, et que l'algorithme de partitionnement et de distribution équilibre relativement bien la charge mémoire, la distribution des coefficients de la matrice initiale conduit à une bonne scalabilité mémoire. De plus les structures de données décrivant la matrice et les autres structures de contrôle sont négligeables ; en effet le vecteur de coefficients représente en moyenne plus de 95% de l'espace mémoire requis. Il est de plus important de souligner que ces structures de données sont parfaitement distribuées. Chaque processeur a une vision uniquement locale des données qu'il manipule ; les blocs de contributions distantes contiennent les informations (calculées lors de l'étape de distribution) pour réaliser la communication et la réception chez le processeur destinataire.

Cependant, l'algorithme fan-in que nous avons retenu, conduit à une augmentation de l'espace mémoire nécessaire pour réaliser la factorisation. Grâce à l'allocation dynamique des blocs de contributions agrégées, ce surcoût est réparti au cours de la factorisation. L'approche mixte de distribution 1D/2D engendre une meilleure régulation du surcoût mémoire que l'approche 1D (voir figures 7.3, 7.4, 7.5 et 7.6). Pour ces quatre courbes, on trace le maximum des pourcentages de surcoûts mémoire sur les 16 processeurs en fonction du temps d'exécution (à chaque nouvelle allocation ou désallocation mémoire).

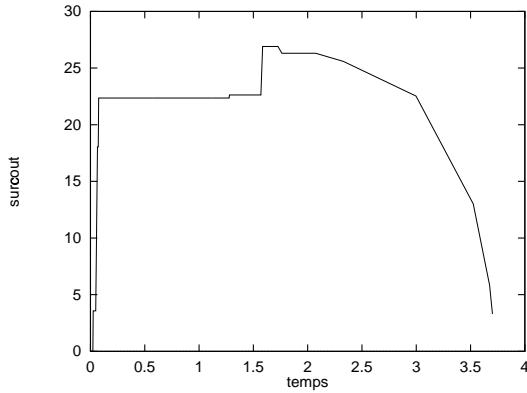


FIG. 7.3: GRID511 : surcoût mémoire sur 16 processeurs avec une distribution 1D.

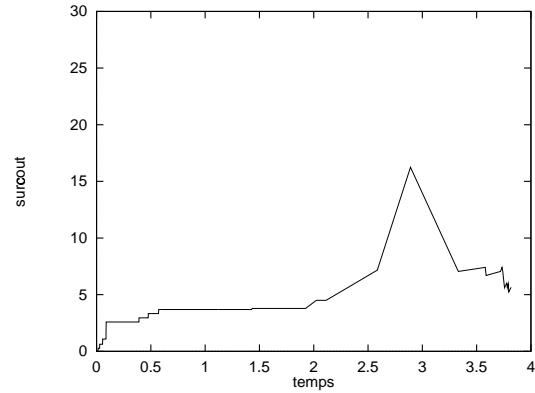


FIG. 7.4: GRID511 : surcoût mémoire sur 16 processeurs avec une distribution 2D.

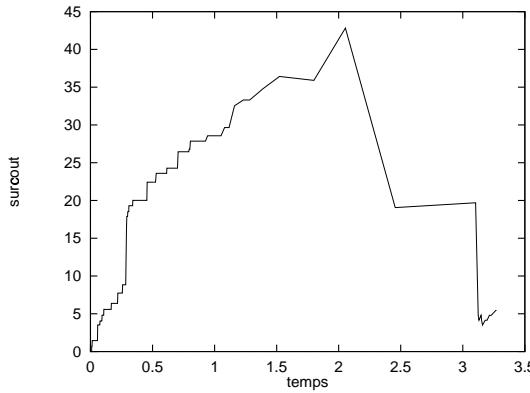


FIG. 7.5: OILPAN : surcoût mémoire sur 16 processeurs avec une distribution 1D.

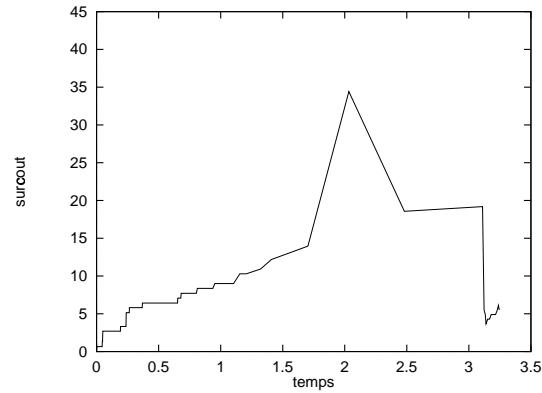


FIG. 7.6: OILPAN : surcoût mémoire sur 16 processeurs avec une distribution 2D.

Le surcoût mémoire reste raisonnable comme le montrent ces expérimentations. Dans le cas où la ressource mémoire est critique, il est également possible d'envoyer ces structures avec une agrégation partiellement achevée afin de libérer de l'espace mémoire, ce qui s'apparente à un schéma de type fan-both [6].

## 7.5 Application au cas d'une matrice pleine

Une autre évaluation intéressante de notre approche consiste à comparer les performances obtenues par notre solveur sur une matrice pleine. On sait qu'une distribution bloc cyclique 2D donne de bons résultats; notre distribution n'étant pas aussi régulière, on peut comparer ces deux techniques.

Lorsque l'on compare les traces d'exécution obtenues avec notre algorithme pour une distribution 1D (voir figure 7.7) et pour une distribution 2D complète (voir figure 7.8), on comprend les problèmes de scalabilité intrinsèques à la distribution 1D. Pour des matrices creuses ce phénomène est moins visible car les blocs pleins sont plus petits et une bonne du parallélisme est induite par le creux (parallélisme de niveau 1). Cependant, une distribution 2D apporte une meilleure scalabilité mais génère aussi une masse de communications plus importantes.

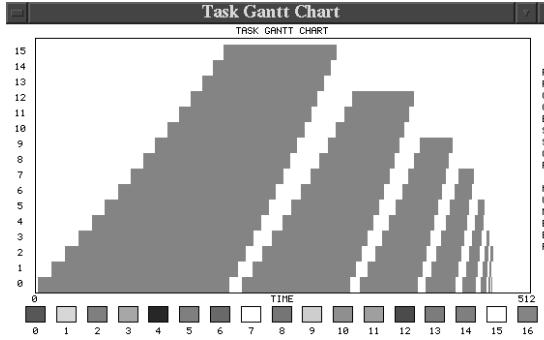


FIG. 7.7: Diagramme de Gantt pour une matrice pleine  $4096 \times 4096$  sur 32 processeurs avec une distribution 1D.

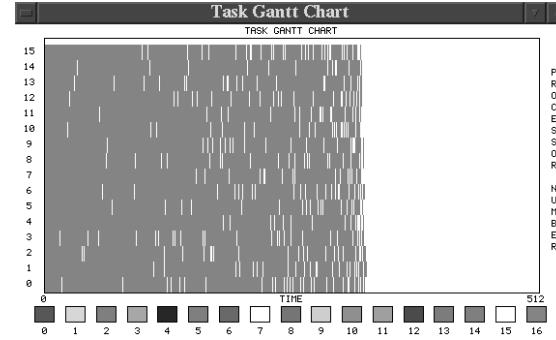


FIG. 7.8: Diagramme de Gantt pour une matrice pleine  $4096 \times 4096$  sur 32 processeurs avec une distribution 2D.

Sur 16 processeurs, pour une matrice de taille  $4096 \times 4096$  pleine, PaStiX réalise une factorisation  $LDL^t$  en 11.2 secondes ; ce qui peut être comparé au temps obtenu par SCALAPACK du domaine public (mais utilisant la bibliothèque IBM ESSL), qui réalise une factorisation  $LL^t$  en 9.5 secondes. Nous pensons qu'il s'agit plutôt d'un bon résultat.

Une perspective intéressante à ce travail consisterait à utiliser notre algorithme de distribution avec le solveur associé, pour implémenter une factorisation sur des matrices pleines dont le partitionnement serait obtenu analytiquement avec les techniques étudiées dans la première partie de cette thèse. En effet, on doit pouvoir écrire analytiquement un découpage bloc irrégulier de la matrice pleine optimisant les effets de recouvrement calcul-communication. L'algorithme de distribution construirait alors un ordonnancement efficace des calculs par bloc, exécutés ensuite par l'algorithme de factorisation.

## 7.6 Conclusion et Perspectives

Dans ce chapitre nous avons montré l'efficacité de notre approche mixte de distribution 1D/2D ; les performances ont été améliorées par rapport au solveur 1D pour la plupart des tests considérés. Les comparaisons avec la bibliothèque PSPASES sont plutôt encourageantes et devront être approfondies. Ce travail est toujours en cours de développement, en particulier pour trouver un critère efficace de basculement entre les distributions 1D et 2D, afin de renforcer encore la scalabilité.

# Bibliographie

- [1] G. Alaghband. Parallel sparse matrix solution and performance. *Parallel Computing*, 21(9), 1407–1430, 1995.
- [2] F. L. Alvarado, A. Pothen, and R. Schreiber. Highly parallel sparse triangular solution. In Alan George, John R. Gilbert, and Joseph W. H. Liu, editors, *Graph Theory and Sparse Matrix Computation*. Springer-Verlag, 1993.
- [3] F. L. Alvarado, and R. Schreiber. Optimal parallel solution of sparse triangular systems. *SIAM J. Scientific Computing*, 14, 446–460, 1993.
- [4] P. Amestoy, T. Davis, and I. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. and Appl.*, 17 :886–905, 1996.
- [5] P. Amestoy, and I. S. Duff. Memory management issues in sparse multifrontal methods on multiprocessors. *Int. J. Supercomputer Applics.*, 7, 64–82, 1993.
- [6] C. Ashcraft. The fan-both family of column-based distributed cholesky factorization algorithms. In Alan George, John R. Gilbert, and Joseph W.-H. Liu, editors, *Graph Theory and Sparse Matrix Computations*. Springer-Verlag, New York, NY, 1993.
- [7] C. Ashcraft, S. C. Eisenstat, and J. W.-H. Liu. A fan-in algorithm for distributed sparse numerical factorization. *SIAM Journal on Scientific and Statistical Computing*, 11 :593–599, 1990.
- [8] C. Ashcraft, S. C. Eisenstat, J. W.-H. Liu, B. W. Peyton, and A. H. Sherman. A compute-ahead implementation of the fan-in sparse distributed factorization scheme. Technical Report ORNL/TM-11496, Oak Ridge National Laboratory, Oak Ridge, TN, 1990.
- [9] C. Ashcraft, S. C. Eisenstat, J. W.-H. Liu, and A. H. Sherman. A comparison of three column based distributed sparse factorization schemes. Technical Report YALEU/DCS/RR-810, Yale University, New Haven, CT, 1990. Also appears in *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, 1991.

- [10] T. Blank, R. F. Lucas, and J. J. Tiemann. A parallel solution method for large sparse systems of equations. *IEEE Transactions on Computer Aided Design, CAD-6(6)* :981–991, November 1987.
- [11] P. Charrier, and J. Roman. Algorithmique et calculs de complexité pour un solveur de type dissections emboîtées. *Numerische Mathematik*, 55, 463–476, 1989.
- [12] P. Charrier, and J. Roman. Study of an adaptive blocking for a parallel nested dissection algorithm. In J. Dongarra, K. Kennedy, P. Messina, D. C. Sorensen, R. G. Voigt, eds, *Proceedings of the 5th SIAM Conference on parallel processing for scientific computing*, Siam Editions, 71–76, 1992.
- [13] P. Charrier, and J. Roman. Partitioning and mapping for parallel nested dissection on distributed memory architectures. In *Proceedings of CONPAR92-VAPP V*, Lectures Notes in computer science, 634, Springer-Verlag, 295–306, 1992.
- [14] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. LAPACK Working Note : ScaLAPACK : A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performances. Technical Report 95, Department of Computer Science - University of Tennessee, 1995.
- [15] J. M. Conroy. Parallel nested dissection. *Parallel Computing*, 16 :139–156, 1990.
- [16] J. M. Conroy, S. G. Kratzer, and R. F. Lucas. Multifrontal sparse solvers in message passing and data parallel environments - a comparative study. In *Proceedings of PARCO*, 1993.
- [17] J. M. Conroy, S. G. Kratzer, and R. F. Lucas. Data-parallel sparse matrix factorization. In J. G. Lewis, ed., *Proceedings 5th SIAM Conference on Linear Algebra*, SIAM Press, Philadelphia, 377–381, 1994.
- [18] J.W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. Technical Report CSL-94-14, Xerox Palo Alto Research Center, 1995.
- [19] J.W. Demmel, J.R. Gilbert and X.S. Li. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. Technical Report, University of California, Berkeley, february 1997.
- [20] J.J. Dongarra, I. S. Duff, J. DuCroz, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprogramms. *ACM Trans. on Math. Soft.*, 16,1 :1-28, 1990.
- [21] I. S. Duff. Parallel implementation of multifrontal schemes. *Parallel Computing*, 3 :193–204, 1986.

- [22] I.S. Duff. A review of frontal methods for solving linear systems. *Computer Physics Communications*, 1996.
- [23] I. S. Duff. Sparse numerical linear algebra : direct methods and preconditioning. Technical Report TR/PA/96/22, CERFACS, 1996.
- [24] I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Transactions on Mathematical Software*, 15 :1–14, 1989.
- [25] I. S. Duff, R. G. Grimes, and J. G. Lewis. Users' guide for the Harwell-Boeing sparse matrix collection (release I). Technical Report TR/PA/92/86, Research and Technology Division, Boeing Computer Services, Seattle, WA, 1992.
- [26] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software*, (9) :302–325, 1983.
- [27] I. S. Duff and J. K. Reid. The multifrontal solution of unsymmetric sets of linear equations. *SIAM Journal on Scientific and Statistical Computing*, 5(3) :633–641, 1984.
- [28] S. C. Eisenstat and J. W. H. Liu. Exploiting structural symmetry in a sparse partial pivoting code. *SIAM J. Sci. Comput.*, 14 :253–257, 1993.
- [29] K. Eswar, P. Sadayappan, and V. Visvanathan. Supernodal sparse Cholesky factorization on distributed-memory multiprocessors. In *International Conference on Parallel Processing*, pages 18–22 (vol. 3), 1993.
- [30] L. Facq, and 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, and B. Virot, eds, *Parallélisme et applications irrégulières*, 135–147, Hermès, 1995.
- [31] K. A. Gallivan et al. *Parallel Algorithms for Matrix Computations*. SIAM, Philadelphia, PA, 1990.
- [32] K. A. Gallivan, R. J. Plemmons, and A. H. Sameh. Parallel algorithms for dense linear algebra computations. *SIAM Review*, 32(1) :54–135, March 1990. Also appears in K. A. Gallivan et al. *Parallel Algorithms for Matrix Computations*. SIAM, Philadelphia, PA, 1990.
- [33] M. Garey and D. S. Johnson. *Computers and Intractability*. Freeman, San Francisco, CA, 1979.
- [34] G. A. Geist and M. T. Heath. Parallel Cholesky factorization on a hypercube multiprocessor. Technical Report ORNL-6190, Oak Ridge National Laboratory, Oak Ridge, TN, 1985.

- [35] G. A. Geist and M. T. Heath. Matrix factorization on a hypercube multiprocessor. In M. T. Heath, editor, *Hypercube Multiprocessors 1986*, pages 161–180. SIAM, Philadelphia, PA, 1986.
- [36] G. A. Geist and E. G.-Y. Ng. Task scheduling for parallel sparse Cholesky factorization. *Internation Journal of Parallel Programming*, 18(4) :291–314, 1989.
- [37] A. George, J. R. Gilbert, and J. W. H. Liu. *Graph Theory and Sparse Matrix Computation*. Springer-Verlag, IMA vol 56, 1993.
- [38] A. George, M. T. Heath, J. W.-H. Liu, and E. G.-Y. Ng. Sparse Cholesky factorization on a local memory multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 9 :327–340, 1988.
- [39] A. George, M. T. Heath, J. W.-H. Liu, and E. G.-Y. Ng. Solution of sparse positive definite systems on a hypercube. *Journal of Computational and Applied Mathematics*, 27 :129–156, 1989. Also available as Technical Report ORNL/TM-10865, Oak Ridge National Laboratory, Oak Ridge, TN, 1988.
- [40] A. George and J. W.-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.
- [41] A. George, J. W.-H. Liu, and E. G.-Y. Ng. Communication reduction in parallel sparse Cholesky factorization on a hypercube. In M. T. Heath, editor, *Hypercube Multiprocessors 1987*, pages 576–586. SIAM, Philadelphia, PA, 1987.
- [42] A. George, J. W.-H. Liu, and E. G.-Y. Ng. Communication results for parallel sparse Cholesky factorization on a hypercube. *Parallel Computing*, 10(3) :287–298, May 1989.
- [43] J. R. Gilbert and R. Schreiber. Highly parallel sparse Cholesky factorization. *SIAM Journal on Scientific and Statistical Computing*, 13 :1151–1172, 1992.
- [44] A. Grama, A. Gupta, V. Kumar, and G. Karypis. *Introduction to Parallel Computing : Design and Analysis of Algorithms*. Benjamin/Cummings Publishing Company, Redwood City, CA, 1994.
- [45] A. Gupta, G. Karypis, and V. Kumar. Highly scalable parallel algorithms for sparse matrix factorization. Technical Report 94-63, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1994. To appear in *IEEE Transactions on Parallel and Distributed Systems*, 1997. Postscript file available via anonymous FTP from the site <ftp://ftp.cs.umn.edu/users/kumar>.
- [46] A. Gupta and V. Kumar. A scalable parallel algorithm for sparse matrix factorization. Technical Report 94-19, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1994. A short version appears in *Supercomputing '94 Proceedings*. TR available in *users/kumar* at anonymous FTP site em <ftp.cs.umn.edu>.

- [47] A. Gupta and V. Kumar. Parallel algorithms for forward and back substitution in direct solution of sparse linear systems. In *Supercomputing '95 Proceedings*, December 1995.
- [48] A. Gupta and E. Rothberg. An evaluation of left-looking, right-looking, and multifrontal approaches to sparse Cholesky factorization on hierarchical-memory machines. *Int. J. High Speed Comput.*, 5 :537-593, 1993.
- [49] A. Gupta and E. Rothberg. An efficient block-oriented approach to parallel sparse Cholesky factorization. *SIAM J. Scientific Computing*, 15 :6, 1413–1439. Also appears in *Supercomputing '93 Proceedings*, 1993.
- [50] M. T. Heath, E. G.-Y. Ng, and B. W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33 :420–460, 1991. Also appears in K. A. Gallivan et al. *Parallel Algorithms for Matrix Computations*. SIAM, Philadelphia, PA, 1990.
- [51] M. T. Heath and P. Raghavan. Performance of a fully parallel sparse solver. In Scalable High Performance Computing Conference, Los Alamitos, CA, 1994. IEEE Computer Society Press. Submit to *Int. J. of Supercomputer Appl.*
- [52] M. T. Heath and C. H. Romine. Parallel solution of triangular systems on distributed-memory multiprocessors. *SIAM Journal on Scientific and Statistical Computing*, 9(3) :558–588, 1988.
- [53] P. Hénon, P. Ramet, and J. Roman. A Mapping and Scheduling Algorithm for Parallel Sparse Fan-In Numerical Factorization. In P. Amestoy, P. Berger, M. Daydé, I. Duff, V. Frayssé, L. Giraud, and D. Ruiz, editors, *EuroPAR'99, LNCS 1685*, pages 1059–1067. Springer Verlag, 1999.
- [54] P. Hénon, P. Ramet, and J. Roman. PaStiX : A Parallel Sparse Direct Solver Based on a Static Scheduling for Mixed 1D/2D Block Distributions. Soumis à Irregular'2000.
- [55] M. Joshi, G. Karypis, V. Kumar, A. Gupta, and Gustavson F. PSPASES : Scalable Parallel Direct Solver Library for Sparse Symmetric Positive Definite Linear Systems. Technical report, University of Minnesota and IBM Thomas J. Watson Research Center, May 1999.
- [56] G. Karypis, V. Kumar. METIS – A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices – Version 4.0. University of Minnesota, September 1998.
- [57] J. W.-H. Liu. Computational models and task scheduling for parallel sparse Cholesky factorization. *Parallel Computing*, 3 :327–342, 1986.
- [58] J. W.-H. Liu. The role of elimination trees in sparse factorization. *SIAM J. Matrix Anal. Appl.*, 11 :134-172, 1990.

- [59] J. W.-H. Liu. The multifrontal method for sparse matrix solution : Theory and practice. Technical Report CS-90-04, York University, Ontario, Canada, 1990. Also appears in *SIAM Review*, 34 :82–109, 1992.
- [60] R. F. Lucas. *Solving planar systems of equations on distributed-memory multiprocessors*. PhD thesis, Department of Electrical Engineering, Stanford University, Palo Alto, CA, 1987. Also see *IEEE Transactions on Computer Aided Design*, 6 :981–991, 1987.
- [61] F. Manne. *Load Balancing in Parallel Sparse Matrix Computations*. PhD thesis, University of Bergen, Norway, 1993.
- [62] J. Michel, F. Pellegrini and J. Roman. Unstructured graph partitioning for sparse linear system solving. In *Proc. IRREGULAR'97, Paderborn*, LNCS, Springer Verlag, june 1997.
- [63] E. Ng and B. W. Peyton. Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM J. Sci. Comput.*, 14 :1034–1056, 1993.
- [64] F. Pellegrini and J. Roman. Sparse matrix ordering with SCOTCH. In *Proceedings of HPCN'97, Vienna, LNCS 1225*, pages 370–378, April 1997.
- [65] F. Pellegrini, J. Roman, and P. Amestoy. Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering. In *Proceedings of IRREGULAR'99, Puerto Rico, LNCS 1586*, pages 986–995, April 1999. Extended version to be published in *Concurrency : Practice and Experience*.
- [66] A. Pothen and C. Sun. Distributed multifrontal factorization using clique trees. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, pages 34–40, 1991.
- [67] A. Pothen, and C. Sun. A mapping algorithm for parallel sparse Cholesky factorization. *SIAM J. Scientific Computing*, 14(5), 1253–1257, 1993.
- [68] R. Pozo and S. L. Smith. Performance evaluation of the parallel multifrontal method in a distributed-memory environment. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 453–456, 1993.
- [69] J. K. Reid. *Sparse Matrices*. In *The State of the Art in Numerical Analysis*, A. Iserles and M. J. D. Powell eds, Oxford University Press, pages 59–85, 1987.
- [70] J. Roman. Calculs de complexité relatifs à une méthode de dissection emboîtée. *Numerische Mathematik*, 47, 175–190, 1985.
- [71] 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.

- [72] E. Rothberg. *Exploiting the memory hierarchy in sequential and parallel sparse Cholesky factorization*. PhD thesis, Stanford University, Palo Alto, CA, 1993.
- [73] E. Rothberg. Performance of panel and block approaches to sparse Cholesky factorization on the iPSC/860 and Paragon multicomputers. *SIAM J. Scientific Computing*, 17 :3, 699–713, 1996. Also appears in *Proceedings of the 1994 Scalable High Performance Computing Conference*, May 1994.
- [74] E. Rothberg and A. Gupta. An efficient block-oriented approach to parallel sparse Cholesky factorization. *SIAM J. Sci. Comput.*, 15(6) :1413–1439, November 1994.
- [75] E. Rothberg and R. Schreiber. Improved load distribution in parallel sparse Cholesky factorization. In *Supercomputing '94 Proceedings*, 1994.
- [76] R. Schreiber. Scalability of sparse direct solvers. Technical Report RIACS TR 92.13, NASA Ames Research Center, Moffet Field, CA, May 1992. Also appears in A. George, John R. Gilbert, and J. W.-H. Liu, editors, *Sparse Matrix Computations : Graph Theory Issues and Algorithms* (An IMA Workshop Volume). Springer-Verlag, New York, NY, 1993.



# Conclusion générale et perspectives

Dans la première partie de cette thèse nous avons présenté une contribution concernant la problématique du recouvrement calcul/communication. Nous avons tout d'abord présenté un tour d'horizon assez large sur les techniques de recouvrement. Nous avons proposé un modèle ainsi qu'un schéma de résolution permettant de déterminer analytiquement ou numériquement la taille optimale de paquets qui maximise le recouvrement dans les algorithmes macro-pipelines réguliers. Nous avons ensuite proposé un calcul de la suite de tailles de paquets qui optimise le recouvrement pour le problème de la factorisation de Cholesky d'une matrice symétrique, la prise en compte de la symétrie constituant une première étape vers la prise en compte des problèmes irréguliers. Des expérimentations ont été effectuées sur un Paragon Intel et un SP2 IBM pour valider cette étude sur des algorithmes classiques d'algèbre linéaire dense (factorisation *LU* et méthode SOR). Des gains significatifs ont été obtenus par l'utilisation de la bibliothèque OPIUM permettant le calcul de la taille optimale de paquets connaissant la complexité des calculs encadrant les communications et le modèle de communication.

Dans la seconde partie de cette thèse, nous avons présenté un algorithme performant de distribution et d'ordonnancement des calculs conduisant à une bonne régulation et au masquage quasi-total des communications pour un solveur parallèle direct par bloc de grands systèmes linéaires creux de matrices symétriques définies positives. Nous avons décrit notre approche qui dissocie les phases de repartitionnement et de distribution. Le repartitionnement, nécessaire pour exploiter le parallélisme induit par les calculs dans les blocs denses de la matrice creuse, est basé sur une stratégie de descente récursive de l'arbre d'élimination ; la distribution et l'ordonnancement des calculs reposent sur une simulation de la factorisation parallèle. Nous avons tout d'abord proposé une approche basée sur une distribution 1D que nous avons ensuite étendue à des distributions mixtes 1D/2D offrant une meilleure scalabilité. L'intérêt d'utiliser une telle distribution mixte a été mise en évidence expérimentalement ; cela a fait apparaître l'existence d'un seuil variable à partir duquel il faut passer d'une distribution 1D à une distribution 2D. Les performances obtenues et analysées sur un IBM SP2, et pour un jeu de 33 problèmes tests

en grande partie issus d'applications industrielles, se comparent très favorablement aux meilleurs solveurs parallèles directs actuels, utilisés dans le même cadre.

A court et moyen terme, nous comptons poursuivre nos recherches sur les techniques de recouvrement calcul/communication pour les problèmes irréguliers. En particulier, ceci concerne, pour des matrices symétriques denses, le calcul d'une partition irrégulière adaptative engendrant un macro-pipeline bidimensionnel et maximisant le recouvrement calcul/communication lors de la factorisation de Cholesky pour une distribution blocs 2D. Il sera alors intéressant d'utiliser les techniques présentées dans la seconde partie de cette thèse pour construire un ordonnancement compatible avec cette partition, et faire ainsi une étude comparative des performances du solveur associé avec celles de ScaLAPACK. Ce travail pourra alors être étendu aux calculs sur les blocs denses d'une matrice creuse structurée par blocs ; l'objectif visé sera alors d'exploiter aussi ces techniques de recouvrement calcul/communication pour des macro-pipelines intervenant dans les solveurs traitant les matrices creuses.

En ce qui concerne l'ordonnancement des calculs sur les matrices creuses, nous travaillons également à définir et valider des critères efficaces pour "basculer" entre une distribution 1D et une distribution 2D. En particulier, une étude algorithmique prenant en compte la taille des séparateurs correspondant aux blocs les plus gros (dans la partie haute de l'arbre d'élimination), ainsi que le nombre de processeurs qui doivent participer au calcul est à finaliser.

D'autre part, l'émergence de nouvelles architectures parallèles à base de nœuds SMP reliés par un réseau d'interconnexion, comme par exemple la SP3 IBM, motive également la généralisation de ces techniques. Notre approche devra donc être généralisée pour construire une distribution adéquate prenant en compte l'hétérogénéité de l'architecture. Cette extension devrait être possible en modélisant les 2 niveaux d'accès aux données (distants pour le réseau et locaux en mémoire partagée), et en utilisant ces modèles dans la seconde phase de l'algorithme de repartitionnement/distribution.

Enfin, à plus long terme, et pour traiter des systèmes encore plus grands (problèmes d'electromagnétisme étudiés par le CEA nécessitant plus de  $10^6$  inconnues pour des maillages 3D), il sera intéressant d'appliquer ces techniques dans le cadre de la mise en place de méthodes hybrides de résolution de systèmes linéaires creux couplant des solveurs directs et itératifs.

## **Annexes**



# Annexe A

## Fonctions de complexité

### A.1 Algorithme de Cholesky-CROUT en dense

On décrit dans cette annexe les fonctions de coût attachées aux différentes phases de la factorisation  $LDL^t$  et utilisées dans les chapitres 6 et 7.

Soit  $N$  le nombre total de blocs-colonnes de la partition ; pour chaque bloc  $k$  on note  $n_k$  son nombre de colonnes.

On pose également  $h_l = \sum_{i=l+1}^N n_i$  et  $\tilde{h}_l = \sum_{i=l}^N n_i$ , pour  $1 \leq l \leq N$ .

On considère alors l'algorithme suivant :

Pour  $k := 1$  à  $N$  faire

Factoriser  $A_{kk}$  en  $L_{kk}D_kL_{kk}^t$ ; (Phase 1)

Pour  $j := k + 1$  à  $N$  faire (Phase 2)

Résoudre  $L_{kk}F_j^t = A_{jk}^t$ ;

Résoudre  $D_kL_{jk}^t = F_j^t$ ;

Pour  $j := k + 1$  à  $N$  faire (Phase 3)

Pour  $i := j$  à  $N$  faire

$$A_{ij} = A_{ij} - L_{ik}F_j^t;$$

La phase 3 doit être considérée en deux étapes celles-ci ne s'exécutant pas toujours sur les mêmes processeurs :

- l'étape de calcul de la contribution,
- l'étape de mise à jour dans la matrice.

## Phase 1

Pour chaque étape  $k$ , on a

$$C_1(k) = \frac{n_k^3}{3} + \frac{n_k^2}{2} - \frac{5n_k}{6}.$$

## Phase 2

Pour chaque étape  $k$ , on a

$$C_2(k) = \sum_{j=k+1}^N \left[ \underbrace{2n_j \frac{n_k(n_k-1)}{2}}_{L_{kk}A_{jk}^t = A_{jk}^t} + \underbrace{n_k n_j}_{D_k L_{jk}^t = A_{jk}^t} \right]$$

soit  $C_2(k) = h_k n_k^2$ .

## Phase 3

Pour chaque étape  $k$ , on a

$$C_3(k) = \sum_{j=k+1}^N \left[ \sum_{i=j}^N \left( \underbrace{2n_i n_j n_k}_{L_{ik}F_j^t} \right) + \underbrace{\frac{n_j(n_j+1)}{2}}_{A_{jj}=A_{jj}-\bullet} + \sum_{i=j+1}^N \underbrace{n_i n_j}_{A_{ij}=A_{ij}-\bullet} \right]$$

soit  $C_3(k) = \sum_{j=k+1}^N \left[ 2n_k n_j \tilde{h}_j + \frac{n_j(n_j+1)}{2} + n_j h_j \right]$ .

## Coût total

$$C_{dense} = \sum_{k=1}^N C_1(k) + C_2(k) + C_3(k).$$

## A.2 Algorithme de Cholesky-CROUT en creux

Considérons la structure creuse par blocs issue de la factorisation symbolique. Pour un bloc-colonne  $k$ ,  $1 \leq k \leq N$ , on note  $l_k$  sa taille, et  $s_k$  son nombre de blocs extra-diagonaux. Pour le  $p^{\text{ième}}$  bloc extra-diagonal du bloc-colonne  $k$  on note  $r_{kp}$  son nombre de ligne et,  $n(k, p)$  le numéro du bloc-colonne “en regard”.

$$\text{On a alors } \begin{cases} BStruct(L_{k*}) = \{j < k / \exists p, 1 \leq p \leq s_j \quad tq \quad n(j, p) = k\} \\ BStruct(L_{*k}) = \{i > k / \exists p, 1 \leq p \leq s_k \quad tq \quad n(k, p) = i\} \end{cases}$$

où  $BStruct(L_{k*})$  est l'ensemble des blocs-colonnes qui modifient le bloc-colonne  $k$ , et  $BStruct(L_{*k})$  est l'ensemble des blocs-colonnes modifiés par le bloc-colonne  $k$ .

On note ensuite :

$$\left\{ \begin{array}{l} \forall j \in BStruct(L_{k*}), h_{jk} = \sum_{p/n(j,p)=k} r_{jp} \quad \text{et} \quad \overline{h_{jk}} = \sum_{t \in BStruct(L_{*j}), t>k} h_{jt} \\ \forall i \in BStruct(L_{*k}), h_{ki} = \sum_{p/n(k,p)=i} r_{kp} \quad \text{et} \quad \overline{h_{ki}} = \sum_{t \in BStruct(L_{*k}), t>i} h_{kt} \\ g_{ki} = h_{ki} + \overline{h_{ki}} \\ g_k = \sum_{i \in BStruct(L_{*k})} h_{ki} \end{array} \right.$$

En utilisant ces notations et en adaptant les fonctions de complexité rappelées pour le cas dense, on obtient pour le cas creux et pour une étape  $k$  donnée :

– le coût de la mise à jour de la contribution du bloc-colonne  $j \in BStruct(L_{k*})$  :

$$C_3''(j, k) = \frac{h_{jk}(h_{jk} + 1)}{2} + h_{jk}\overline{h_{jk}} \quad (\text{Phase 3})$$

– le coût du calcul local :

$$C_1(k) + C_2(k) = \frac{l_k^3}{3} + \frac{l_k^2}{2} - \frac{5}{6}l_k + g_k l_k^2 + l_k \quad (\text{Phase 1 et 2})$$

– le coût du calcul de la contribution pour le bloc  $i \in BStruct(L_{*k})$  :

$$C_3'(k, i) = 2 l_k g_{ki} h_{ki} \quad (\text{Phase 3})$$

– le coût d'échange de données avec le bloc  $i \in BStruct(L_{*k})$  :

$$\frac{h_{ki}(h_{ki} + 1)}{2} + h_{ki}\overline{h_{ki}}$$

Le nombre total d'opérations pour la factorisation en creux sera donné par :

$$C_{creux} = \sum_{k=1}^N \left[ \sum_{j \in BStruct(L_{k*})} [C_3''(j, k)] + C_1(k) + C_2(k) + \sum_{i \in BStruct(L_{*k})} [C_3'(k, i)] \right].$$

D'autre part, le nombre d'opérations pour la descente-remontée sera donné par :

$$\sum_{k=1}^N l_k^2 + 4l_k g_k.$$

## Annexe B

# Intégration dans une chaîne logicielle

Les algorithmes présentés dans la deuxième partie de cette thèse ont été intégrés dans une chaîne logicielle complète pour la résolution parallèle directe de grands systèmes linéaires creux. On y trouve les trois étapes de pré-traitement :

- le partitionnement et la renumérotation des inconnues ;
- la factorisation symbolique par blocs ;
- le repartitionnement et la distribution des calculs par blocs ;

ainsi que les étapes parallèles d'assemblage et de factorisation/résolution.

Cette chaîne logicielle est en cours d'évaluation sur un code industriel vectorisé de mécanique des structures, non linéaire en temps et en deux ou trois dimensions d'espace. C'est un code de calcul de structure par éléments finis qui résout les équations de la mécanique des milieux continus en statique ou en dynamique (quand les effets d'inertie sont négligeables) ; il calcule donc la réponse d'une structure à diverses sollicitations (forces ponctuelles, pressions, dilatations thermiques, accélérations diverses). La réponse de la structure est caractérisée par les déplacements aux nœuds de la discrétisation par éléments finis ; ce sont ces déplacements qui constituent le vecteur inconnu du système à résoudre  $Ku = F$ , le chargement constituant le second membre de ce système. A partir du déplacement  $u$ , on calcule les contraintes dans la structure ; dans le cas d'apparition de grands déplacements ou d'une plastification, le problème devient non linéaire (on peut pour simplifier considérer que la matrice  $K$  de l'itération  $p + 1$  est fonction de  $u$  calculé à l'itération  $p$ ). Les contraintes dans une structure permettent de prévoir la destruction éventuelle de celle-ci.

Une autre application en projet de notre chaîne logicielle est celle un code d'elettromagnétisme ; c'est ce type d'application qui nous a conduit à choisir une factorisation  $LDL^t$  permettant de traiter des matrices à coefficients complexes.

La figure B.1 présente l'enchaînement des différentes étapes liées à la résolution parallèle directe pour l'application de mécanique des structures.

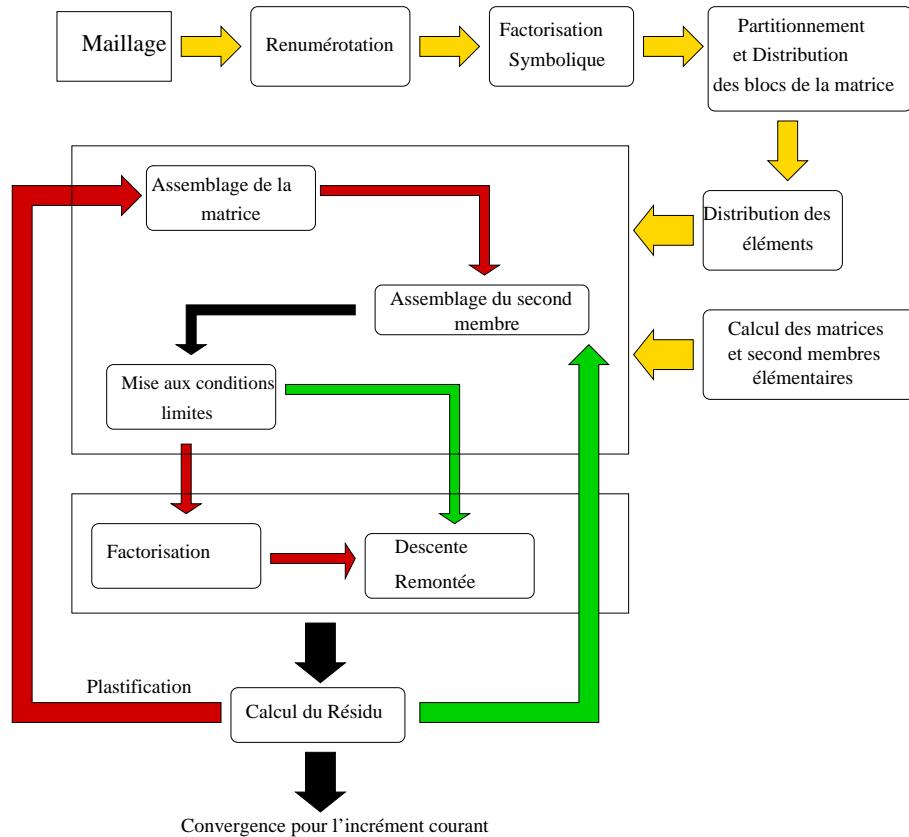


FIG. B.1: Enchaînement des différentes fonctionnalités.

Dans les deux sections suivantes nous présentons un module ne faisant pas partie de la chaîne de calcul ; son rôle est d'étonner l'architecture cible. Lors de l'installation du logiciel sur une nouvelle architecture, ce module va effectuer un certains nombres de tests :

- des test de calculs qui ont pour but d'évaluer les performances des routines de calcul vectoriel BLAS et de recopie mémoire, en particulier celles utilisées lors de la factorisation, à savoir les routines COPY, SCAL, GEMM, TRSM, PPF (factorisation de Crout d'un bloc diagonal plein spécifique à la bibliothèque ESSL sur SP2/IBM) ;
- des tests de communications qui ont pour but d'évaluer les performances du système de communication, en particulier le temps de latence et la bande passante des primitives de communication point-à-point MPI, pour les modes synchrones et asynchrones.

Le résultat de ce module est la construction d'un fichier contenant les constantes caractéristiques de l'architecture cible et qui sera donc utilisé lors de l'étape de repartitionnement et de distribution de la matrice (chapitre 6 et 7).

## B.1 Modélisation expérimentale des BLAS

Dans cette section nous décrivons les valeurs des paramètres obtenues sur la machine IBM SP2 du CINES (basée sur des nœuds 120 MHz Power2SC). Pour toutes les routines BLAS considérées, les paramètres  $i, j$  et  $k$  correspondent à des nombres de données double précision. On note  $\Omega_{OPT}$  le temps associé à l'opération  $OPT$ .

Pour les routines BLAS 1, on utilise une modélisation de type fonction affine par morceaux :

$$\begin{aligned} - \text{COPY : } & \begin{cases} \Omega_{COPY}(i) = 10^{-10} & \text{si } i < 60 \\ \Omega_{COPY}(i) = 10^{-7} \cdot (i - 60)/60 & \text{si } 60 \leq i < 130 \\ \Omega_{COPY}(i) = 5 \cdot 10^{-9} \cdot i - 5 \cdot 10^{-8} & \text{si } i \geq 130 \end{cases} \\ - \text{SCAL : } & \begin{cases} \Omega_{SCAL}(i) = 5 \cdot 10^{-10} & \text{si } i < 100 \\ \Omega_{SCAL}(i) = 5 \cdot 10^{-9} \cdot i & \text{si } i \geq 100 \end{cases} \end{aligned}$$

Pour modéliser la courbe d'efficacité des routines BLAS 3, on utilise pour le moment le moteur de calcul “Maple” pour générer une régression polynomiale multivariables :

$$\begin{aligned} - \text{GEMM : } & \Omega_{GEMM}(i, j, k) = a \cdot ijk + b \cdot ij + c \cdot ik + d \cdot jk + e \cdot i + f \cdot j + g \cdot k + h \\ & \text{avec } \begin{cases} a = 0.42 \cdot 10^{-8} \\ b = 0.83 \cdot 10^{-7} \\ c = 0.79 \cdot 10^{-7} \end{cases}, \begin{cases} d = 0.70 \cdot 10^{-7} \\ e = -0.16 \cdot 10^{-5} \\ f = -0.11 \cdot 10^{-5} \end{cases} \text{ et } \begin{cases} g = -0.66 \cdot 10^{-6} \\ h = 0.31 \cdot 10^{-4} \end{cases} \\ - \text{TRSM : } & \Omega_{TRSM}(i, j) = a \cdot i^2 j + b \cdot i^2 + c \cdot ij + d \cdot i + e \cdot j + f \\ & \text{avec } \begin{cases} a = 0.41 \cdot 10^{-8} \\ b = 0.82 \cdot 10^{-7} \end{cases}, \text{ et } \begin{cases} c = -0.23 \cdot 10^{-7} \\ d = -0.59 \cdot 10^{-5} \end{cases} \text{ et } \begin{cases} e = 0.57 \cdot 10^{-6} \\ f = 0.90 \cdot 10^{-4} \end{cases} \\ - \text{PPF : } & \Omega_{PPF}(i) = 6.17 \cdot 10^{-9} \cdot (2 \cdot i^3 + 3 \cdot i^2 - 5 \cdot i)/6 \end{aligned}$$

## B.2 Modélisation expérimentale des routines de communications

Dans cette section, nous présentons les résultats des expérimentations effectuées pour modéliser les routines de communication. Pour cela, nous mesurons le temps, en moyenne, d'un “ping-pong” entre 2 processeurs en fonction de la taille des messages

(réels simple précision). Entre chaque échange, les processeurs effectuent un calcul élémentaire de coût 0.18s sur le SP2 IBM du LaBRI basé sur des nœuds Power2 à 66Mhz, et de coût 0.1s sur le SP2 IBM du CINES basé sur des nœuds Power2SC à 120Mhz. Les prises de temps sont effectuées pour les routines MPI bloquantes et non bloquantes.

Le calculateur SP2 IBM dispose de deux modes de communications. Pour des messages de tailles inférieures à 4Ko un protocole de type “envoie immédiat” est utilisé (dans ce cas, les *buffers* MPI sont dimensionnés pour garantir un espace mémoire pour les réceptions). Un protocole de type “rendez vous” est utilisé pour des messages de tailles supérieures à 4Ko. De plus, l’implémentation de MPI permet l’activation d’interruptions pour forcer les communications pendant les phases de calcul dans le cas du protocole de type “rendez vous”. Les figures ci-dessous présentent les résultats obtenus avec et sans activation de ces interruptions.

Pour chaque courbes, on présente les temps obtenus pour les communications bloquante et non bloquante ainsi que la courbe que l’on obtiendrait si le recouvrement était total.

L’architecture la plus favorable pour exploiter le recouvrement est celui du couple TB3/Power2SC (figure B.4) pour laquelle il reste environ 50% de CPU avec des tailles de messages inférieures à 4Ko ; ce qui correspond à ce qui est annoncé dans la documentation du constructeur. Nous remarquons également que les interruptions engendrent un surcoût mesurable. Enfin, l’allure des courbes montrent que la modélisation du comportement de la communication par une fonction affine est justifiée.

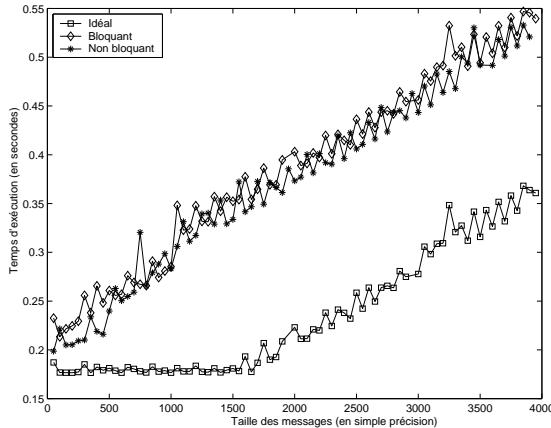


FIG. B.2: SP2 LaBRI sans interruptions.

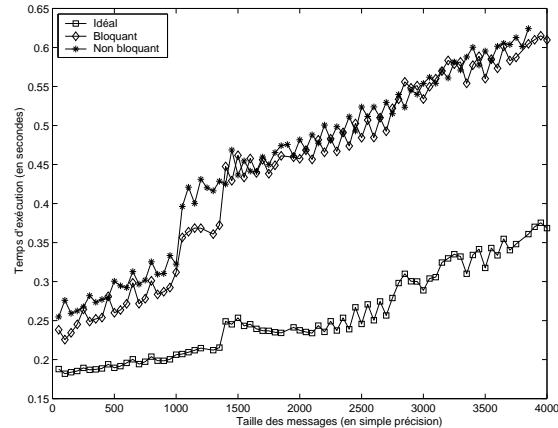


FIG. B.3: SP2 LaBRI avec interruptions.

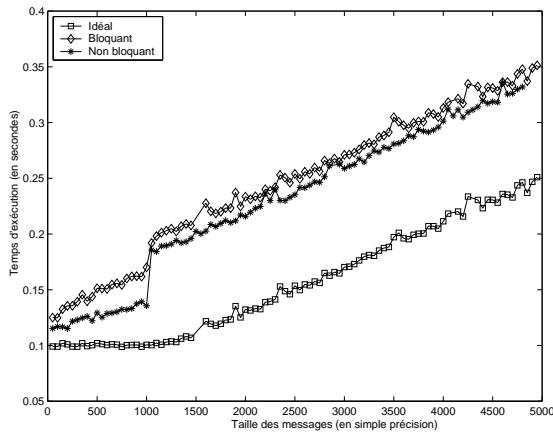


FIG. B.4: SP2 CINES sans interruptions.

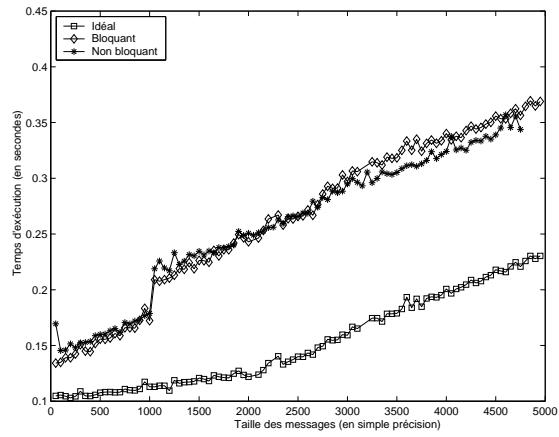


FIG. B.5: SP2 CINES avec interruptions.



## Annexe C

# Liste des publications de l'auteur

### Revues internationales avec comité de lecture :

- [1] D. Goudin, P. Hénon, F. Pellegrini, P. Ramet, J. Roman et J.-J. Pesqué. *Parallel Sparse Linear Algebra and Application to Structural Mechanics*, accepté dans Numerical Algorithms, Baltzer Science Publisher, 2000, 25 pages.

### Congrès internationaux avec comité de sélection et avec actes :

- [2] F. Desprez, P. Ramet et J. Roman. *Optimal Grain Size Computation for Pipelined Algorithms*. Actes de EuroPar'96, Lyon, France, septembre 1996, LNCS 1123, pages 165-172. Springer Verlag.
- [3] P. Hénon, P. Ramet et J. Roman. *A Mapping and Scheduling Algorithm for Parallel Sparse Fan-In Numerical Factorization*. Actes de EuroPar'99, Toulouse, France, septembre 1999, LNCS 1685, pages 1059-1067. Springer Verlag.
- [4] P. Hénon, P. Ramet et J. Roman. *PaStiX : A Parallel Sparse Direct Solver Based on a Static Scheduling for Mixed 1D/2D Block Distributions*. Accepté à Irregular'2000, Cancun, Mexique, mai 2000, à paraître dans LNCS.

### Congrès nationaux avec comité de sélection et avec actes :

- [5] P. Ramet. *Calcul de la Taille Optimale des Paquets pour les Algorithmes Macro-Pipelines*. RenPar'8, Bordeaux, France, juin 1996, pages 21-24.
- [6] P. Ramet. *Calcul de la Suite Optimale de Taille de Paquets pour la Factorisation de Cholesky*. RenPar'9, Lausanne, Suisse, mai 1997, pages 111-114.
- [7] D. Goudin, P. Hénon, F. Pellegrini, P. Ramet, J. Roman et J.-J. Pesqué. *Algèbre Linéaire Creuse Parallèle pour les Méthodes Directes : Application à la Parallélisation d'un Code de Mécanique des Structures*. Journées sur l'Algèbre Linéaire Creuse et ses Applications Industrielles, IRISA, Rennes, mars 1999.