N° d'ordre : 2432

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Pascal Hénon**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Distribution des données et régulation statique des calculs et des communications pour la résolution de grands systèmes linéaires creux par méthode directe.

Soutenue le : 27 Novembre 2001

Après avis des rapporteurs :

Patrick Amestoy .	Maître de Conférence HDR
Pierre Manneback	Professeur

Devant la commission d'examen composée de :

Patrick Amestoy .	Maître de Conférence HDR	Rapporteur
Olivier Coulaud ..	Directeur de Recherche	Examinateur
Pierre Manneback	Professeur	Rapporteur
Yves Robert	Professeur	Président
Jean Roman	Professeur	Directeur de thèse

– 2001 –

REMERCIEMENTS

Je tiens vivement à remercier ici :

Patrick Amestoy, maître de conférences à l'IRIT-Toulouse, pour avoir rapporté sur ce travail. Sa lecture très approfondie de ce document et ses nombreuses remarques m'ont permis de compléter utilement ce document. Je tiens aussi à le remercier pour les encouragements et les conseils qu'il m'a donnés à chacune de nos rencontres tous au long de ma thèse ;

Pierre Manneback, professeur à l'Université Polytechnique de Mons en Belgique, pour l'honneur qu'il m'a fait en acceptant de rapporter sur ce travail ;

Yves Robert, professeur à l'Ecole Normale Supérieur de Lyon, pour l'honneur qu'il m'a fait en acceptant de présider ce jury ;

Olivier Coulaud, directeur de recherche à l'INRIA, pour avoir accepter d'être membre de ce jury ainsi que l'intérêt qu'il a porté à mes travaux ;

Je tiens à remercier tout particulièrement Jean Roman, professeur à l'ENSEIRB-Bordeaux, pour avoir encadré et dirigé ce travail avec autant de sérieux et d'implication. Je lui suis particulièrement reconnaissant pour le temps qu'il a pris sur ses week-ends et soirées afin de contribuer à ces travaux et les orienter par sa grande expérience du domaine. Il faut dire qu'il a tout d'abord du se mettre au Sedanais du Nord, dialecte peu connu en Aquitaine, pour quelques relectures d'articles...

Je tiens aussi à remercier tous les membres de l'équipe alienor et plus particulièrement : David Goudin, François Pellegrini, Guillaume Latu. Je les remercie pour savoir si bien allier travail et humour. Je remercie particulièrement Pierre Ramet avec lequel ça a été un plaisir de travailler, de voyager et de prendre plein de cappu... merci pour toutes les fois où nous nous sommes retrouvés le soir ou le week-end pour "boucler" un article ou arriver à faire passer ces fichus cas tests... Nous avons au cours de ses "séances" (bien rigoler finalement et) mis au point certaines théories telles que le morbugs et autres méthodes de Génie Logicielle assez particulières...

Un grand merci aussi à toutes les personnes du CEA/CESTAS avec lesquelles j'ai eu l'occasion de travailler.

Je tiens aussi à remercier les personnes qui ne sont pas des collaborateurs directs mais dont le soutien m'a été très précieux :

Merci à mes camarades de bureau qui en même temps que moi ont affronté les difficultés et les doutes que l'on rencontre au cours d'une thèse. D'abord les anciens : Valère, Davy et Guillaume qui sont avec moi depuis le début de cette thèse. Nos quatre caractères différents ont souvent formé les points cardinaux orientant des discussions diverses et (très) variées. Il y aussi ceux qui nous ont rejoint avec leur bonne humeur et leur sympathie naturelle afin d'animer encore plus ces discussions : Irek et Nico. Merci Nico pour les blagues et plus particulièrement pour l'avant-première de “Michel l'ingénieur” (bien avant les radios).

Au delà des personnes du LaBRI il y a toutes les personnes de ma famille dont l'influence et le soutien ont toujours été réconfortant et essentiel. Merci plus particulièrement à ma Tante Jeanine à laquelle je dois beaucoup pour ma motivation de faire de la recherche et dont le soutien a été déterminant dans mes études et à mon frère Dominique dont la confiance m'a souvent redonné confiance.

Un merci tout spécial à mon pote Adib avec lequel j'ai passé des soirées inoubliables et dont les conseils m'ont été plus que précieux. Donner du bonheur et du rêve aux gens est une qualité rare que possède Adib.

Enfin ces remerciements ne pourraient être complets sans un grand merci à ma Karine qui a été auprès de moi pendant ces trois années de thèse pour m'épauler et me rendre la vie plus belle.

Distribution des données et régulation statique des calculs et des communications pour la résolution de grands systèmes linéaires creux par méthode directe.

Résumé : La résolution par méthode directe de grands systèmes linéaires creux symétriques définis positifs est une étape cruciale très coûteuse en temps et en mémoire pour beaucoup d'applications scientifiques et industrielles. Dans cette thèse, nous considérons le problème du partitionnement en blocs et de l'ordonnancement des calculs pour un algorithme de factorisation parallèle sans pivotage dynamique. Nous nous intéressons plus particulièrement à l'extensibilité du solveur parallèle et au compromis entre le coût mémoire et l'efficacité en temps. Nous validons cette étude par des expérimentations parallèles sur un nombre important de problèmes industriels irréguliers de grande taille.

Mots clés : Calcul Haute Performance, Factorisation de Cholesky, solveur direct parallèle, matrice creuse, régulation statique.

Discipline : Informatique

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

Data distribution and static scheduling of computations and communications for direct solution of sparse linear systems.

Abstract : Solving large sparse symmetric positive definite systems of linear equations is a crucial and time-consuming step, arising in many scientific and engineering applications. In this work, we consider the block partitioning and scheduling problem for sparse parallel factorization without pivoting. We focus on the scalability of the parallel solver, and on the compromise between memory overhead and efficiency. We validate this study with parallel experiments on a large collection of irregular industrial problems.

Keywords : High performance computation, Cholesky factorization, parallel direct solver, sparse matrix, static partitioning and mapping.

Discipline : Computer Science

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

Table des matières

Liste des tableaux

Table des figures

Première partie

Introduction et cadre général de l'étude

Chapitre 1

Introduction à la résolution parallèle des grands systèmes linéaires creux

1.1 Cadre et objectifs de ce travail

La résolution de systèmes linéaires creux est une brique de base essentielle pour traiter numériquement de nombreux problèmes de calcul scientifique. Ces systèmes, qui apparaissent en particulier dans le cadre de la discrétisation d'équations aux dérivées partielles par éléments ou volumes finis, sont de très grande taille pour les applications irrégulières 3D en vraie grandeur. Les coûts opératoires et en mémoire induits sont tels que le parallélisme est alors une technique incontournable pour résoudre ces très grands systèmes (plusieurs millions d'inconnues).

La nature irrégulière de l'enchaînement des tâches de calcul induite par le caractère creux (termes nuls de la matrice associée au système) font de la résolution en parallèle des grands systèmes linéaires creux un sujet de recherche algorithmique très actif. Un des points cruciaux pour l'efficacité de la résolution parallèle de tels systèmes est conjointement la distribution des données et la régulation des calculs associés. En effet, de part l'irrégularité du problème traité, le choix de la distribution des données conditionne l'équilibrage de charge dans le temps, l'extensibilité et donc les performances globales des solveurs parallèles pour les systèmes linéaires creux.

Une première contribution de ce travail concerne une méthode de régulation statique des calculs et des communications pour la résolution de systèmes linéaires creux par méthode directe. Ces méthodes jouissent d'une très bonne réputation en ce qui concerne la robustesse numérique et sont incontournables pour des problèmes 3D mal conditionnés ; on se rapportera à [40] pour une application de nos travaux à un problème de mécanique des structures avec plastification et grands déplacements (travaux en collaboration avec le CEA/CESTA dans le cadre du contrat No.7V155-5AC).

Nous nous sommes donc intéressés non seulement aux problèmes liés à la distribution des données, mais aussi à ceux touchant l'ordonnancement des calculs et des communications comme prétraitement pour des factorisations parallèles directes sur des machines au comportement prédictible. Bien que nous ayons appliqué nos travaux essentiellement aux résolutions utilisant la factorisation de Cholesky ou de Crout, l'adaptation à d'autres méthodes de résolution est tout à fait envisageable car elle est tout à fait générale.

Les méthodes directes ont toutefois, par rapport aux méthodes itératives [76], l'inconvénient de nécessiter un plus grand espace mémoire (nombre de termes non nuls de la matrice factorisée, données intermédiaires pour la parallélisation) ; limiter l'encombrement global pour traiter les cas les plus grands possibles est actuellement un vrai challenge. Une deuxième contribution de ce travail est l'intégration dans notre algorithme de régulation du mécanisme d'agrégation partielle des contributions calculées pour notre solveur parallèle supernodal (cette technique assurera un bonne "extensibilité" mémoire de notre solveur parallèle), ainsi que l'intégration dans ce même algorithme de régulation des opérations de manipulation des données pour l'utilisation de notre solveur parallèle dans un cadre "out-of-core" (ce deuxième point devrait permettre de traiter avec une efficacité raisonnable de très grands systèmes creux).

Nous allons présenter maintenant ce que sont les méthodes de résolution directes les plus courantes pour fixer notre cadre d'étude algorithmique.

Considérons la résolution du système linéaire creux $A.x = b$, où A est la matrice creuse, x est le vecteur solution recherché et b le vecteur second membre du système. Les méthodes directes consistent à calculer la solution du système initial par la résolution successive de systèmes triangulaires ou diagonaux déduits de la matrice initiale par factorisation préalable. Il existe ainsi plusieurs factorisations possibles dont les plus courantes sont :

- la factorisation "LU" : A régulière est décomposée sous la forme $A = L.U$ avec L matrice triangulaire inférieure à diagonale unité et U matrice triangulaire supérieure ;
- la factorisation de Cholesky : A symétrique définie positive est décomposée sous la forme $A = L.L^t$ avec L matrice triangulaire inférieure ;
- la factorisation de Crout : A symétrique définie positive est décomposée sous la forme $A = L.D.L^t$ avec L matrice triangulaire inférieure à diagonale unité et D matrice diagonale. Il est à noter qu'il est possible d'utiliser la factorisation

$L.D.L^t$ pour des matrices symétriques non définies positives ; dans ce cas la matrice D est diagonale par bloc.

La factorisation "LU" non symétrique est généralement à pivotage dynamique ce qui signifie qu'au cours de la factorisation on a parfois besoin de faire pivoter certaines lignes de la matrice afin de préserver une bonne stabilité numérique. Ces pivotages étant dépendants des termes calculés lors de la factorisation de la matrice, on ne peut les anticiper statiquement (i.e. avant la factorisation effective). On peut toutefois estimer la structure englobante de tous les pivotages numériques et utiliser cette structure statique pour estimer la structure creuse [23] ; on peut aussi utiliser des techniques de prétraitement de la matrice initiale A et, en utilisant un pivotage statique, obtenir une factorisation approchée de la matrice initiale [28, 29, 61].

Par contre les factorisations de Cholesky et de Crout sans pivotage permettent une connaissance a priori de tous les coûts de calculs qui interviennent durant la factorisation. C'est particulièrement cette factorisation que nous avons étudiée et à laquelle nous avons appliqué nos méthodes de distribution des données et de régulation statique des calculs et des communications.

Une fois la factorisation terminée, la solution est alors obtenue par la résolution des systèmes triangulaires ($L.y = b$, $D.z = y$ et $L^t.x = z$ par exemple pour la factorisation de Cholesky-Crout).

Comme nous l'avons indiqué précédemment, le travail présenté dans ce document porte sur la distribution des données et l'ordonnancement des calculs et des communications pour la factorisation Cholesky-Crout. Nous nous intéressons donc dans cette étude aux systèmes linéaires dont la matrice est symétrique, définie positive, et pour lesquels la résolution peut se faire sans pivotage numérique. Une approche statique pour le calcul au préalable d'une "bonne" distribution/régulation est alors possible et souhaitable.

Les stratégies de résolution peuvent différer selon la structure physique de la machine parallèle utilisée. Nous nous placerons pour notre étude dans le cadre des machines MIMD à mémoire distribuée traditionnelles, ou hybrides c'est à dire constituées d'un réseau haut débit connectant des multiprocesseurs à mémoire partagée (nœuds SMP) ; nous serons alors, pour notre algorithme de distribution/régulation statique, amené à modéliser les différents niveaux de hiérarchie pour les échanges de données (réseau, mémoire partagée). Ce dernier type de machine est le plus en vogue actuellement et est celui qui permet la résolution des plus grands systèmes de par la quantité de mémoire mise à disposition. Par exemple, le SP3 du CINES dispose de 28 nœuds SMP NH2, ayant chacun 16 Powers 3 (1,5 GFlops crête) et 16 Go de mémoire partagée, soit au total pour 448 processeurs, 672 GFlops crête et

448 Go de mémoire. Un autre exemple est la machine TERA du CEA/DAM composée de nœuds SMP quadri-processeurs EV 68 cadencés à 833Mhz et 1,66 Gflops de puissance crête, avec 4 Go de mémoire par nœud et un réseau Quadrics.

D'un point de vue calculatoire, il est maintenant bien connu qu'il faut profiter au mieux des effets pipelines et des effets caches qui sont très importants sur les processeurs actuels ; c'est pourquoi l'utilisation des bibliothèques BLAS (Basic Linear Algebra Subroutines) [54, 55] s'est généralisée, et c'est pourquoi les solveurs directs performants utilisent une algorithmique par blocs ; à nouveau, nous serons amenés à modéliser finement les opérateurs BLAS utilisés pour notre solveur parallèle pour notre algorithmique de distribution/régulation statique.

Le cadre parallèle retenu dans un premier temps pour ce travail a donc été celui des machines MIMD à mémoire distribuée ou en réseau de nœuds SMP, avec un modèle de programmation de type SPMD, utilisant une bibliothèque de communication de type MPI [2, 4] adaptée aux machines SMP, et la bibliothèque de calcul BLAS.

Dans ce contexte, l'essentiel des contributions de cette thèse est :
- l'obtention d'une distribution des données et d'un ordonnancement des calculs très performants pour une factorisation parallèle extensible ;
- la possibilité de contrôler de manière statique l'utilisation de la mémoire pour cette factorisation parallèle ;
- l'obtention de distributions et d'ordonnancements spécifiques aux machines à nœuds SMP ;
- la validation de ces études sur un grand nombre de problèmes irréguliers industriels de grande taille et sur un grand nombre de processeurs.

Ces différentes contributions constituent l'intégralité de l'algorithmique de la partie distribution/ordonnancement du prétraitement qui "pilote" le solveur PaStiX [46–50] développé au LaBRI initialement et conjointement avec Pierre Ramet dans le cadre d'une partie de sa thèse [67]. La validation pratique de l'ensemble des travaux sur ce solveur PaStiX a été faite d'autre part dans le cadre de la thèse de David Goudin [40] pour de grandes applications complexes de mécanique des structures discrétisées par éléments finis [38, 39].

Le reste de ce chapitre rappelle les étapes nécessaires à la résolution parallèle d'un système linéaire creux par la factorisation de Cholesky-Crout.

Le chapitre 2 décrit les différents problèmes que nous tenterons de résoudre et dresse un état de l'art des différentes méthodes existantes de distribution et d'ordonnancement des calculs pour les méthodes directes. Nous verrons dans les paragraphes 2.3 et 2.4 les problèmes rencontrés dans les méthodes de distribution classiques. Nous expliquerons notamment comment une méthode qui combine le repartitionnement des blocs de la matrice, la distribution et un ordonnancement statique des tâches de calcul et de communication permet de régler ces problèmes.

Le chapitre 3 donne l'idée générale de notre méthode de régulation statique des calculs et des communications basée sur un comportement prédictible du solveur et de la plate-forme d'exécution. Pour cela, nous décrirons les modélisations que nous utilisons pour guider nos stratégies de régulation statique de la factorisation parallèle. Nous décrirons aussi dans ce chapitre l'algorithme parallèle du solveur apte à suivre les schémas d'exécution pré-calculés.

Le chapitre 4 décrit les algorithmes utilisés pour le repartitionnement, la distribution et l'ordonnancement des calculs et des communications. Il présente des résultats de complexité et montre comment les algorithmes sont naturellement adaptés aux architectures basées sur des nœuds SMP ou des grappes de processeurs à comportement prédictible. Ce chapitre donnera aussi des résultats et des analyses sur les performances obtenues par nos techniques de régulation statique pour de très grands systèmes creux. Nous présenterons ainsi des résultats sur les temps de factorisation parallèle pour des machines IBM SP2 et SP3, et les comparerons à ceux obtenus par le solveur de référence PSPASES [57].

Dans le chapitre 5 nous abordons les aspects mémoires dans la factorisation parallèle et les moyens que nous avons de les contrôler. Nous traiterons d'une part les problèmes liés au surcoût mémoire induit par l'algorithmique parallèle de la factorisation ; nous montrerons comment nous pouvons modifier nos schémas d'ordonnancement des communications afin de diminuer ce surcoût. Nous montrerons et développerons d'autre part les premiers résultats concernant une adaptation des techniques "Out-of-Core" à notre factorisation parallèle. Le mécanisme "Out of Core" vise à ne travailler en mémoire vive qu'avec les données nécessaires aux calculs courants, ce qui permet d'appréhender des tailles de problèmes beaucoup plus importantes. Nous verrons dans cette étude préliminaire que ces techniques s'adaptent particulièrement bien à notre mécanisme de régulation statique des calculs et des communications puisque celui-ci permet d'anticiper les données à accéder. Il faudra, comme dans les études précédentes, tirer partie au mieux du fonctionnement des entrées-sorties asynchrones de la plate-forme d'exécution.

Ce document se terminera par une conclusion et les perspectives de ce travail.

1.2 Présentation des différentes étapes de la résolution parallèle d'un système linéaire creux par méthode directe

Dans cette section, nous présentons de manière générique les différentes notions et les différentes étapes pour la conception d'un solveur parallèle direct pour matrices creuses. On pourra se réferrer à [26, 27, 34, 36, 62, 68] et aux références incluses pour une vue d'ensemble du domaine.

1.2.1 Renumérotation des inconnues

Un des points essentiels propres aux méthodes directes sur les matrices creuses est le problème du *remplissage* [36]. 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* [36] associé à l'élimination de Gauss. Ce remplissage étant directement lié à l'ordre d'élimination des inconnues, il s'agit donc de trouver une renumérotation de ces inconnues, c'est à dire une *renumérotation* [5, 36, 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 lors de la factorisation.

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

L'arbre d'élimination joue un rôle fondamental pour la parallélisation des solveurs creux directs car il indique les dépendances dans les calculs : il ne peut y avoir de dépendances (et donc une séquentialité dans les calculs) entre deux éliminations d'inconnues que si les sommets associés sont sur une même branche de l'arbre d'élimination. Dans le paragraphe 2.1 nous reviendrons plus en détail sur l'exploitation des informations que l'arbre d'élimination peut fournir pour la distribution et l'ordonnancement des calculs.

Dans un cadre parallèle, le but est donc de trouver une renumérotation des sommets de G qui *minimise le remplissage* et qui *maximise l'indépendance dans les calculs de la factorisation*, c'est à dire conduisant à des arbres d'élimination *larges et de faible hauteur*. Comme on le verra dans le chapitre 3 paragraphe 3.1, 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. Nous expliciterons alors le type de renumérotation que nous utilisons et son impact sur la distribution en blocs de la matrice.

1.2.2 Factorisation symbolique par blocs

Une autre étape fondamentale est celle dite de *factorisation symbolique par blocs* [21] qui va permettre de calculer algorithmiquement la structure creuse par blocs de la matrice L à partir de celle de A. Le but est d'optimiser la *factorisation numérique* [11–14] 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 *blocs-colonnes* de la matrice, le *graphe quotient* G^*/P à partir de G et de cette partition P. L'intérêt de l'algorithme de factorisation symbolique par blocs repose sur le fait que pour les partitions P considérées dans ce travail, les opérations de passage au quotient et d'élimination commutent, c'est à dire que l'on a l'égalité $(G/P)^* = G^*/P$. La complexité [21] en temps et en espace de cet algorithme croît alors comme le nombre total de blocs extra-diagonaux dans la structure ainsi construite pour L (voir figure 1.1) 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 blocs* (voir figure 1.1) et sa structure va décrire maintenant les dépendances dans les calculs par bloc du solveur. Pour pouvoir définir cet arbre d'élimination par blocs, nous introduisons les deux définitions suivantes.

Définition 1.2.1
Pour chaque bloc-colonne k, $1 \leq k \leq N$,

- *$BCol(k)$ est l'ensemble des indices $i > k$ de blocs-colonnes en face des blocs extra-diagonaux du bloc-colonne k dans la matrice L ;*
- *$BRow(k)$ est l'ensemble des indices $j < k$ de blocs-colonnes tels que $k \in BCol(j)$.*

Ainsi, dans l'exemple de la figure 1.1, on a $N = 7$ et pour $k = 3$, $BCol(3) = \{7\}$ et $BRow(3) = \{1, 2\}$.

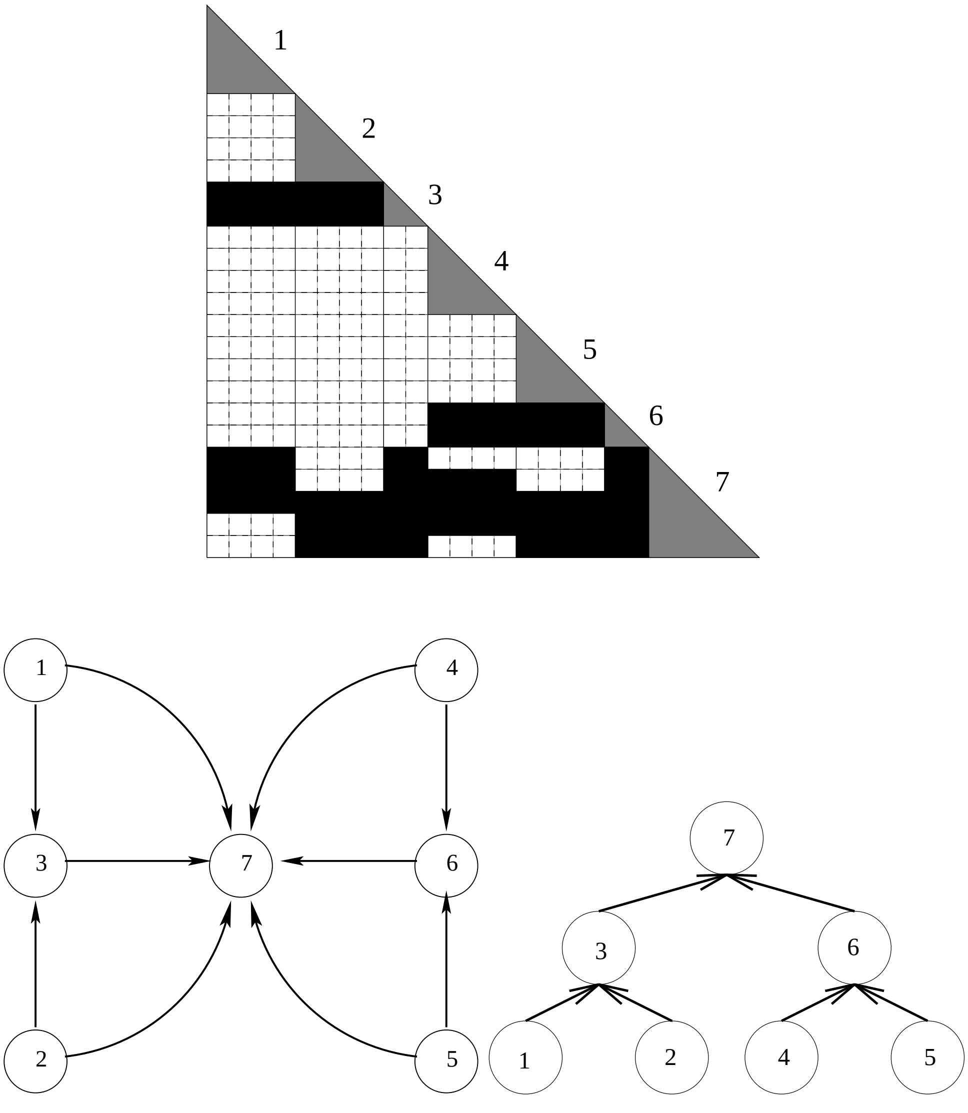


FIG. 1.1 – Un exemple de matrice factorisée L structurée par blocs-colonnes, d'arbre d'élimination par blocs et de graphe des échanges entre blocs-colonnes. Chacun des 7 blocs-colonnes est constitué d'un bloc diagonal dense (en gris) et de plusieurs blocs extra-diagonaux denses (en noir).

L'arbre d'élimination par blocs, obtenu par réduction transitive des échanges entre blocs-colonnes à partir du graphe d'élimination par blocs, a alors pour définition :

Définition 1.2.2
L'arbre d'élimination par blocs a N sommets, et le père du sommet k, $1 \leq k \leq N$, est l'élément minimum de $BCol(k)$.

Cet arbre d'élimination par blocs (voir figure 1.1) est un arbre recouvrant du graphe des échanges entre blocs-colonnes (voir figure 1.1) durant l'algorithme de factorisation par blocs (voir figure 1.2) ; ce graphe n'est autre que le graphe quotient G^*/P que l'on calcule donc symboliquement sous la forme $(G/P)^*$.

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

1.2.3 Repartitionnement et distribution des données

On doit donc réaliser un prétraitement des données qui va 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. Ce prétraitement va donc calculer de manière statique une *régulation équilibrée* pour le solveur (critère d'équilibrage de charge des processeurs et prise en compte des contraintes de précédence) ; on utilisera pour cela la structure de l'arbre d'élimination par blocs introduite précédemment. En effet, si k est un numéro de bloc-colonne, $BRow(k)$ sera l'ensemble des indices de blocs-colonnes qui modifient le bloc-colonne k par l'ajout de contributions (ligne 8 de la figure 1.2), et $BCol(k)$ sera l'ensemble des indices de blocs-colonnes qui sont modifiés par le bloc-colonne k. Ainsi, l'arbre d'élimination par blocs indique les contraintes de précédence au sens suivant : le calcul local relatif à un bloc-colonne k (ligne 2 à 5 de la figure 1.2) et les modifications apportées aux blocs-colonnes de $BCol(k)$ ne peuvent se faire qu'après que les modifications apportées sur k par les blocs-colonnes de $BRow(k)$ aient été prises en compte ; les éléments de $BCol(k)$ sont dans l'ascendance de k dans l'arbre d'élimination et ceux de $BRow(k)$ sont dans sa descendance. Le parallélisme induit par le creux est d'autant plus important que l'arbre d'élimination par blocs est large.

Le but sera donc 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. C'est donc le parallélisme induit par le creux ;
2. le parallélisme à grain moyen, dû à la possibilité de raffiner la partition des inconnues en découpant un nœud de l'arbre d'élimination et en le distribuant sur plusieurs processeurs. C'est le parallélisme induit par une factorisation sur des blocs denses. On parlera aussi de parallélisme au niveau des nœuds ;
3. le parallélisme à grain fin, ou micro-parallélisme. Celui-ci est obtenu en tirant partie du parallélisme interne d'un processeur lors du calcul sur les blocs (utilisation optimale de l'effet pipeline des processeurs superscalaires). Ce dernier niveau de parallélisme est fondamental pour obtenir de bonnes performances. Il repose, modulo *une bonne taille de blocs*, essentiellement sur l'utilisation des primitives BLAS3 [42, 43, 63, 71, 72, 74].

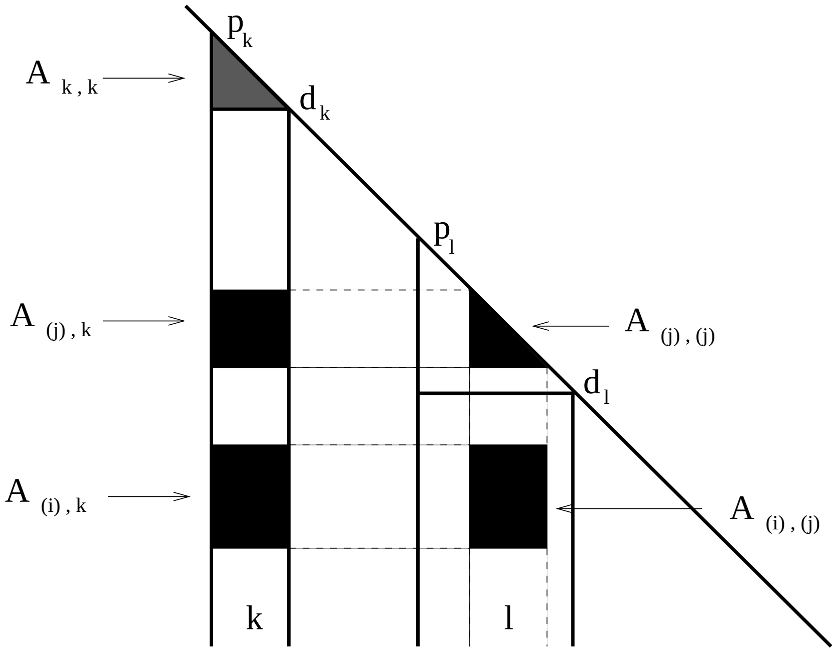
A noter que l'on essaiera aussi d'optimiser la localité des échanges et des accès aux données pour éviter des contentions dans le réseau du calculateur et des concurrences d'accès au niveau des mémoires partagées.

A partir de la structure de données par blocs calculée par la factorisation symbolique [21], 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-colonnes 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 pourront alors être distribués par exemple de manière bloc cyclique pour utiliser le parallélisme contenu dans les calculs denses. On construit ainsi de manière statique une bonne régulation des calculs [32, 35, 69, 73].

Cette technique peut être utilisée selon un schéma de distribution unidimensionnelle (1D) des blocs-colonnes, on exploitera alors le parallélisme lié à l'indépendance des calculs existant entre blocs-colonnes dans la boucle la plus externe de l'algorithme (voir algorithme par blocs générique pour une factorisation $A = L.D.L^t$ figure 1.2), ou selon un schéma bidimensionnel (2D) des blocs-colonnes dans lequel on exploite en plus le parallélisme lié à l'indépendance des calculs existant entre les blocs élémentaires de cette distribution (Phases 2 & 3 de l'algorithme de la figure 1.2). Une distribution 2D permet une meilleure extensibilité du solveur parallèle [72] mais générera un surcoût de communication par rapport à une distribution 1D. Nous montrerons dans les paragraphes 4.1 et 4.2 qu'une distribution mixte est un bon compromis.

1.2.4 Factorisation parallèle

Il existe 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* ou *fan-out* [14–17, 45, 63] et l'approche *multifrontale* [6–8, 22, 30, 31, 42, 43]. Ces approches diffèrent essentiellement sur la manière de répercuter les modifications (ou contributions) entre les blocs de la matrice (ligne 8 de l'algorithme de la figure 1.2). Dans l'approche multifrontale, on associe à chaque bloc-colonne une matrice frontale qui contiendra toutes les contributions du bloc-colonne générées lors de la phase 3. Cette matrice frontale est ensuite “passée” au père du bloc-colonne dans l'arbre d'élimination par bloc. La matrice frontale d'un nœud de l'arbre d'élimination sera ensuite constituée de l'agrégation des matrices frontales de ses fils et de ses propres contributions. Dans ces méthodes le parallélisme de second niveau s'exprime dans le découpage et la parallélisation du calcul des blocs denses représentés par les matrices frontales. Une description sur la parallélisation unidimensionnelle ou bidimensionnelle est donnée dans [7, 8].



Notations : Pour chaque bloc-colonne k, $1 \leq k \leq N$,
- p_k et d_k sont respectivement les première et dernière colonnes du bloc-colonne ;
- $A_{k,k}$ est le bloc diagonal dense ;
- b_k est le nombre de blocs extra-diagonaux dans le bloc-colonne k ;
- $A_{(j),k}$ est le $j^{\text{ième}}$ bloc extra-diagonal dense avec $1 \leq j \leq b_k$, (j) étant un multi-indice décrivant un intervalle de lignes.

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

1.Pour $k := 1$ à N faire
2. Factoriser $A_{k,k}$ en $L_{k,k}D_kL^t_{k,k}$; (Phase 1)
3. Pour $j := 1$ à b_k faire (Phase 2)
4. Résoudre $L_{k,k}A^t_{(j),k} = A^t_{(j),k}$;
5. Résoudre $D_kL^t_{(j),k} = A^t_{(j),k}$;
6. Pour $j := 1$ à b_k faire (Phase 3)
7. Pour $i := j$ à b_k faire
8. $A_{(i),(j)} = A_{(i),(j)} - L_{(i),k}D_kL^t_{(j),k}$;

FIG. 1.2 – Factorisation $A = L.D.L^t$ par blocs séquentielle.

En ce qui concerne les méthodes supernodales qui constituent les méthodes retenues dans ce travail, l'approche "left-looking" consiste à répercuter sur le bloc-colonne courant les modifications des blocs-colonnes "de gauche" concernés, puis à calculer la valeur finale du bloc-colonne courant ; de manière symmétrique, l'approche "right-looking" consiste à calculer définitivement le bloc-colonne courant et à répercuter ses contributions sur les blocs-colonnes "de droite" concernés. Par exemple l'algorithme séquentiel de la figure 1.2 est exprimé sous la forme "right-looking" puisque la phase 3 répercute définitivement les contributions du bloc-colonne k qui vient d'être calculé sur les blocs-colonnes de droite.

Dans un cadre parallèle, il y a deux manières de transmettre les données lors de la factorisation Cholesky-Crout. La première appelée "fan-in" consiste à compresser les volumes de données échangées ; toutes les contributions venant de gauche pour un bloc ou un bloc-colonne et calculées sur un même processeur sont agrégées dans un bloc recouvrant de ces contributions. La seconde appelée "fan-out" consiste à envoyer au plus tôt les contributions telles qu'elles ont été calculées ; cette approche n'a de réel intérêt que dans une approche "right-looking". La figure 1.3 illustre ces deux principes dans le cas unidimensionnelle. La figure de gauche illustre l'approche "fan-out" ; sur le processeur P_1, deux blocs-colonnes ont des contributions pour le même bloc-colonne sur le processeur P_2 ; dans l'approche "fan-out" ces contributions sont envoyées séparemment dès qu'elles sont prêtes, le processeur P_2 les ajoute alors successivement lorsqu'il les reçoit. La figure de droite illustre l'approche "fan-in" ; dans cette approche, les contributions des deux blocs-colonnes sur le processeur P_1, lorsqu'elles ont été calculées, sont agrégées dans un bloc-colonne recouvrant qui est ensuite envoyé au processeur P_2.

Les méthodes "fan-out/right-looking" sont généralement utilisées pour la factorisation de matrices pleines où on a intérêt à envoyer au plus tôt les contributions car les contraintes de précédence entre les calculs sont fortes. Dans le cadre de la factorisation parallèle d'une matrice creuse, l'approche "fan-in" est plus performante car elle permet de limiter considérablement la masse de communication. Par contre, le surcoût mémoire généré par le stockage des agrégats de contribution peut se montrer rédibitoire comme le montrera le paragraphe 5.1. Le paragraphe 3.3 présentera notre algorithme parallèle qui se situe plutôt dans le domaine "fan-in/right-looking" compte tenue de la technique d'agrégation des contributions retenue. Nous verrons dans le paragraphe 5.1 qu'afin de contrôler les surcoûts mémoire dûs au stockage de ces agrégats, notre approche permet de décider d'agréger partiellement au lieu de totalement afin de réduire ces surcoûts mémoire. Ainsi dans ce cas, notre régulation statique permettra de déplacer l'algorithmique de la factorisation parallèle vers le domaine "fan-out/right-looking" en acceptant de perdre un peu en performance temps pour gagner en extensibilité mémoire.

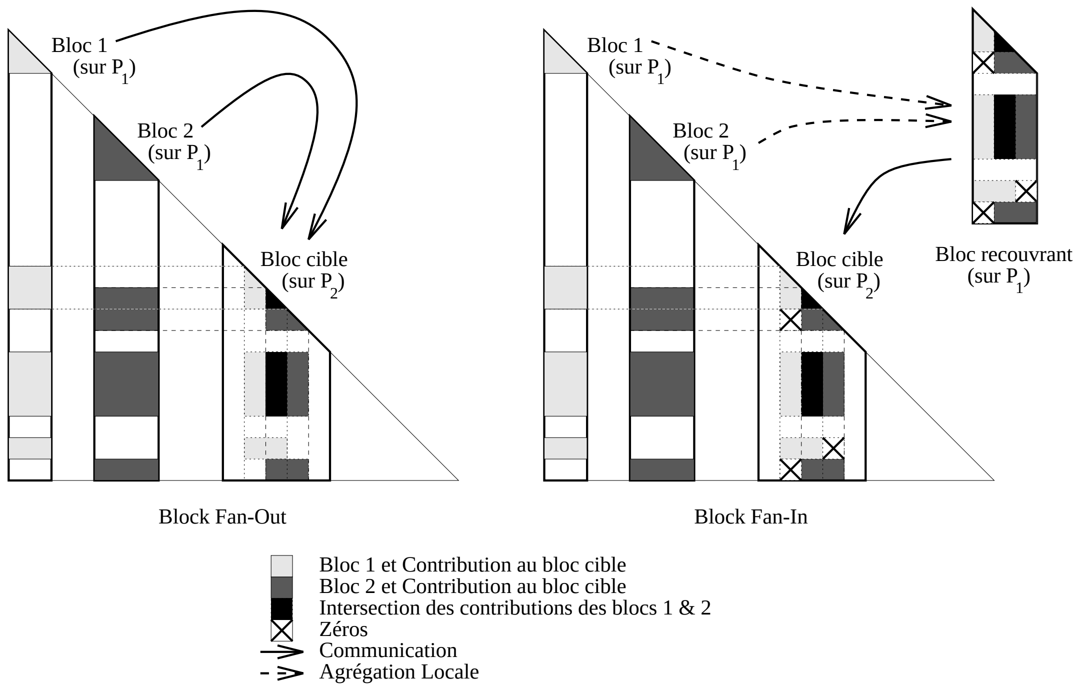


FIG. 1.3 – *Block fan-in*, *block fan-out* et calcul du bloc-colonne recouvrant.

1.2.5 Résolution des systèmes triangulaires

Une fois la matrice L calculée, on obtient la solution finale en résolvant successivement les trois systèmes triangulaires $L.y = b$ (phase dite de "descente"), $D.z = y$, puis $L^t.x = z$ (phase dite de "remontée"). Dans le cas d'une résolution mono-second membre (i.e. b est un vecteur), cette phase est peu coûteuse en nombre d'opérations par rapport à la factorisation, et c'est pourquoi on ne redistribue généralement pas la matrice [44, 57]. La question peut se poser dans le cas d'une résolution multi-seconds membres (i.e. b est une matrice rectangulaire $n \times m$ où n est la dimension du système et m le nombre de second membres) si m est assez grand. Dans ce dernier cas, des redistributions spécifiques peuvent être nécessaires.

Chapitre 2

Etat de l'Art sur les problèmes de distribution et d'ordonnancement

2.1 Utilisation de l'arbre d'élimination par blocs

L'arbre d'élimination, que nous avons introduit aux paragraphes 1.2.1 et 1.2.2, joue un rôle fondamental dans la distribution des données dans les méthodes de factorisation. De part l'algorithmique par blocs des solveurs, c'est l'*arbre d'élimination par blocs* qui est donc utilisé. La renumérotation joue ici un rôle crucial puisque c'est elle qui induit en particulier la forme de l'arbre d'élimination. En effet, plus l'arbre est large, plus il exhibe de parallélisme induit par le creux. Généralement les techniques actuelles de renumérotations, que nous décrirons plus en détails dans le paragraphe 3.1, ainsi que les techniques de distribution des données associées, permettent d'affecter au moins un sous-arbre complet par processeur. Les factorisations des blocs associés à ces sous-arbres peuvent alors être traitées de manière totalement indépendante sur chaque processeur.

Cependant il a été observé (par exemple dans [6]) qu'au moins 75% des coûts de calcul sont souvent effectués dans les trois ou quatre premiers niveaux de l'arbre d'élimination par blocs. Il est donc indispensable d'obtenir un autre parallélisme à l'intérieur des nœuds du haut de l'arbre d'élimination. Ce parallélisme au niveau des nœuds est généralement obtenu en divisant les tâches de calcul, et donc les blocs, et en les pipelinant comme pour la factorisation d'une matrice pleine. La figure 2.1 illustre les types de parallélisme que l'on retrouve suivant le niveau dans l'arbre d'élimination.

Nous allons, dans le paragraphe 2.2, décrire les solutions qui ont été adoptées par les bibliothèques de résolution parallèle de systèmes linéaires creux actuelles. Nous nous intéresserons plus précisemment aux trois aspects essentiels suivants dans la

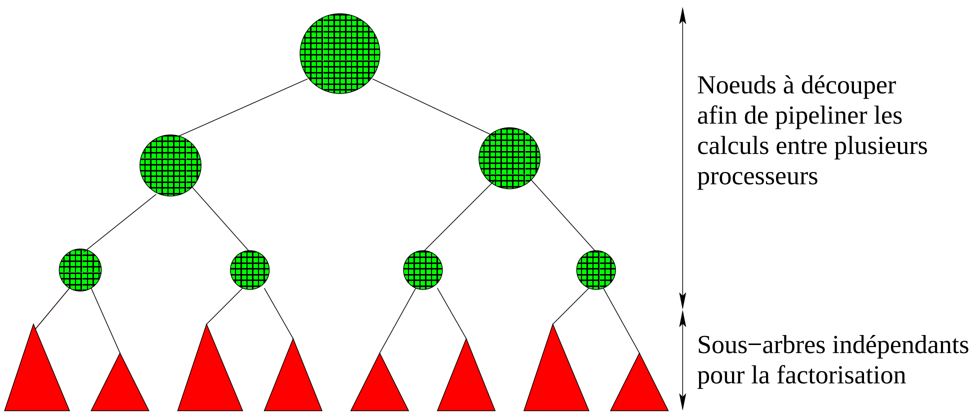


FIG. 2.1 – Types de parallélisme exhibés dans l'arbre d'élimination.

parallélisation de la factorisation Cholesky-Crout :

1. la distribution des blocs de la matrice sur les différents processeurs ;
2. la parallélisation des tâches de calcul associées aux nœuds du haut de l'arbre d'élimination ;
3. l'ordonnancement de l'ensemble des tâches de calcul.

Nous avons choisi pour cet état de l'art les solveurs numériques les plus performants et les plus représentatifs des techniques actuelles ; les aspects distribution et parallélisation étant assez proches pour les factorisations $L.U$ et $L.D.L^t$, nous avons aussi inclus dans cette partie des exemples de distributions et d'ordonnancements des calculs issus de solveurs utilisant la factorisation $L.U$. Nous avons aussi jugé intérressant de présenter quelques stratégies de parallélisation utilisées dans des solveurs qui visent des machines à mémoires partagées.

2.2 Etat de l'Art des méthodes actuelles

L'idée la plus simple pour la factorisation en parallèle d'une matrice creuse est d'adapter les techniques utilisées dans la factorisation parallèle des matrices pleines. Pour les matrices pleines, de nombreuses études ont montré [25, 81] qu'une distribution cyclique par blocs était très efficace et extensible. Cette distribution repose sur une partition isométrique de la matrice pleine en blocs de tailles suffisantes pour avoir une bonne efficacité des primitives BLAS, et sur une distribution cyclique de cette partition sur une grille virtuelle de processeurs comme illustré dans la figure 2.2(a). L'intérêt de cette méthode pour le cas plein est qu'elle permet une très bonne utilisation de l'effet pipeline tout en produisant un équilibrage de charge idéal. Le schéma de communication est aussi très adapté aux architectures de machines reposant sur des grilles ou des hypercubes de processeurs puisque qu'un processeur n'a besoin de communiquer qu'avec les processeurs sur la même ligne et sur la même colonne que lui dans la grille de processeurs. Dans [61] les auteurs adaptent cette méthode aux systèmes creux pour une factorisation $L.U$ avec pivotage statique(il est à noter que dans le cas étudié par ces auteurs, la réduction transitive du graphe d'élimination ne conduit pas à un arbre mais à un DAG (*Direct Acyclic Graph*)) ; la figure 2.2(b) illustre une telle distribution sur une matrice creuse. Il est à noter que dans le cadre de l'étude les principaux intérêts conservés par une telle distribution dans le cas du creux sont la simplicité de sa mise en œuvre, une bonne robustesse au niveau des ressources systèmes notamment concernant la gestion des communications, et le fait d'être bien adaptée à des machines possédant des réseau de communication basé sur une architecture de grille ou d'hypercube comme celui du T3E par exemple. Néanmoins, il est évident qu'une telle distribution de part l'irrégularité dans la taille et dans la répartition des blocs ne peut pas toujours permettre un bon équilibrage de charge. Des études ont toutefois été réalisées pour essayer d'équilibrer la charge en redimensionnant les lignes et les colonnes de la partition bloc de la matrice [75]. Toutefois ces méthodes de distribution sont peut utilisées actuellement ; elles ont été délaissées au profit de celles se basant sur l'arbre d'élimination par blocs de la matrice.

Une bonne régulation de la charge et de l'ordonnancement des calculs doit donc s'appuyer sur l'arbre d'élimination par bloc. Comme nous l'avons indiqué précédemment, la matrice est constituée de blocs-colonnes (super-nœuds) dont certains seront traités par une algorithmique séquentielle (sous-arbres indépendants), et d'autres qui devront être repartitionnés afin de fournir assez de calculs élémentaires pour être pipelinés en parallèle. Il y a essentiellement deux approches qui permettent de déterminer dans quelle catégorie de super-nœuds chaque nœud de l'arbre d'élimination par blocs devra être classé, et quels processeurs participeront à sa factorisation. La première consiste à répartir la charge en partant du bas de l'arbre d'élimination et en remontant ; on détermine ainsi en premier une répartition des sous-arbres du bas entre les processeurs qui équilibre la charge, et l'on essaye en remontant dans l'arbre de distribuer et de repartitionner les nœuds afin de préserver cet équilibrage. La seconde consiste à partir du haut de l'arbre d'élimination et à répartir de manière récursive l'ensemble des processeurs proportionnellement au travail de chaque branche de l'arbre d'élimination jusqu'à obtenir des sous-arbres indépendants pour chaque processeur ; tous les nœuds qui ne sont pas dans ces sous-arbres sont repartitionnés pour être factorisés en parallèle sur le groupe de processeurs lui correspondant dans le parcours descendant. Nous allons maintenant

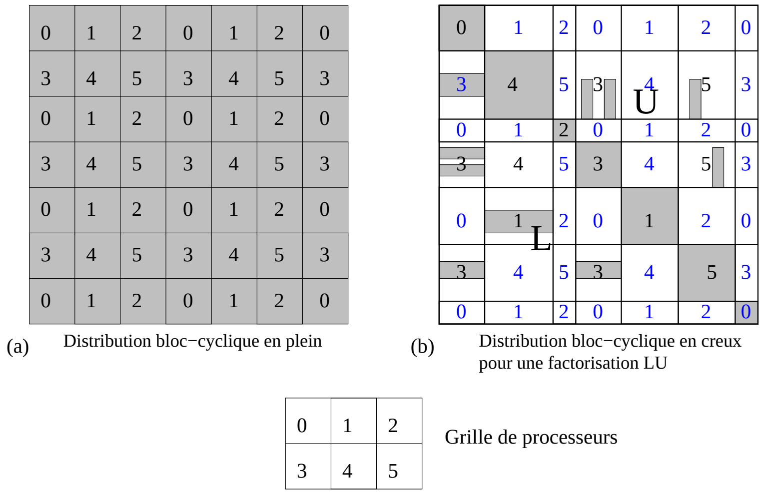


FIG. 2.2 – Distribution bloc-cyclique sur une grille de 6 processeurs pour une matrice pleine et une matrice creuse.

détailler un peu plus chacune de ces méthodes.

Les approches basées sur un parcours ascendant de l'arbre d'élimination commence par calculer une *strate initiale* qui permet un bon équilibrage de charge en distribuant les sous-arbres enracinés en ses nœuds sur les processeurs. La figure 2.3 donne l'algorithme de la construction d'une strate initiale [35] qui assure une distribution associée à cette strate dont le déséquilibre de charge n'exède pas une certaine tolérance.

Ensuite, On construit une partition en strates en disant que la strate suivante se compose uniquement des pères des nœuds composant la strate courante (voir figure 2.4). La remontée de l'arbre se fait alors itérativement en distribuant successivement les strates jusqu'à la racine de l'arbre d'élimination. Lorsque les nœuds les plus gros en coût de calcul déséquilibrent trop la répartition de charge lors de cette distribution ascendante, ils sont repartitionnés afin de répartir au mieux leurs charges de calcul entre plusieurs processeurs.

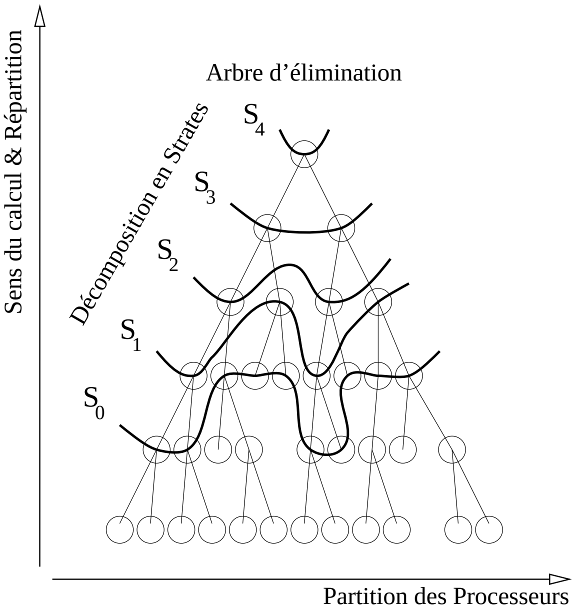


FIG. 2.4 – Décomposition par strates de l'arbre d'élimination par bloc.

Construction et distribution de la strate initiale L_0 :

1. $S_0 \leftarrow$ Racine de l'arbre d'élimination par blocs ;
2. **Faire**
3. Trouver le nœud $q \in S_0$ le plus coûteux à factoriser ;
4. $S_0 \leftarrow (S_0 \setminus q) \cup \{les\ enfants\ de\ q\}$ (Voir ci-dessous) ;
5. Distribution des sous-arbres de S_0 sur les processeurs ;
6. Estimation du déséquilibre de charge ;
7. **Tant que** le déséquilibre est supérieur à la tolérance.

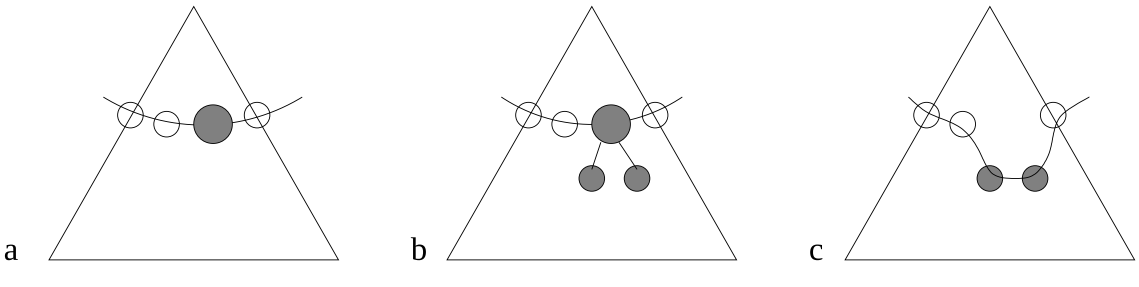


FIG. 2.3 – Construction et distribution de la strate initiale

Le deuxième type de méthode équilibre la charge en repartitionnant/distribuant

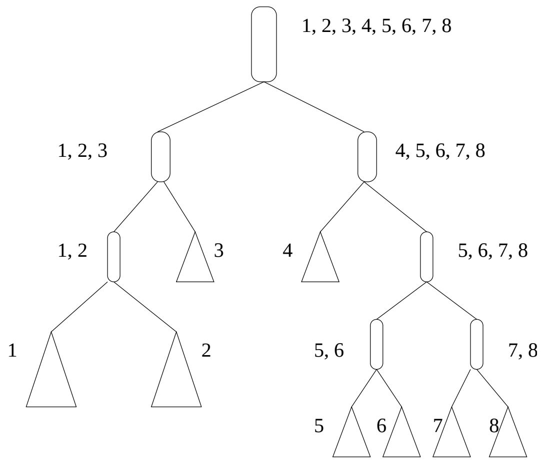


FIG. 2.5 – Distribution récursive par *proportional mapping.*

les nœuds en descendant l'arbre d'élimination. On commence par distribuer la racine de l'arbre entre tous les processeurs. Puis on partitionne l'ensemble des processeurs en groupes que l'on affecte sur l'ensemble des sous-arbres de la racine, chaque groupe ayant un cardinal proportionnel au poids du sous-arbre. La répartition des processeurs sur les sous-arbres est alors répétée récursivement jusqu'à ce qu'il n'y ait plus qu'un seul processeur par sous-arbre (voir figure 2.5). Ces méthodes sont communément appelées *proportionnal mapping* [66].

La manière de répartir les processeurs entre les sous-arbres a donné quelques variantes à la méthode. Ainsi, le *subtree-to-subcube* [56] cherche à optimiser la localité des communications qui auront lieu pour des architectures parallèles basées sur des grilles ou des hypercubes ; on assure dans cette méthode que chaque sous-arbre sera distribué sur un sous-cube de l'hypercube. On peut généraliser le *subtree-to-subcube* à n'importe quel type de réseau en partitionnant le graphe des processeurs proportionellement aux sous-arbres ; on parle alors de *subtree-to-submesh.*

Les méthodes de *proportionnal mapping* permettent ainsi d'obtenir une très bonne localité des communications. En effet, lorsqu'un super-nœud de l'arbre d'élimination est repartitionné et distribué sur un groupe de processeurs, toutes les contributions qu'il doit recevoir sont calculées par ce même groupe de processeurs. Par contre, le problème principal de ces méthodes est celui des erreurs d'arrondis dues au calcul en arithmétique entière lors de la répartition des processeurs assignés à un super-nœud proportionnellement au coût de calcul de ses sous-arbres. L'autre problème qui vient parfois contrarier l'équilibrage de charge provient du fait qu'un super-nœud n'est pas toujours assez large (taille critique due aux BLAS3) pour pouvoir être repartitionné et distribué sur l'ensemble de processeurs qui lui a été assigné.

Jusqu'à présent nous avons vu comment on peut choisir les super-nœuds qui seront factorisés localement sur un processeur, et ceux qui devront être raffinés en tâches élémentaires pour être factorisés en parallèle sur plusieurs processeurs. Nous allons voir maintenant les possibilités qu'exploitent les solveurs actuels pour paralléliser le calcul des super-nœuds devant être raffinés.

Afin de pouvoir pipeliner les calculs liés à la factorisation des plus gros super-nœuds ceux-ci doivent être redécoupés en blocs plus fins. Pour profiter des effets pipelines générés par l'utilisation des primitives BLAS, il convient de découper ces blocs avec une taille minimun qui dépend des caractéristiques de la machine cible. Par exemple, dans le cas de la SP2 du CINES (Thin nodes P2SC 120 Mhz) en réel double précision, la largeur de bloc minimum est de l'ordre de 60. La manière de paralléliser le calcul des super-nœuds doit donc tenir compte du nombre de processeurs disponibles et de la taille du super-nœud pour être "optimale". Ainsi à taille égale de super-nœud, pour un faible nombre de processeurs on a plutôt intérêt à avoir des distributions qui favorisent l'efficacité des primitives BLAS par des calculs plus compacts sur les blocs ; par exemple, on pourra utiliser une distribution monodimemsionelle cyclique par blocs-colonnes (1D). Par contre pour un nombre plus élevé de processeurs, et afin d'être globalement plus extensible, on peut utiliser des distributions qui favorisent le pipeline des calculs en parallèle ; par exemple, on pourra utiliser une distribution bidimensionelle cyclique par blocs (2D).

Dans MUMPS [7, 8] qui est un solveur multifrontal, on distingue ainsi plusieurs types de parallélisme ; le type 1 sera celui des sous-arbres indépendants, le type 2 celui des nœuds intermédiaires et le type 3 celui de la racine de l'arbre d'élimination par blocs. Les nœuds de type 2 et leurs matrices frontales sont distribués par blocs-lignes comme indiqué sur la figure 2.6. Dans cette configuration, le calcul d'un bloc-colonne d'un super-nœud de type 2 se fait comme suit : le processeur maître commence par factoriser le bloc diagonal qui a reçu toutes ses contributions, puis il l'envoie à tous les processeurs esclaves (communication (1) sur la figure 2.6). Chaque esclave peut alors utiliser cette information pour calculer sa partie du bloc-colonne et mettre à jour la matrice frontale des contributions. Tous les esclaves, sauf le dernier, peuvent alors envoyer le bloc qu'ils viennent de calculer à tous les esclaves en "dessous d'eux" (communications (2) et (3) sur la figure 2.6).

Le parallélisme de type 3 n'intervient que pour la racine de l'arbre d'élimination par blocs. C'est généralement le plus gros super-nœud et celui auquel tous les processeurs doivent participer. Dans MUMPS, on utilise pour la factorisation de ce super-nœud une distribution bloc-cyclique car elle permet a priori une très bonne extensibilité et donc une bonne utilisation de tous les processeurs disponibles.

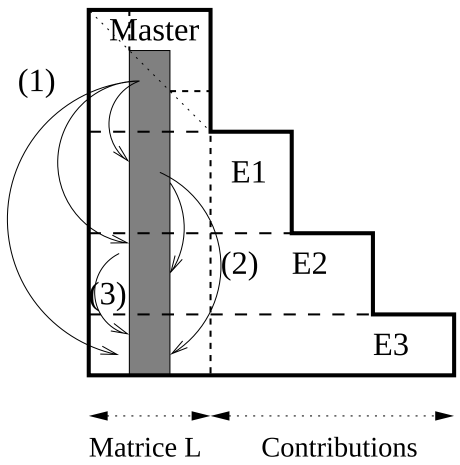


FIG. 2.6 – Calcul des nœuds de type 2 d'une matrice symétrique dans MUMPS.

Dans PSPASES [42,43,57] qui est aussi un solveur multifrontal, la distribution repose sur un parcours descendant de l'arbre d'élimination de type *subtree-to-subcube*, et on utilise une distribution bloc-cyclique de la matrice frontale basée sur une grille de processeur assignée aux super-nœuds (voir figure 2.7). Cette distribution est très efficace pour l'approche multifrontale en termes de communication, car à chaque opération d'ajout de contribution les processeurs communiquent par paire.

Regardons maintenant les aspects concernant l'ordonnancement et la régulation.

Dans le cas de PSPASES, l'ordonnancement des calculs et des communications suit la même logique que celle utilisée pour des distributions cycliques par blocs dans le cas des matrices pleines.

Par contre, un certain nombre de solveurs utilisent un ordonnancement de nature dynamique des calculs et des communications. Dans MUMPS par exemple, on utilise un "pool" distribué de tâches de calculs liées aux blocs locaux prêtes à être exécutées ; lorsqu'un processeur a fini une tâche de calcul, il prend la suivante dans son pool de tâches ayant reçues toutes ses contributions. La figure 2.8 décrit l'algorithme d'ordonnancement utilisé. On voit que, dans cet algorithme, la priorité est donnée à la reception des messages (lignes 6, 7), car on a ainsi une plus grande chance de rendre disponible des tâches de calcul supplémentaires. Il est en effet important de consommer assez vite les messages en attente afin de débloquer les ressources systèmes consommées par MPI (buffers d'émission et de réception).

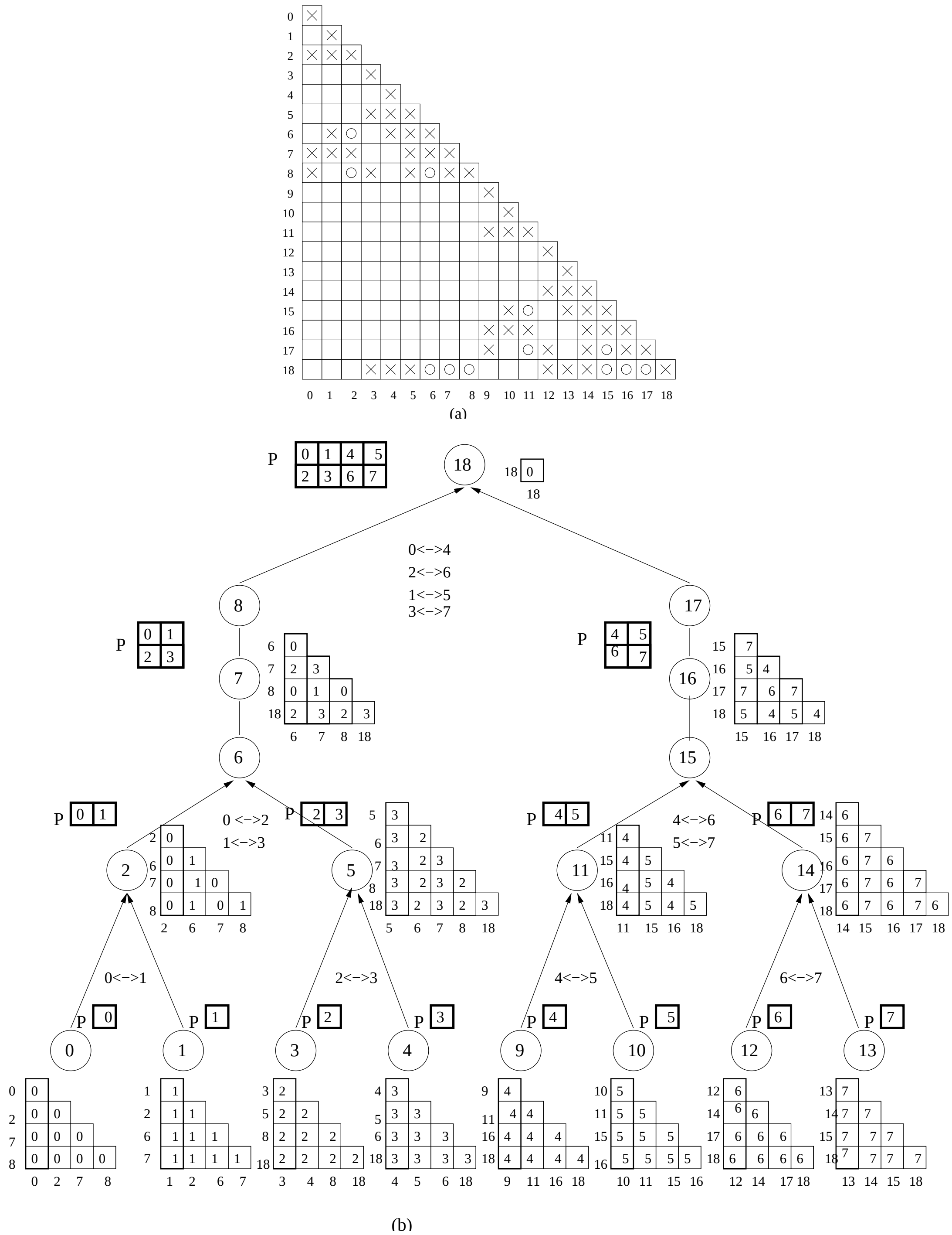


Fig. 2.7 – (a) Un exemple de matrice symétrique : les "X" représentent les termes non nuls de A dans la partie triangulaire supérieure, et les termes non nuls de L sont représentés dans la partie triangulaire inférieure avec des "O" en ce qui concerne les termes de remplissage. (b) Distribution des matrices frontales utilisées au cours de la factorisation multifrontale en parallèle. Pour chaque super-nœud, les colonnes de L sont en gras et les autres représentent la matrice de contribution.

```
Tant que (tous les nœuds ne sont pas traités)
    Si le pool des tâches prêtes est vide alors
        Attente blocante d'un message ;
        Traitement du message ;
    Sinon
        Si il y a un message en attente alors
            Traitement du message ;
        Sinon
            Extraire une tâche de calcul du pool et la traiter.
```

FIG. 2.8 – Algorithme d'ordonnancement des calculs et des communications dans MUMPS.

Dans les solveurs qui visent des machines à mémoire partagée, l'ordonnancement dynamique des tâches de calcul a une part très importante. En effet, sur ces machines, on peut raisonnablement considérer que l'accès aux données a le même coût quel que soit le processeur, et il n'y a donc pas de raison de distribuer la matrice sur les processeurs ; c'est un ordonnancement dynamique couplé à un fonctionnement maître -esclave de type glouton des calculs qui régit la factorisation parallèle. C'est aussi le cas dans les codes multifrontaux MA41 et MA49 [6,10] de factorisation $L.U$ et $Q.R$ pour architecture parallèle, disponibles dans la bibliothèque HARWELL [51].

Dans [24], les auteurs présentent une version de SuperLU qui vise les machines à mémoire partagée à faible nombre de processeurs ; dans cette approche supernodale, les tâches de calcul sont associées à une distribution par bloc-colonne. Lorsqu'un processeur a fini une tâche de calcul, l'ordonnanceur lui assigne une autre tâche en respectant les trois règles de priorités suivantes :

1. l'ordonnanceur teste si le père dans l'arbre d'élimination du dernier bloc-colonne traité est prêt à être factorisé, c'est-à-dire si tous les autres enfants de ce bloc-colonne ont été traités. Si c'est le cas, la factorisation de ce bloc-colonne devient la tâche courante sur le processeur ;

2. si le père n'est pas prêt, alors l'ordonnanceur assigne une autre tâche de calcul prête à être traitée ;

3. enfin, s'il n'existe pas d'autre bloc-colonne prêt alors l'ordonnanceur assigne le père au processeur, le calcul de celui-ci étant pipeliné avec la reception des contributions de ses nœuds enfants traités par d'autres processeurs.

Dans d'autres solveurs comme PARDISO [77] ou SPOOLES [18], dédiés eux aussi aux machines à mémoire partagée, le choix d'un ordonnancement dynamique décidant à "la volée" sur quel processeur et dans quel ordre seront traitées les tâches de calculs a aussi été fait. Notons que ce choix est ici tout à fait légitime puisque une distribution cherchant à équilibrer la charge de calcul ne peut pas faire mieux qu'un équilibrage dynamique qui, grâce à la mémoire partagée, ne nécessite pas de redistribution de données. Dans PARDISO, qui choisit une approche supernodale, les auteurs ont même développé une souplesse supplémentaire qui autorise les calculs des contributions à être faits en tant que tâches de calculs indépendantes. Ainsi les contributions peuvent être traitées, soit comme dans une approche "right-looking" par le processeur qui "émet", ou soit comme dans une approche "left-looking" par le processeur qui "reçoit". Notons toutefois que tous ces solveurs pour machines à mémoire partagée sont surtout adaptés à un nombre modéré, de processeurs car au delà les phénomènes de synchronisation pour l'accès concurrent aux données deviennent souvent pénalisants. Pour obtenir plus de scalabilité avec ces techniques en mémoire partagée, on peut utiliser une technique de "proportionnal mapping" pour déterminer les sous-arbres indépendants dont les calculs seront assignés à des processeurs *a priori*; les calculs du haut de l'arbre seront eux toujours traités avec un équilibrage dynamique de la charge.

2.3 Problèmes liés à la distribution des blocs et à l'ordonnancement des calculs et des communications

Comme nous l'avons vu dans le paragraphe précédent, et pour les solveurs dédiés aux machines à mémoire distribuée, l'équilibrage de charge repose en particulier sur l'estimation du coût de calcul dans les sous-arbres de l'arbre d'élimination par blocs pour les méthodes de type *proportionnal mapping*, et sur le coût de calcul des nœuds pour les méthodes montantes. Dans beaucoup de distributions, on se fie alors au nombre d'opérations; or celui-ci ne traduit pas les effets pipelines dûs à l'utilisation des bibliothèques BLAS; il convient donc de modéliser le comportement en temps des primitives de ces bibliothèques afin d'obtenir les estimations les plus proches possibles de la réalité.

Un premier problème est le comportement du coût réel des BLAS qui n'est pas linéaire par rapport à la taille des blocs; lorsque l'on repartitionnera un super-nœud k on modifiera son coût global et celui de tous les blocs qui appartiennent à $Brow(k)$ (voir paragraphe 1.2) et qui seront repartitionnés eux aussi. Cette imprécision est augmentée du fait des surcoûts supplémentaires qui peuvent être induits par l'algorithme de parallélisation qui met en œuvre des stratégies spécifiques comme celle de l'agrégation des contributions dans le cas d'une méthode supernodale.

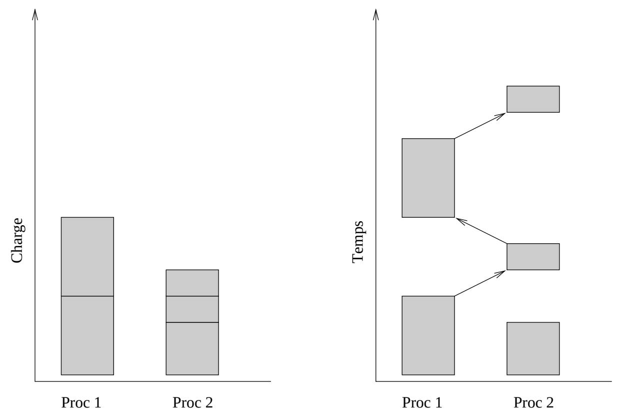


FIG. 2.9 – Contre-exemple à la fausse équivalence (charge de calcul $\equiv$ avancement dans le temps)

Le second des problèmes et non le moindre réside dans le fait que dans la plupart des méthodes actuelles, la distribution des blocs ne se fait qu'en fonction d'un équilibrage de la charge de calcul. Ceci ne tient pas compte des temps de communication, des attentes de messages pour pouvoir réaliser le calcul, etc ... Or comme le montre trivialement la figure 2.9, ce n'est pas parce qu'un processeur possède la plus grosse charge de calcul qu'il sera le plus en retard à la fin du calcul. Les contraintes de précédence jouent en effet un rôle fondamental.

Le troisième type de problème réside dans le fait que pour être efficace, le solveur devra aussi gérer efficacement les alternances entre traitement des tâches de calculs prêtes (ayant reçues toutes leurs contributions), émission des messages de contributions et ajoût des messages reçus, et ce bien sûr en fonction d'une distribution calculée dans le prétraitement. Par exemple dans [9], les auteurs montrent que la bonne gestion de ces évènements dans MUMPS permet d'obtenir une scalabilité satisfaisante jusqu'à plus de 200 processeurs. Il faut aussi décider dans quel ordre traiter un ensemble de blocs prêts car il est difficile avec les informations locales à un processeur de déterminer le plus critique. De même, faut-il interrompre le traitement des blocs prêts pour ajouter des messages reçus afin de pouvoir traîter un bloc plus critique ? La figure 2.10 montre par exemple comment l'ordre de traitement peut générer du retard.

Les solveurs essayent souvent de réguler ces problèmes lors de l'exécution par des critères plus ou moins pertinents, mais il n'en reste pas moins que ce sont des choix indéterministes et qui n'assurent pas en général d'être proche de l'optimalité.

En ce qui concerne les solveurs dédiés aux machines à mémoire partagée, une

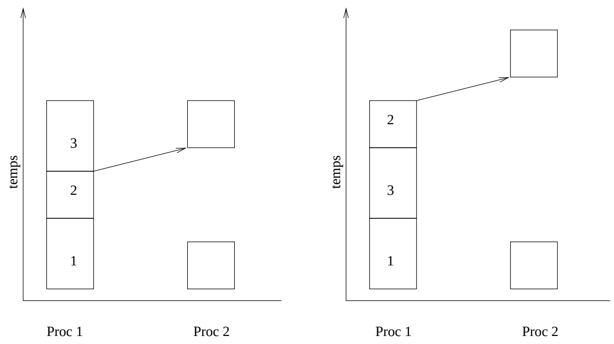


FIG. 2.10 – Importance de l'ordre de traitement des blocs prêts.

distribution et un ordonnancement dynamique des tâches de calcul est le meilleur choix car il peut être réalisé plus simplement et plus efficacement du fait qu'il prendra le mieux en compte l'état de disponibilité réel des processeurs au moment où une tâche de calcul est prête à être traitée.

2.4 Positionnement de notre étude

Dans la partie suivante de ce document, nous allons présenter une méthode de distribution et de régulation des calculs et des communications pilotant une factorisation parallèle supernodale de Cholesky-Crout. Notre approche tente de régler la plupart des problèmes qui interviennent dans la parallélisation de cette factorisation pour des machines à mémoire distribuée ou constituées de nœuds SMP.

Comme dans le cas des solveurs pour machine à mémoire partagée, notre méthode s'appuie sur une distribution prenant en compte au mieux possible la disponibilité des processeurs lorsqu'une tâche de calcul est prête. Pour cela nous nous appuyons sur une simulation du comportement parallèle de notre solveur. Cette simulation repose sur une modélisation des bibliothèques BLAS et du réseau de communication de la machine ciblée. Ainsi, nous verrons comment nous allons réaliser cette distribution par simulation et comment nous avons conçu un mécanisme d'ordonnancement des calculs et des communications qui assure que la factorisation parallèle se déroule selon les prédictions réalisées.

Les chapitres 3 et 4 exposeront donc les idées et l'algorithmique de notre méthode. Dans le chapitre 5, nous verrons un autre aspect important de notre méthode qui est la possibilité de contrôler les surcoûts mémoires importants induits par le parallélisme et par l'agrégation des contributions selon la méthode de parallélisation que nous avons choisie. Nous présenterons aussi dans ce chapitre 5 une étude préliminaire concernant une version "Out-of-Core" de notre solveur ; notre approche, de part la connaissance a priori de tous les accès aux données durant la factorisation parallèle, est particulièrement bien adaptée à ce type de méthode.

Deuxième partie

Une méthode de distribution et de régulation statique des calculs et des communications appliquée à la factorisation Cholesky-Crout en parallèle

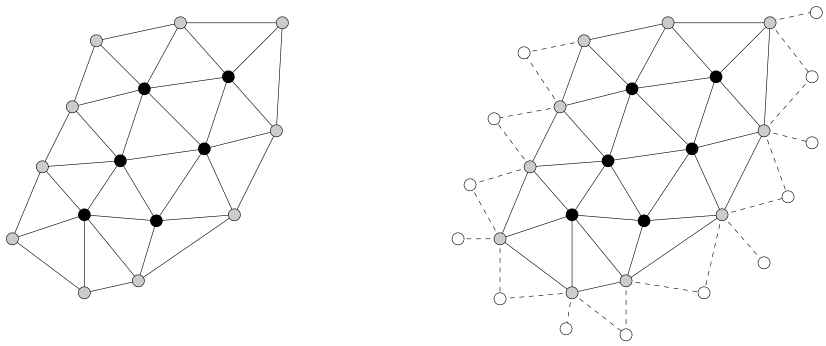
Chapitre 3

Description générale de la méthode

3.1 Renumérotation utilisée et factorisation symbolique par blocs

Dans le paragraphe 2.1, nous avons insisté sur le rôle de l'arbre d'élimination par blocs pour le repartitionnement et la distribution. Classiquement, le premier objectif de la renumérotation est de limiter le remplissage dans la matrice L et ainsi de diminuer le nombre d'opérations à effectuer lors de la factorisation. De part l'utilisation d'une algorithmique par blocs adaptée à l'utilisation des primitives BLAS dans les solveurs, une autre exigence est que les super-nœuds du haut de l'arbre d'élimination soit "assez gros" pour que leurs calculs soient pipelinés entre plusieurs processeurs.

Les deux méthodes les plus utilisées actuellement sont celles du degré minimun et des dissections emboîtées. L'idée de l'algorithme du degré minimun [5, 80] est de choisir à chaque étape le pivot correspondant au sommet de plus petit degré du graphe associé à la matrice. L'algorithme de dissections emboîtées [36] consiste à déterminer un ensemble de sommets S qui sépare le graphe en deux parties A et B non voisines, renuméroter S avec les plus grands numéros disponibles, et recommencer récursivement sur les sous-graphes A et B jusqu'à ce que leurs tailles deviennent inférieure à une valeur limite. Cette numérotation garantit qu'aucun terme non nul ne peut apparaître lors de la factorisation entre des inconnues de A et des inconnues de B. Dans les distributions logicielles qui font références actuellement [58,59,64,65], seuls les premiers niveaux de récursion de la dissection emboîtée sont effectués, les sous-graphes restants étant alors numérotés par degré minimun en dessous d'une certaine taille.



a. Sans halo. **b.** Avec halo.

FIG. 3.1 – Connaissance du degré des sommets dans un sous-graphe avec ou sans *halo*. Les sommets en noir sont les sommets internes au sous-graphe, les sommets en gris sont les sommets de la frontière du sous-graphe, et les sommets en blanc sont ceux du *halo*, utilisé par l'algorithme du *Halo Approximate Minimum Degree*.

Pour la renumérotation, nous utilisons le logiciel SCOTCH [64, 65] de partitionnement de graphes irréguliers développé au LaBRI en collaboration avec Patrick Amestoy. Ce logiciel conduit à un bon compromis entre la conservation du creux et l'équilibrage de l'arbre d'élimination. SCOTCH utilise la méthode de dissections emboîtées jusqu'à un certain seuil, qui correspond à un nombre de sommets de l'ordre de quelques centaines dans la plupart des cas, puis les sous-graphes sont renumérotés en utilisant l'algorithme du *Halo Approximate Minimun Degree* [65] à partir de ce seuil. Comme le montre la figure 3.1, l'amélioration apporté par *HAMD* tient du fait que la renumérotation locale dans les sous-graphes se fait en tenant compte du degré et de la numérotation des sommets de leurs cocycles (halo). La factorisation symbolique par blocs, déjà introduite au paragraphe 1.2, est réalisée à partir de l'algorithme présenté dans [21]. Ce dernier a, pour la renumérotation utilisée ici, une complexité linéaire en temps et en espace ; il produit donc une structure de données en blocs-colonnes pour la matrice factorisée, dont on déduit l'arbre d'élimination par blocs.

Les numérotations obtenues par ces techniques hybrides sont très performantes car elles produisent des arbres d'élimination par blocs qui ont de bonnes propritétés :

– le haut de l'arbre d'élimination correspond aux inconnues avec les plus grands numéros ; les premiers séparateurs issus de la dissection emboîté correspondent ainsi aux super-nœuds du haut de l'arbre d'élimination par blocs. Ces super-nœuds sont donc d'autant plus gros qu'ils sont hauts dans l'arbre d'élimination par blocs ; ceci convient tout à fait aux besoins de la parallélisation qui font intervenir d'autant plus de processeurs dans le calcul d'un super-nœud qu'il est haut dans l'arbre d'élimination par blocs ;

- à chaque étape la dissection emboîtée sépare le graphe en deux sous-graphes ayant un nombre de sommets à peu près égal ; ceci permet d'avoir un arbre d'élimination par blocs binaire et bien équilibré dans le haut. Généralement il y a au moins autant de sous-arbres au niveau où s'arrête la dissection emboîtée que de processeurs ;
- la numérotation locale (*HAMD*) utilisée dans les sous-graphes après la dissection emboîtée limite très souvent mieux le remplissage que la dissection emboîtée pour ces tailles de graphes. La forme de l'arbre d'élimination induite par cette renumérotation n'est pas très importante car elle intervient au niveau des sous-arbres qui seront traités indépendament et séquentiellement sur chaque processeur.

Les travaux menés actuellement sur ces techniques de renumérotation consistent d'une part en la validation d'un critère algorithmique adaptatif pour le basculement entre dissections emboîtées et degré minimun (faire intervenir par exemple la topologie des sous-graphes à séparer en plus de leurs tailles), et d'autre part en l'essai d'autres heuristiques locales pour la renumérotation des sous-graphes (remplacement du degré minimum par une heuristique de type *minimum fill* [70]).

3.2 Introduction à notre méthode de distribution et d'ordonnancement

Comme nous l'avons exposé dans le chapitre 2, les problèmes rencontrés pour une parallélisation efficace de la factorisation Cholesky-Crout peuvent être classés essentiellement en deux catégories ; d'une part, des problèmes dûs à la mesure du travail qui sert à effectuer l'équilibrage de charge, et d'autre part ceux liés à l'ordonnancement dynamique des différentes tâches de calcul et de communication lors de la factorisation numérique parallèle.

Lors de l'équilibrage de charge, l'estimation du travail à partir de la matrice symbolique par blocs est généralement assez grossière car elle est basée uniquement sur le nombre d'opérations à effectuer. Pour être efficace, les algorithmes des solveurs sont exprimés par blocs afin de permettre l'utilisation des primitives BLAS. Or la complexité en temps de ces primitives est loin d'être linéaire en temps (voir paragraphe 3.4). De plus cette estimation du travail ne prend pas en compte les autres phénomènes qui interviennent dans la factorisation parallèle tel que le coût supplémentaire généré par l'agrégation locale des contributions, et les temps d'attente dûs aux latences des communications.

Pour ce qui est des problèmes d'ordonnancement dynamique, ils sont de deux natures :

- l'ordonnancement des tâches de calcul est crucial (voir paragraphe 2.3) : un bon ordonnancement doit pouvoir tenir compte du fait qu'une tâche de calcul est plus critique qu'une autre si elle permet de débloquer une nouvelle tâche de calcul plus rapidement ;
- l'envoi et le traitement en réception des messages qui influencent l'ordre local des calculs : l'ordonnancement “optimal” est celui qui permet de recevoir les contributions aux blocs de la matrice locale dans l'ordre suivant lequel seront traitées les tâches de calcul associées.

Nous proposons de résoudre tous ces problèmes par une *régulation statique* qui s'appuie sur une simulation en temps de la factorisation. La régulation se fait par un ordonnancement des calculs et des communications calculé lors de la phase de distribution. Ainsi tous les problèmes qui sont d'habitude traités en dynamique lors de la factorisation numérique sont reportés dans la phase de prétraitement, l'intérêt étant que lors de la phase de distribution, on possède beaucoup plus d'informations que lorsque les données sont distribuées. Pour que notre approche soit fiable, nous devons avoir une connaissance assez précise des coûts de calcul et de communication ; nous utilisons pour cela une modélisation en temps des primitives BLAS qui interviennent lors de la factorisation numérique, ainsi qu'une modélisation en temps du passage d'un message de taille donnée sur la machine cible (voir paragraphe 3.4).

Le repartitionnement de la matrice se fait d'ordinaire en même temps que la distribution. Or le repartitionnement des plus gros blocs change le coût de calcul (voir le modèle des BLAS dans le paragraphe 3.4), et modifie les schémas de communication possibles (géométrie des blocs d'agrégation) ; nous avons donc séparé les phases de repartitionnement et de distribution. Ainsi dans notre approche ces deux phases ont les rôles suivants :

La phase de repartitionnement découpe les blocs-colonnes associés au plus gros super-nœuds, construit pour chaque bloc-colonne un ensemble de processeurs *candidats* pour sa distribution, et décide si le bloc-colonne repartitionné devra être distribué en 1D (par blocs-colonnes plus petits) ou en 2D (par blocs). La figure 3.2 illustre le travail qui est fait lors de cette phase de répartitionnement sur un exemple où la machine cible serait composée de 16 processeurs. Lors de cette phase les processeurs sont répartis récursivement en partant du haut de l'arbre et proportionnellement au travail dans les sous-arbres ; il est important de préciser que nous autorisons un processeur à être candidat dans *plusieurs sous-arbres* afin de ne pas être confronté aux problèmes d'arrondis communément rencontrés dans les méthodes de *proportionnal mapping* : c'est le cas dans notre exemple pour les processeurs 4 et 12. Il est important de comprendre que ceci est possible car nous ne fixons aucune distribution à ce stade du prétraitement ; c'est la phase de distribution et d'ordonnancement qui aura pour rôle de distribuer les calculs des super-nœuds sur leurs processeurs candidats. Ainsi, la phase de distribution pourra utiliser un même processeur pour le calcul de deux super-nœuds au même niveau de l'arbre d'élimination.

Lorsqu'un super-nœud possède plusieurs processeurs candidats, il faut le repartitionner pour pouvoir pipeliner son calcul. Comme nous l'avons vu dans le paragraphe 2.2, les calculs peuvent être d'un grain plus ou moins fin : nous utilisons soit des tâches de calcul correspondant à un repartitionnement en blocs-colonnes du super-nœud (distribution 1D), soit des tâches de calcul correspondant aux calculs élémentaires sur les blocs du super-nœud repartitionné (distribution 2D ; les types de tâche de calcul sont définis dans le paragraphe 3.3). Une distribution 1D privilégie les effets BLAS de par le calcul sur des plus gros blocs (l'organisation des données permet de traiter globalement de manière compactée l'ensemble des blocs extra-diagonaux d'un même bloc-colonne) alors qu'une distribution 2D est plus extensible à un nombre élevé de processeurs de par son grain de calcul plus fin et de par la possibilité de mieux recouvrir les communications par des calculs. Ainsi, et comme le montre la figure 3.2, nous retrouvons trois types de parallélisme dans l'arbre d'élimination.

Une fois que cette phase est terminé, le graphe des tâches de calculs élémentaires est construit.

La phase de distribution et d'ordonnancement a pour rôle de distribuer les tâches de calcul (et donc les blocs associés) et de les ordonnancer. Pour cela, elle s'appuie sur une simulation en temps de la factorisation qui permet, lorsqu'une tâche de calcul devient prête au sens de la simulation, de choisir *lequel de ses processeurs candidats permettra de la traiter au plus vite*. Nous utilisons un algorithme glouton qui commence par distribuer les feuilles de l'arbre d'élimination par blocs et qui n'ont qu'un seul processeur candidat. Puis toutes les tâches de calculs dont les contributions dépendent de celles précédemment distribuées, sont placées dans des listes de tâches prêtes pour leurs processeurs candidats. On continue alors la distribution en choisissant la tâche qui pourra être traitée la première par un de ses processeurs candidats. On réitère le procédé jusqu'à ce qu'il ne reste plus de tâche à distribuer. Nous verrons dans le paragraphe 4.3 une description détaillée de cet algorithme.

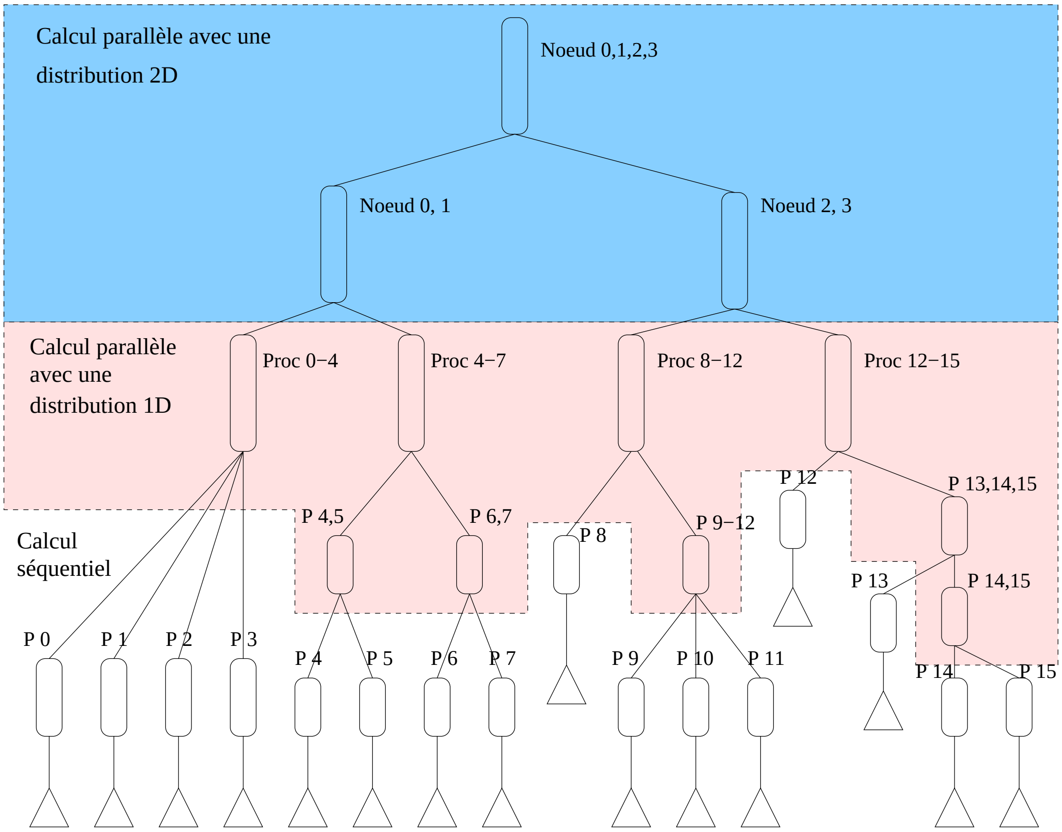


Fig. 3.2 – Repartitionnement et processeurs candidats.

L'ordonnancement obtenu pour les tâches de calcul induit alors *un ordonnancement compatible des communications*, nous reviendrons sur ce point dans les paragraphes 3.3 et 4.3.

La stratégie que nous proposons repose donc sur un fort couplage entre la phase de distribution et l'algorithme parallèle de factorisation : l'exécution parallèle doit suivre exactement l'ordonnancement calculé lors de la simulation, sinon l'équilibrage de charge n'a plus de sens. Dans le paragraphe 3.3, nous présentons notre algorithme parallèle qui permet de respecter ces prédictions.

3.3 Algorithme de factorisation parallèle

Notre algorithme parallèle est une version supernodale de la factorisation $L.L^t$ (Cholesky) ou $L.D.L^t$ (Crout) avec agrégation locale des contributions. De ce point de vue, on se rapproche de l'algorithme *Fan-In* [17] où les processeurs communiquent uniquement par des blocs agrégés de contributions. L'algorithme que nous proposons peut donc traiter à la fois des distributions par blocs-colonnes (1D) ou des distributions par blocs (2D). Si l'occupation mémoire devient un paramètre crucial bloquant pour le calcul alors, comme nous le verrons dans le paragraphe 5.1, des blocs agrégés de contributions peuvent être envoyés avant d'avoir été totalement agrégés pour libérer temporairement de la mémoire : on parlera alors d'agrégation partielle.

Ainsi, un bloc L_{ij}, $i \geq j$ dans le bloc-colonne j recevra nécessairement un bloc agrégé de l'ensemble des processeurs appartenant à $Procs(L_{ij})$ (la figure 3.3 illustre la contribution de L_{ik} pour le bloc L_{ij}) défini comme suit (voir définition 1.2.1 pour les ensembles $BCol$ et $BRow$) :

Définition 3.3.1
$Procs(L_{ij}) := \{map(i,k) \mid k \in BRow(j) \text{ et } i \in BCol(k))\}$
où $map(i,k)$ est la fonction de distribution qui associe chaque bloc L_{ik} à un processeur.

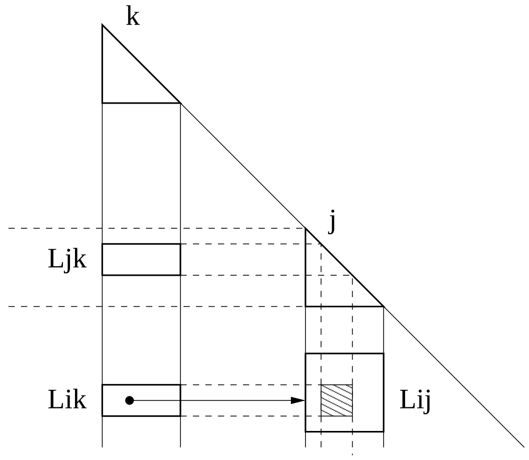


FIG. 3.3 – Contribution du bloc L_{ik} pour le bloc L_{ij}

Le code de la factorisation parallèle est donc découpé en tâches qui doivent être ordonnancées par la phase de distribution que nous présenterons dans la section 4.3. Ces tâches sont atomiques au sens des dépendances de calcul : une fois que le calcul d'une tâche a débuté, on est assuré que celui-ci ne sera interrompu par aucune attente de contribution. Les tâches que nous avons définies peuvent être liées au calcul d'un bloc-colonne (distribution 1D) ou alors liées à une opération élémentaire entre blocs (distribution 2D).

Avant de définir ces tâches nous introduisons quelques notations :
- P : nombre total de processeurs ;
- TNb_p : nombre de tâches affectées au processeur p, $1 \leq p \leq P$;
- $K_p[n]$: $n^{\text{ième}}$ tâche locale pour le processeur p ;
- pour le bloc-colonne k, $1 \leq k \leq N$, le symbole $\star$ signifie $\{j \in BCol(k)\}$;
- Soit $j \geq k$; la notation $[j]$ signifie $\{i \in BCol(k) \cup \{k\}$ avec $i \geq j\}$;
- AUB_{ij} : tout bloc agrégé de contribution pour le bloc A_{ij}. Nous noterons par extension $\text{AUB}_{[j]j}$ l'ensemble compacté des blocs agrégés de contribution d'un processeur donné pour le bloc-colonne j.

Les calculs sur les blocs peuvent ainsi être classés en quatre types élémentaires : `COMP_1D` qui assure le traitement d'un bloc-colonne dans le cas d'une distribution 1D, et `FACTOR - BDIV - BMOD`, qui sont les tâches élémentaires pour le cas d'une distribution 2D. Les tâches associées sont alors définies comme suit dans la factorisation version $L.D.L^t$ (les lignes marquées d'un nombre entre parenthèses sont celles qui changent pour avoir une version $L.L^t$) :

– `COMP1D(k)` : *factoriser le bloc-colonne k et calculer toutes les contributions destinées aux blocs-colonnes de $BCol(k)$*

. Factoriser A_{kk} en $L_{kk}D_kL_{kk}^t$; (1)
. Résoudre $L_{kk}F_\star^t = A_{\star k}^t$ et $D_kL_{\star k}^t = F_\star^t$; (2)
. **Pour** $j \in BCol(k)$ **Faire**
. Calculer $C_{[j]} = L_{[j]k}F_j^t$; (3)
. **Si** $map([j], j) == p$ **Alors** $A_{[j]j} = A_{[j]j} - C_{[j]}$
. **Sinon** $\mathrm{AUB}_{[j]j} = \mathrm{AUB}_{[j]j} + C_{[j]}$;

– `FACTOR(k)` : *factoriser le bloc diagonal k*

. Factoriser A_{kk} en $L_{kk}D_kL_{kk}^t$; (4)

– `BDIV(j,k)` : *mettre à jour le bloc extra-diagonal j dans le bloc-colonne k*

. Résoudre $L_{kk}F_j^t = A_{jk}^t$ et $D_kL_{jk}^t = F_j^t$; (5)

– `BMOD(i,j,k)` : *calculer la contribution du bloc i dans le bloc-colonne k pour le bloc i dans le bloc-colonne j (voir figure 3.3)*

. Calculer $C_i = L_{ik}F_j^t$; (6)
. **Si** $map(i, j) == p$ **Alors** $A_{ij} = A_{ij} - C_i$
. **Sinon** $\mathrm{AUB}_{ij} = \mathrm{AUB}_{ij} + C_i$;

L'algorithme de la factorisation $L.D.L^t$ pour le processeur p écrit par tâches est décrit dans la figure 3.4, la routine `Phase_Envoi()` est elle décrite dans la figure 3.5 :

Pour $n = 1$ à TNb_p **Faire**
  Selon (le type de $K_p[n]$) **Faire**
    `COMP1D` : Recevoir et ajouter tous les $\mathrm{AUB}_{[k]k}$ dans $A_{[k]k}$;
      `COMP1D(k)` ;
      `Phase_Envoi()` ;
    `FACTOR` : Recevoir et ajouter tous les AUB_{kk} dans A_{kk} ;
      `FACTOR(k)` ;
      envoyer $L_{kk}D_k$ à tous les processeurs dans $map([k], k)$;
    `BDIV` : Recevoir $L_{kk}D_k$ et recevoir et ajouter les AUB_{jk} pour A_{jk} ;
      `BDIV(j,k)` ;
      envoyer F_j^t à tous les processeurs dans $map([j], k)$;
    `BMOD` : Recevoir F_j^t ;
      `BMOD(i,j,k)` ;
      `Phase_Envoi()` ;

FIG. 3.4 – Algorithme de la factorisation parallèle.

Les changements dans l'algorithme pour une version $L.L^t$ seront :

- (1) $\rightarrow$ Factoriser A_{kk} en $L_{kk}L^t_{kk}$
- (2) $\rightarrow$ Résoudre $L_{kk}L^t_{\star} = A^t_{\star k}$
- (3) $\rightarrow$ Calculer $C_{[j]} = L_{[j]k}L^t_{jk}$
- (4) $\rightarrow$ Factoriser A_{kk} en $L_{kk}L^t_{kk}$
- (5) $\rightarrow$ Résoudre $L_{kk}L^t_{jk} = A^t_{jk}$
- (6) $\rightarrow$ Calculer $C_i = L_{ik}L^t_{jk}$
- Dans les lignes 8 et 9 de l'algorithme parallèle, il faut changer $L_{kk}D_k$ en L_{kk} ; de même dans les lignes 11 et 12 il faut changer F^t_j en L^t_{jk}.

Sur un processeur p donné, K_p est *le vecteur des tâches de calcul local ordonnées par priorité* (ligne 2 sur la figure 3.4). Chaque tâche doit avoir reçu toutes ses contributions pour pouvoir débuter. Une nouvelle tâche ne pourra commencer que lorsque la tâche courante aura fini de calculer et de mettre à jour toutes les données locales qu'elle doit modifier. Quand un bloc d'agrégation AUB reçoit sa dernière contribution, on dit que celui-ci est "complétement agrégé" et est donc prêt à être envoyé. Un ordonnancement efficace assure que l'ordre d'envoi des AUBs correspond à l'ordre de réception sur le processeur destinataire. En effet, ceci permettra de ne pas retarder le calcul de tâches urgentes par la mise-à-jour de tâches moins prioritaires. C'est pourquoi nous introduisons, pour la `Phase_Envoi` (voir figures 3.4 et 3.5) cette propriété notée $\mathcal{P}$ qui sera vraie pour un AUB quand son ordre d'envoi correspond à l'ordre de traitement des tâches destinataires. Dans le paragraphe 4.3, nous expliciterons une définition de $\mathcal{P}$ qui vérifie cette propriété. Si l'on note S_q l'ensemble, à un instant donné, de tous les AUBs prêts à être envoyés au processeur q, l'algorithme de la phase d'envoi est alors :

`Phase_Envoi()` :
. **Pour** $q = 1$ **à** P, $q \neq p$, **Faire**
.   **Si** $S_q \neq \emptyset$ **Alors**
. $E_q = \{\text{AUB} \in S_q \ / \ \mathcal{P}(\text{AUB}) \text{ est vraie}\}$;
.   Envoyer E_q à q ; $S_q = S_q \setminus E_q$;

FIG. 3.5 – Algorithme de la phase d'envoi.

3.4 Modélisation de la plate-forme

La distribution que nous utilisons repose sur une simulation des temps des tâches de calcul et des communications au fur-et-à-mesure de la distribution. Cette simulation pour être précise doit s'appuyer sur une modélisation aussi précise que possible des temps de communications point-à-point et des opérations BLAS.

Les modélisations que nous allons décrire dans ce paragraphe, portent essentiellement sur l'IBM SP2 du CINES à Montpellier et sur l'IBM SP3 qui le remplaça plus tard. Tous les résultats qui sont présentés dans les chapitres 4 et 5 ont été obtenus sur ces machines. Voici leurs caractéristiques :

- IBM SP2 : 192 nœuds monoprocesseurs ; les processeurs sont des 120 MHz Power2SC thin node (480 MFlops de puissance crête) avec 256 MBytes de mémoire physique par nœud ; l'interface réseau repose sur un switch TB3 permettant un débit maximal de 110 MBytes/seconde.
- IBM SP3 : 28 nœuds SMP NH2 comprenant 16 Powers 3 (1.5 GFlops crête) et 16 Go de mémoire partagée ; soit au total pour 448 processeurs, 672 GFlops crête et 448 Go de mémoire ; l'interface réseau des nœuds repose sur un switch COLONY assurant un débit maximal de 420 MBytes/seconde.

Les opérations BLAS dont nous avons besoin sont essentiellement :
- le `PPF`(ou le `POF`) : factorisation $L.D.L^t$ (ou $L.L^t$) d'une matrice pleine,
- le `TRSM` : résolution d'un système triangulaire, éventuellement multi seconds-membres,
- le `GEMM` : multiplication de deux matrices,
- le `GEAD` : addition de deux matrices.

Les bibliothèques BLAS, généralement fournies par le constructeur d'une machine, utilisent intensivement les effets pipelines permis notamment par les architectures superscalaires. Nous utilisons une modélisation de ces primitives basée sur l'expérimentation. Cela consiste dans un premier temps à mesurer les temps d'exécution d'une routine sur un échantillon de taille représentatif des tailles généralement rencontrées dans le calcul, puis à rechercher un polynôme de même degré que celui de la formule du nombre d'opérations scalaires effectuées par la routine. Par exemple pour le `POF`, le nombre d'opérations nécessaires à la factorisation $L.L^t$ d'une matrice pleine de rang n est :

$$\frac{n^3}{3} + \frac{n^2}{2} - \frac{5.n}{6} \quad .$$

Nous cherchons donc à modéliser les mesures expérimentales de temps avec un polynôme de la forme :

$$a\, n^3 + b\, n^2 + c\, n + d \quad .$$

Sur le SP3 du CINES, en réel double précision, la régression polynomiale donne les valeurs suivantes :

$$a \approx 0.2353.10^{-9};\ b \approx 0.1468.10^{-7};\ c \approx 0.307.10^{-6};\ d \approx 0.309.10^{-4}.$$

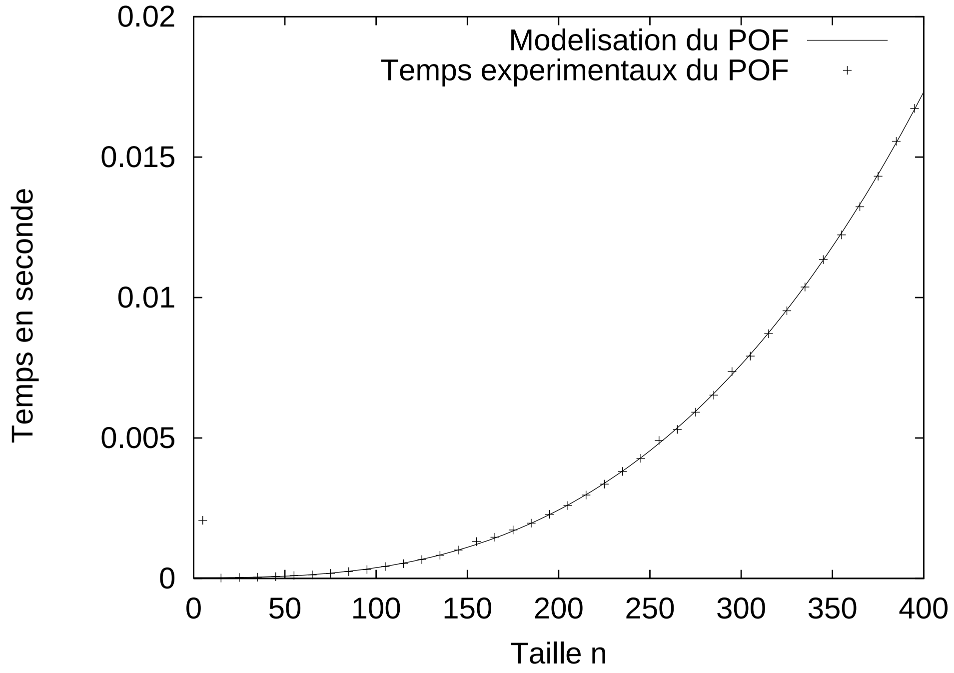


FIG. 3.6 – Correspondance entre la modélisation du POF et les résultats expérimentaux

Ces cœfficients prouvent, comme on s'y attendait, que les temps des BLAS ne sont pas proportionnels au nombre d'opérations théoriques. La figure 3.6 montre que le modèle obtenu est très proche de la réalité.

La modélisation donne aussi une très bonne approximation pour des opérations dépendant de plusieurs paramètres comme le GEMM par exemple. Dans la factorisation Cholesky-Crout, le GEMM est utilisé pour la multiplication d'une matrice de dimensions $(p \times q)$ par une matrice de dimensions $(q \times r)$. Le nombre d'opérations arithmétiques est donc : 2 $p.q.r$. Nous cherchons alors à modéliser les mesures expérimentales par un polynôme de la forme :

$$a\, p.q.r + b\, p.r + c\, p.q + d\, r.q + e\, p + f\ r + g\ q + h \ \ .$$

Sur le SP3 du CINES, en réel double précision, la régression polynomiale donne les valeurs suivantes :

$$a \approx 0.147.10^{-8};\ b \approx 0.123.10^{-7};\ c \approx 0.110.10^{-7};\ d \approx 0.123.10^{-8}$$
$$e \approx 0.527.10^{-8};\ f \approx 0.380.10^{-6};\ g \approx 0.185.10^{-6};\ h \approx 0.406.10^{-5}\ .$$

La figure 3.7 montre la nette correspondance du modèle par rapport aux résultats expérimentaux (pour cette représentation on a fixé la dimension q à 60).

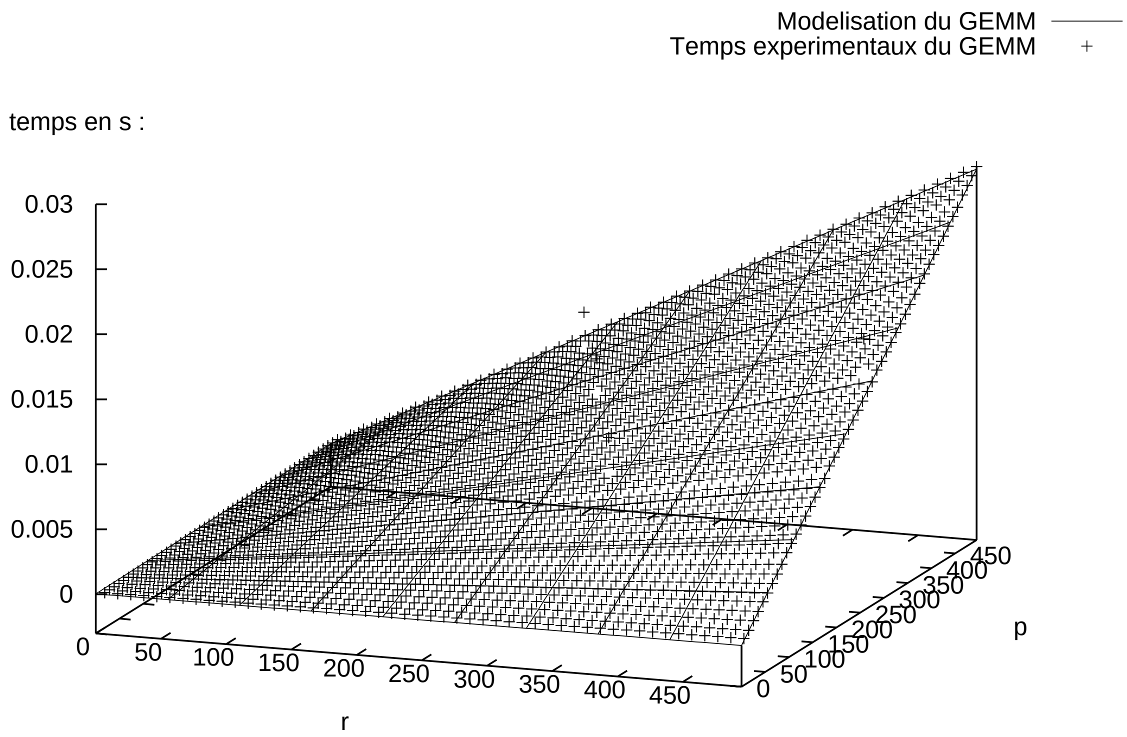


Fig. 3.7 – Modélisation du `GEMM` : multiplication entre une matrice $(p \times q)$ et une matrice $(q \times r)$. Pour la représentation, on a fixé $q = 60$.

Les communications utilisées dans le solveur sont de type asynchrone ; leur implémentation utilise l'interface MPI. Sur les machines à nœuds SMP comme le SP3, la bibliothèque MPI utilise de manière optimisée la mémoire partagée pour les communications intra-nœud et le réseau pour les communications extra-nœud. Pour une communication asynchrone en point-à-point, le temps mis par l'émetteur pour "reprendre la main" est égal à la latence β et le temps t qu'il faut au message pour atteindre le récepteur est donné par la relation classique $t = \beta + \lambda.l$ où λ représente le débit et l la longueur en octets du message. Comme le montre la figure 3.8 pour le SP3 (et pour des échanges asynchrones), la latence et le débit varient spécifiquement selon que les communications se passent de manière intra ou extra-nœud, et suivant le nombre de communications simultanées.

La dégradation des performances des échanges MPI est propre à l'architecture en nœuds SMP : dans le cas des communications extra-nœuds, l'explication réside dans le fait que l'ensemble des processeurs d'un même nœud utilisent la même interface réseau du switch pour communiquer vers l'extérieur ; dans les cas des communications intra-nœuds, la dégradation est causée par la concurrence des accès en mémoire.

Ainsi dans l'étape de distribution/ordonnancement nous utilisons plusieurs valeurs de β et de λ pour estimer les temps de communication suivant la localisation des processeurs et leur nombre dans le calcul d'un super-nœud (voir paragraphe 4.5).

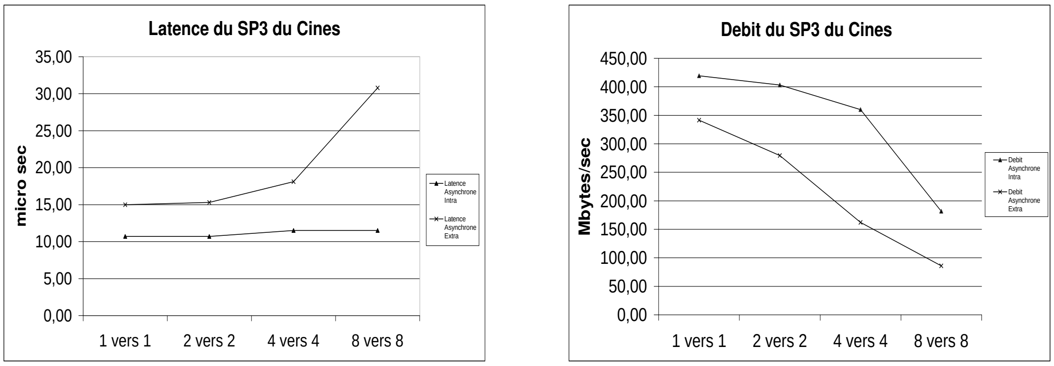


FIG. 3.8 – Mesure de la latence et du débit sur le SP3 du CINES

3.5 Présentation de nos cas tests

TAB. 3.1 – Description des cas tests. $\mathrm{NNZ_A}$ est le nombre de termes non nuls dans la partie triangulaire inférieure de la matrice A, $\mathrm{NNZ_L}$ est le nombre de termes non nuls dans la matrice L et OPC est le nombre d'opérations requises pour la factorisation. Les matrices sont triées par ordre suivant le rapport $\frac{\mathrm{NNZ_L}}{\mathrm{OPC}}$ qui est une mesure de la réutilisabilité potentielle des données [60] (plus ce rapport est petit plus on peut s'attendre à un bon comportement du code).

Nom	Colonnes	$\mathrm{NNZ_A}$	$\mathrm{NNZ_L}$	OPC	$\frac{\mathrm{NNZ_L}}{\mathrm{OPC}}$	Description
GRID1023	1046529	4179980	5.615e+07	2.083481e+10	2.69e-3	Regular 2D Mesh
CUBE39	59319	730778	2.210e+07	2.240674e+10	0.99e-3	Regular 3D Mesh
CUBE47	103823	1290898	4.828e+07	6.963850e+10	0.69e-3	Regular 3D Mesh
BCSSTK32	44609	985046	5.239e+06	1.162900e+09	4.50e-3	Rutherford-Boeing
BBMAT	38744	1274141	1.716e+07	1.250040e+10	1.37e-3	Rutherford-Boeing
TOOTH	78136	452591	1.031e+07	6.267094e+09	1.64e-3	3D element Mesh
OCEAN	143437	409593	2.029e+07	1.301477e+10	1.56e-3	3D element Mesh
M14B	214765	1679018	6.236e+07	6.112540e+10	1.02e-3	3D element Mesh
OILPAN	73752	1761718	8.912e+06	2.984944e+09	2.98e-3	PARASOL
QUER	59122	1403689	9.118e+06	3.280680e+09	2.78e-3	PARASOL
INVEXTR1	30412	906915	7.256e+06	3.766788e+09	1.93e-3	PARASOL
SMDOOR	162610	3873534	2.541e+07	1.530774e+10	1.66e-3	PARASOL
SHIP001	34920	2304655	1.427e+07	9.033767e+09	1.58e-3	PARASOL
X104	108384	5029620	2.634e+07	1.712902e+10	1.54e-3	PARASOL
MT1	97578	4827996	3.114e+07	2.109265e+10	1.48e-3	PARASOL
BMW3_2	227362	5530634	4.420e+07	3.007981e+10	1.47e-3	PARASOL
MIXTANK	29957	982542	9.280e+06	7.316933e+09	1.26e-3	PARASOL
BMWCRA_1	148770	5247616	6.597e+07	5.701988e+10	1.16e-3	PARASOL
CRANKSG1	52804	5280703	3.142e+07	3.007141e+10	1.05e-3	PARASOL
SHIPSEC8	114919	3269240	3.572e+07	3.684269e+10	0.97e-3	PARASOL
CRANKSG2	63838	7042510	4.190e+07	4.602878e+10	0.91e-3	PARASOL
SHIPSEC5	179860	4966618	5.649e+07	6.952086e+10	0.81e-3	PARASOL
SHIP003	121728	3982153	5.872e+07	8.008089e+10	0.73e-3	PARASOL
THREAD	29736	2220156	2.404e+07	3.884020e+10	0.62e-3	PARASOL
BMWCRA_1	148770	5247616	6.597e+07	5.701988e+10	1.16e-3	PARASOL
AUDI	943695	39e+06	1.214e+09	5.266e+12	2.30e-4	PARASOL
COLOGB30	20373	1394292	7.849e+06	4.359803e+09	1.80e-3	3D Cologne
COUP8000T	5.3e+6	374.5e+06	32.128e+08	20.90e+11	1.54e-3	3D Coupole
COUP5000T	3.3e+6	233.2e+06	20.269e+08	13.10e+11	1.54e-3	3D Coupole
COLOGB75	50208	3473292	2.248e+07	1.532035e+10	1.47e-3	3D Cologne
COUP1500T	994983	70303275	5.219e+08	3.79847e+11	1.38e-3	3D Coupole
COUP2000T	1326483	93734775	6.888e+08	5.00924e+11	1.37e-3	3D Coupole
COUP3000T	1989966	140.6e+06	10.333e+08	7.68324e+11	1.35e-3	3D Coupole

Les algorithmes de repartitionnement et de distribution (que nous allons présenter dans le chapitre 4) ainsi que le solveur parallèle, ont été implémentés dans la bibliothèque PaStiX [46,47,49,50] qui fait appel à la version 3.4 du logiciel Scotch [64, 65] pour la renumérotation des matrices creuses.

La table 3.1 présentent les caractéristiques de nos matrices tests et les résultats de leur renumérotation par Scotch. Pour les grilles 2D et 3D, nous n'utilisons pas le logiciel SCOTCH mais l'algorithme de bissection récursive de George and Liu [37] qui est optimal pour ces cas. Les autres matrices creuses viennent de "Rutherford-Boeing Collection" (*http ://math.nist.gov/MatrixMarket/collections*), "PARASOL ESPRIT Project" (*http ://www.parallab.uib.no/parasol*) et du CEA/CESTA (3D Cologne Challenge, Coupole). La plupart de ces matrices sont des matrices de mécanique des structures et de mécanique des fluides. Les matrices BBMAT et MIXTANK ne sont pas des systèmes symétriques à l'origine, nous avons traité ces systèmes en considérant le graphe de la matrice symétrisée $A + A^t$. Certaines d'entre elles dépassent largement le million d'inconnues et le téraopérations pour la factorisation.

Chapitre 4

Algorithmes de repartitionnement et de distribution

4.1 Justification de certains choix algorithmiques

Comme nous l'avons vu dans le paragraphe 3.2, nous avons découplé le problème du repartitionnement et de la distribution. Lors du repartitionnement, les super-nœuds de l'arbre d'élimination par blocs sont redécoupés, et on leur assigne un groupe de processeurs candidats. Le groupe de processeurs candidats est le même pour tous les blocs-colonnes issus du repartitionnement d'un super-nœud ; il représente l'ensemble des processeurs qui auront le droit de participer au calcul pipeliné du super-nœud.

L'intérêt de cette méthode découplant la phase de repartitionnement et celle de distribution, et utilisant des groupes de processeurs candidats est multiple comme on le verra plus précisemment dans la suite :

- la repartition récursive des processeurs candidats favorisera la localité des communications ;
- la parallélisation de la factorisation d'un super-nœud pourra ne pas utiliser tous ses processeurs candidats. En effet, le principe de l'algorithme de distribution fera qu'un processeur ne participe au calcul d'un super-nœud que si cela améliore les temps de calcul au sens de la simulation ;
- on ne rencontre pas les problèmes d'arithmétique entière que l'on trouve généralement dans les méthode de "proportionnal mapping". En effet, on autorise un processeur candidat à se "dédoubler" dans deux sous-arbres pour lesquels la répartition en deux groupes de processeurs candidats ne tombe pas juste. Ceci sera possible car il ne s'agit que d'un candidat, et ce sera uniquement lors de la phase de distribution qu'il pourra être utilisé plus ou moins dans chacun des sous-arbres.

Le découpage du bloc-colonne correspondant à un super-nœud dans la matrice se fait avec une taille de bloc qui est suffisamment grande pour tirer parti efficacement des effets BLAS et pour bien recouvrir les communications par du calcul, et suffisamment petite pour permettre d'alimenter un pipeline de calcul sur plusieurs processeurs. Ce *facteur de blocage* dépend donc des performances des processeurs et du réseau de la machine ; un bon facteur de blocage peut être trouvé par l'expérimentation sur des matrices pleines et est généralement connu pour une machine donnée (valeur utilisée dans des bibliothèques comme ScalaPack ou PESSL [1,3,20] sur machine IBM). Sur le SP3 du CINES, 60 est une bonne taille de découpage pour les blocs-colonnes à distribuer en 1D, mais dans pour une distribution 2D un facteur de blocage de 120 est plus approprié. Ceci s'explique par le fait que dans une distribution 1D, une largeur de bloc-colonne de 60 permet d'obtenir suffisament de puissance de calcul par les BLAS grâce au *compactage* en mémoire (et donc au calcul global) des blocs extra-diagonaux, et ce tout en conservant une granularité de calcul suffisante pour le parallélisme. Par contre, une distribution 2D génère suffisamment de tâches de calcul pour le parallélisme, mais il est nécessaire d'augmenter la taille des blocs pour obtenir suffisament de puissance de calcul dans les primitives BLAS afin de recouvrir efficacement les communications.

Une fois que la phase de repartitionnement est terminée, il faut construire le graphe des tâches de calcul. Comme nous l'avons vu dans le chapitre 3, on peut décider qu'un super-nœud sera parallélisé selon un schéma 1D (on associe aux blocs-colonnes issus de la répartition du super-nœud des tâches de type `COMP_1D` décrites dans le paragraphe 3.3), ou selon un schéma 2D (on associe aux blocs repartitionnés du super-nœud des tâches `FACTOR`, `BDIV` ou `BMOD` suivant le cas). En règle générale et de manière théorique, plus le nombre de processeurs est important, plus une parallélisation selon un schéma 2D est payante par rapport à une parallélisation en 1D [78]. Nous avons constaté expérimentalement que sur une machine donnée et pour un nombre de processeurs qui n'est pas aberrant par rapport à la taille du problème, *le nombre de processeurs candidats* était un bon critère pour passer d'une distribution 1D à une distribution 2D. Comme on pouvait s'y attendre (voir paragraphe 3.2), dans le cas d'un nombre "assez petit" de processeurs candidats et pour des blocs-colonnes de tailles "assez petites", la meilleure utilisation des BLAS, favorisée par le compactage des données en mémoire, fait qu'une distribution 1D est gagnante sur une distribution 2D. Les tableaux 4.1 et 4.2 présentent des temps de factorisation sur quelques cas tests représentatifs (voir paragraphe 3.5 pour les caractéristiques de ces matrices) en utilisant différentes valeurs de seuil de nombre de processeurs candidats pour passer d'un type de distribution à l'autre.

Nous voyons que sur le SP2, les meilleurs temps sont obtenus pour un seuil de l'ordre de 12 processeurs candidats ; comme le montre le tableau 4.2 pour le SP3,

TAB. 4.1 – Influence du seuil de basculement entre distributions 1D et 2D, en nombre de processeurs candidats, sur les temps de factorisation exprimés en secondes sur IBM SP2.

Nom	Seuil	Nombre de processeurs			
		8	16	32	64
CUBE47	6	39.59	27.17	17.82	14.77
	8	**38.61**	25.08	**17.05**	13.53
	12	-	**24.08**	17.15	**12.59**
	16	-	25.14	19.54	15.21
BMWCRA_1	6	**31.03**	18.91	9.59	7.01
	8	31.12	19.06	9.62	6.56
	12	-	**18.75**	**9.55**	**5.94**
	16	-	19.11	9.55	6.11
COLOGB75	6	10.18	5.75	4.16	3.25
	8	**8.99**	5.84	4.09	3.34
	12	-	**5.69**	**3.89**	3.14
	16	-	5.85	4.02	**3.13**

TAB. 4.2 – Influence du seuil de basculement entre distributions 1D et 2D, en nombre de processeurs candidats, sur les temps de factorisation exprimés en secondes sur IBM SP3.

BMWCRA1	Nombre de processeurs			
Seuil	16	32	64	128
4	5.16	3.30	2.17	1.65
8	5.02	3.07	1.93	1.64
12	**4.84**	**2.95**	1.91	**1.52**
16	4.92	3.00	**1.88**	1.59
18	-	2.96	2.79	2.44

THREAD	Nombre de processeurs			
Seuil	16	32	64	128
4	5.52	3.38	2.72	2.54
8	4.32	3.33	2.69	2.56
12	**4.16**	3.12	2.77	**2.40**
16	4.26	3.01	**2.66**	2.45
18	-	**2.76**	2.79	2.44

CUBE47	Nombre de processeurs			
Seuil	16	32	64	128
4	7.24	5.00	4.10	3.50
8	7.13	4.90	3.83	3.44
12	7.08	4.94	3.91	**3.34**
16	**7.00**	4.81	**3.90**	3.69
18	-	**4.84**	3.85	3.77

SHIP003	Nombre de processeurs			
Seuil	16	32	64	128
4	7.83	5.51	3.75	2.92
8	8.12	5.10	3.60	2.95
12	7.78	4.82	**3.49**	**2.88**
16	**7.49**	4.88	3.70	3.11
18	-	**4.79**	3.66	2.86

un bon compromis entre puissance et scalabilité semble aussi se situer pour un seuil de 12 processeurs.

Toutes les tâches de calculs ont pour ensemble de processeurs candidats celui du super-nœud dont elles sont issues. Une fois ce graphe de tâches établi, l'algorithme de distribution et d'ordonnancement a pour rôle de distribuer chacune de ces tâches sur un de leurs processeurs candidats. Nous utilisons pour cela une simulation en temps de l'algorithme parallèle dont l'algorithme (voir paragraphe 4.3) est basé sur le principe qui consiste à distribuer la première tâche prête sur le processeur qui pourra la traiter le plus vite au sens des temps de la simulation. On peut rapprocher le principe de cet algorithme de celui des algorithmes de distribution à la volée utilisés dans les solveurs pour machine à mémoire partagée (voir paragraphe 2.2).

4.2 Repartitionnement des super-nœuds

L'algorithme de repartitionnement (voir figure 4.1) est basé sur la technique de "proportionnal mapping" introduite dans le paragraphe 2.2 : la répartition des processeurs candidats se fait récursivement du haut vers le bas de l'arbre d'élimination par blocs proportionnellement aux coûts des sous-arbres enracinés sous un même nœud.

Pour décrire cet algorithme, nous introduisons les notations suivantes :
- p_k : processeur k, $1 \leq k \leq P$;
- C_k : somme de travail restant à affecter pour le processeur k ;
- $\mathcal{A}$: l'arbre d'élimination par blocs initial ;
- $A_\star$: un sous-arbre de $\mathcal{A}$;
- $Cand(R)$: ensemble de processeurs candidats pour le super-nœud R ;
- $Cand(A)$: unique processeur candidat pour l'ensemble des super-nœuds du sous-arbre A ;
- coût(A) : coût de factorisation selon la modélisation en temps des BLAS de tous les blocs-colonnes du sous-arbre A en utilisant un algorithme séquentiel par blocs.

Etape 0. *Initialisation*
    Pour $l = 1$ **à** P **faire**
        $C_k \leftarrow \frac{\text{coût}(\mathcal{A})}{P}$;
    $\mathcal{P} \leftarrow \{p_1, ..., p_P\}$;
    $\mathcal{C} \leftarrow \{C_1, ..., C_P\}$;
    Repartitionne($\mathcal{A}$,$\mathcal{P}$,$\mathcal{C}$) ;
Etape 1. *Descente récursive de l'arbre d'élimination*
Procédure : Repartitionne(A, $\mathcal{P} = \{p_r, .., p_s\}$, $\mathcal{C} = \{C_r, .., C_s\}$)
    Si $r = s$ **alors**
        Cand(A) $\leftarrow \{p_r\}$;
        Retour de la procédure ;
    $R \leftarrow$ racine(A) ;
    Si $C_r = 0$ **alors** $r \leftarrow r + 1$;
    Si $C_s = 0$ **alors** $s \leftarrow s - 1$;
    Cand(R) $\leftarrow \mathcal{P} = \{p_r, .., p_s\}$;
    Découpage de R selon le facteur de blocage ;
    Réestimation des coûts des blocs repartitionnés ;
    $\alpha \leftarrow \frac{\text{nouveau coût}(A)}{\text{ancien coût}(A)}$;
    Pour $l = r$ **à** s **faire**
        $C_l \leftarrow C_l - \frac{\text{ancien coût}(R)}{s-r+1}$;
        $C_l \leftarrow \alpha.C_l$;
        Si $C_l \leq 0$ **alors** $C_l \leftarrow 0$;
    On crée des groupes de processeurs proportionnellement aux coûts des sous-arbres de R
    Soient $A_1, .., A_t$ les t sous-arbres de A enraçinés en R ;
    $k \leftarrow r$;
    Pour $i = 1$ **à** t **faire**
        $S \leftarrow 0$; $\mathcal{C}_i \leftarrow \emptyset$; $\mathcal{P}_i \leftarrow \emptyset$;
        Tant que $S + C_k \leq$ coût(A_i) **faire**
            $S \leftarrow S + C_k$;
            $\mathcal{P}_i \leftarrow \mathcal{P}_i \cup \{p_k\}$;
            $\mathcal{C}_i \leftarrow \mathcal{C}_i \cup \{C_k\}$;
            $k \leftarrow k + 1$;
        Si ($S <$ coût(A_i)) **alors**
            $\mathcal{P}_i \leftarrow \mathcal{P}_i \cup \{p_k\}$;
            $\tilde{C}_k \leftarrow$ coût$(A_i) - S$;
            $\mathcal{C}_i \leftarrow \mathcal{C}_i \cup \{\tilde{C}_k\}$;
            $C_k \leftarrow C_k - \tilde{C}_k$;
        Repartitionne(A_i, $\mathcal{P}_i$, $\mathcal{C}_i$)

FIG. 4.1 – Algorithme de repartitionnement et de formation des groupes de processeurs candidats.

La principale caractéristique de notre algorithme est de permettre à un processeur de participer dans deux sous-arbres adjacents pour éviter les problèmes d'arithmétique entière que l'on rencontre traditionnellement dans ce type de méthode. Nous appelons un tel processeur candidat un *processeur dédoublé*. Ceci est rendu possible par le fait qu'un processeur dédoublé n'est qu'un candidat pour les super-nœuds des deux sous-arbres ; lors de la phase de distribution, le processeur pourra participer au calcul des deux sous-arbres dans des proportions qui équilibrent leurs temps de calcul. La mise en œuvre algorithmique de ce procédé est assurée comme suit : avant de commencer la descente récursive de l'arbre délimination par blocs, on associe à chaque processeur candidat p_k, $1 \leq k \leq P$, la valeur C_k du travail pour laquelle il devra candidater (lignes 1 et 2). Puis au fur-et-à-mesure de la descente, on retranche à cette valeur la charge de calcul pour laquelle il est candidat (ligne 18). Cette valeur peut devenir négative pour un processeur "dédoublé" car il se peut que sa charge de travail dans un sous-arbre soit trop faible pour être candidat jusqu'en bas d'un sous-arbre ; dans ce cas, on met sa charge de travail à 0 (ligne 20) et ce processeur ne sera plus utilisé dans la suite de la descente de ce sous-arbre (lignes 11 et 12). La répartition des processeurs candidats sur les sous-arbres d'un nœud suit la numérotation croissante des processeurs selon l'ordre de la gauche vers la droite des sous-arbres (lignes 21 à 34) ; un processeur est autorisé à participer à deux sous-arbres adjacents de telle manière que la somme des travaux à pourvoir à chaque groupe de processeurs candidats soit égale aux coûts des sous-arbres (lignes 30 à 34). La charge à affecter à un processeur dédoublé (notée $\tilde{C}_k$ dans l'algorithme) dans un sous-arbre est alors calculée pour que la somme des travaux à affecter aux processeurs candidats soit égale au coût de ce sous-arbre (lignes 32).

Une autre spécificité de notre algorithme tient dans la correction des coûts des sous-arbres au fur-et-à-mesure du découpage des blocs. La figure 4.2 montre les blocs redécoupés selon le facteur de blocage lors du repartitionnement d'un super-nœud.

Comme on le voit sur cette figure, les blocs extra-diagonaux situés en face du bloc diagonal du bloc-colonne repartitionné doivent aussi être repartitionnés pour la cohérence des structures de données par blocs avec l'algorithme parallèle de factorisation. Ces blocs extra-diagonaux appartiennent à des blocs-colonnes associés à des super-nœuds qui se situent dans des sous-arbres qui seront rencontrés en descendant l'arbre d'élimination par blocs. Or nous avons vu dans le paragraphe 3.4 que le coût d'une opération élémentaire BLAS sur des blocs n'est pas linéaire par rapport au nombre d'opérations arithmétiques qu'elle nécessite. Ainsi, et afin de conserver une bonne répartition des processeurs candidats, il est donc nécessaire de corriger l'estimation, selon le modèle des BLAS (voir paragraphe 3.4), du coût des blocs repartitionnés (et donc des sous-arbres qui les contiennent) qui servent à la formation des ensembles de processeurs candidats au cours du repartitionnement (voir lignes

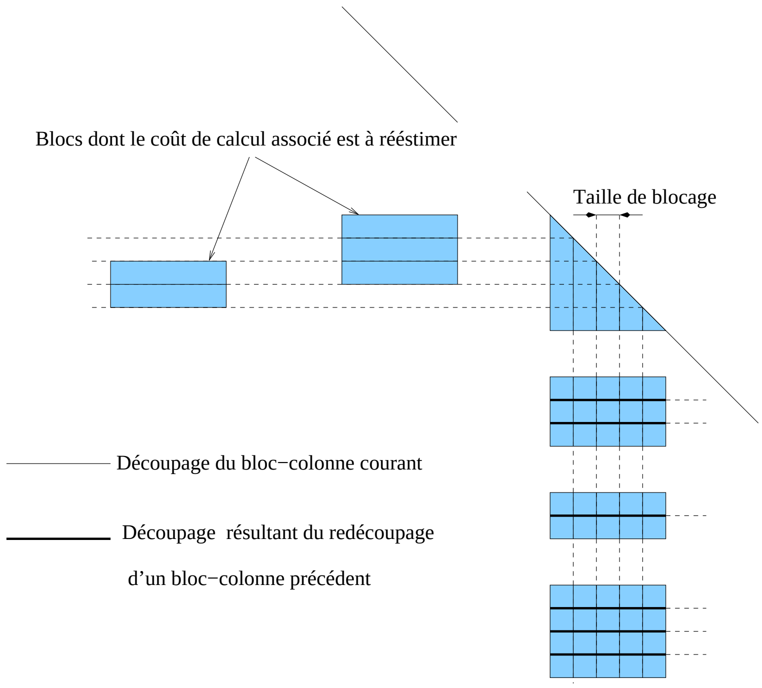


FIG. 4.2 – Découpage d'un bloc-colonne

15, 16, 18 et 19).

4.3 Distribution et ordonnancement

Nous allons maintenant présenter l'algorithme de distribution et d'ordonnancement. La figure 4.4 décrit la distribution des tâches et l'ordonnancement associé. La distribution des blocs de la matrice est induite par celle des tâches suivant les relations illustrées dans la figure 4.3 (les notations ont été introduites dans le paragraphe 3.3) :

- une tâche COMP_1D(k) est liée au bloc-colonne k ;
- une tâche FACTOR(k) est liée au bloc diagonal A_{kk} ;
- une tâche BDIV(j,k) est liée au $j^{\text{ième}}$ bloc extra-diagonal du bloc-colonne k ;

Le cas des tâches de type BMOD est différent ; en effet une tâche BMOD(i, j, k) utilise pour son calcul des blocs dont on a déjà choisi le processeur propriétaire lors de la distribution des tâches BDIV(i,k) et BDIV(j,k). Nous avons donc décidé qu'une tâche BMOD(i, j, k) serait exécutée par le processeur $map(j, k)$ choisi pour la distribution de BDIV(j,k). L'information apportée pour une tâche de type BMOD au cours de cette phase est donc essentiellement son ordonnancement sur son seul processeur candidat.

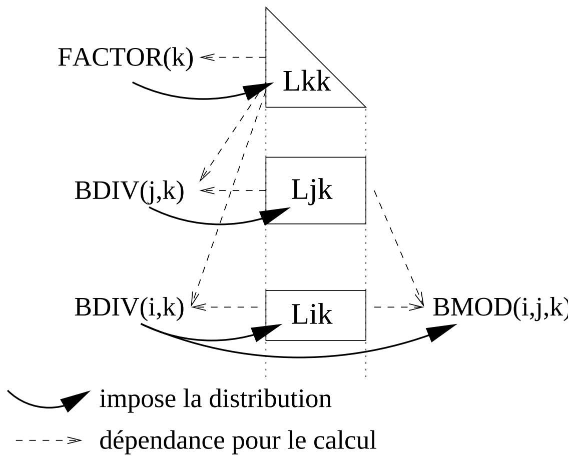


FIG. 4.3 – Dépendance entre la distribution des tâches de calcul et les blocs.

Pour décrire notre algorithme, nous utiliserons les notations suivantes :
- p_l : processeur l, $1 \leq l \leq P$;
- T_l : valeur du temps courant sur le processeur l ;
- K_l : vecteur d'ordonnancement des tâches de calcul sur le processeur l ;
- π_l : priorité courante sur le processeur l (nombre entier égal au nombre de tâches déjà distribuées sur le processeur l) ;
- σ_l : relation d'ordre entre deux tâches prêtes du processeur p_l ;
- R_l : liste des tâches prêtes disponibles sur le processeur l ;
- t_i : tâche de calcul i ;
- $Succ(t_i)$: ensemble des tâches de calculs qui attendent une contribution ou un message de t_i ;
- $Cand(t_i)$: ensemble des processeurs candidats pour la tâche t_i ;
- $map(t_i)$: processeur sur lequel sera exécuté la tâche t_i lors de la factorisation parallèle ;
- $r(t_i, p_j)$: date à laquelle la tâche t_i aura reçu toutes ses contributions si elle est distribuée sur le processeur p_j ;
- $d(t_i, p_j) = \max(r(t_i, p_j), T_j)$: date à laquelle le processeur p_j peut débuter le calcul de la tâche t_i ;
- $\mathrm{calc}(t_i, p_j)$: temps de calcul de la tâche t_i sur le processeur j ;
- $\mathrm{profondeur}(t_i)$: profondeur dans l'arbre d'élimination par blocs du super-nœud associé à la tâche t_i (la raçine est à la profondeur 0 dans l'arbre) ;
- $\mathrm{AUB}(t_i)$: ensemble des blocs de contributions agrégées destinés au bloc associé à la tâche t_i ;

Etape 0. *Initialisation*
1. **Pour** $l = 1$ à P **faire**
2. $T_l \leftarrow 0$;
3. $\pi_l \leftarrow 0$;
4. $R_l \leftarrow$ {les tâches t_k liées aux feuilles de l'arbre telles que $p_l \in Cand(t_k)$} ;

Etape 1. *Sélection d'une tâche à distribuer sur un de ses processeurs candidats*
4. **Pour** $l = 1$ à P **faire**
5. $f_l \leftarrow \min_{\sigma_l}(R_l)$;
6. $d(f_l, p_l) \leftarrow \max(T_l, r(f_l, p_l))$;
7. Soit i tel que $d(f_i, p_i) = \min_{1 \leq l \leq P}\{d(f_l, p_l)\}$;
8. **Pour tout** processeur $p_l \in Cand(f_i)$ **faire**
9. $R_l \leftarrow R_l \setminus \{f_i\}$;
10. $map(f_i) \leftarrow p_i$;
11. $K_i[\pi_i] \leftarrow f_i$;
12. Tous les AUBs $\in$ AUB(f_i) reçoivent π_i pour priorité d'envoi ;
13. $\pi_i \leftarrow \pi_i + 1$;

Etape 2. *Simulation des temps d'exécution de la tâche sélectionnée*
14. $T_i \leftarrow d(f_i, p_i)) + \text{calc}(f_i, p_i)$;
15. **Pour tout** $t_k \in Succ(f_i)$ **faire**
16. **Si** t_k est prête **alors**
17. **Pour tout** processor $p_l \in Cand(t_k)$ **faire**
18. Calculer $r(t_k, p_l)$;
19. $R_l \leftarrow R_l \cup \{t_k\}$;
20. **Si** toutes les tâches ne sont pas distribuées **alors** retour à l'étape 1.

FIG. 4.4 – Algorithme de distribution et d'ordonnancement.

L'idée directrice de cette phase est donc de simuler l'exécution parallèle au fur-et-à-mesure de la distribution. Ainsi pour chaque processeur p_l, nous définissons un compteur du temps courant T_l qui contient, selon la simulation, la valeur du temps consommé par ce processeur p_l après l'exécution de toutes les tâches qui lui ont déjà été assignées. Nous définissons également pour chaque processeur p_l une liste R_l de tâches prêtes ; cette liste contient à un instant donné de la simulation, l'ensemble des tâches dont toutes les contributions ont déjà été calculées et pour lesquelles le processeur p_l est candidat.

Les premières tâches prêtes sont celles qui correspondent au calcul des feuilles de l'arbre d'élimination par blocs puisque celles-ci ne dépendent d'aucune autre tâche (ligne 4 de l'algorithme). L'idée de l'algorithme est de distribuer la tâche qui, parmi toutes les tâches prêtes, peut être calculée au plus tôt par un processeur. Ceci revient à trouver parmi l'ensemble des tâches prêtes le plus tôt de chaque processeur, celle dont le calcul peut débuter le plus tôt. Si on note $r(t_i, p_j)$ la date à laquelle la tâche t_i aura reçu toutes ses contributions si elle est distribuée sur le processeur p_j, alors la date à laquelle la tâche t_i peut débuter au plus tôt sur le processeur p_j est $d(t_i, p_j) = \max(r(t_i, p_j), T_j)$. Mais parmi toutes les tâches prêtes du processeur p_j, il peut exister beaucoup de tâches t_i telle que $t_i < T_j$ (donc le calcul peut donc débuter à la même date $d(t_i, p_j)$), c'est généralement le cas dans la partie inférieure de l'arbre d'élimination, et c'est aussi souvent le cas pour les processeurs dédoublés. C'est pourquoi nous avons introduit la relation d'ordre σ_l qui tient compte de la profondeur du super-nœud, associé à une tâche, dans l'arbre d'élimination par bloc. En effet, entre deux tâches dont le calcul peut débuter à la même date sur un processeur, il est préférable de commencer par celle qui est la plus critique au sens des précédences de calcul données par l'arbre d'élimination. Voici donc la définition de σ_l :

Définition 4.3.1
La relation d'ordre σ_l entre les tâches prêtes du processeur p_l est donnée par

$$t_i \underset{\sigma_l}{<} t_j \Leftrightarrow \begin{cases} T_l < r(t_i, p_l) < r(t_j, p_l) \\ ou \\ r(t_i, p_l) \leq T_l < r(t_j, p_l) \\ ou \\ r(t_i, p_l) \leq T_l \quad et \quad r(t_j, p_l) \leq T_l \quad et \quad profondeur(t_i) > profondeur(t_j). \end{cases}$$

Ainsi le choix de la prochaine tâche et du processeur qui l'exécutera est décidé par la méthode suivante : pour chaque processeur p_l on prend la plus petite tâche prête au sens de σ_l (ligne 5) ; on choisit celle qui pourra être exécutée le plus tôt (ligne 7). La tâche qui a été choisie est ensuite marquée comme déjà distribuée dans les listes de tâches prêtes de tous les processeurs (lignes 8, 9 de l'algorithme). La priorité d'une tâche de calcul, dont on se sert pour ordonnancer les tâches de calculs dans l'algorithme parallèle de factorisation, est le numéro d'ordre π_l dans lequel cette tâche a été distribuée sur le processeur p_l (ligne 11).

Lorsqu'une tâche est distribuée sur un processeur, on met à jour le temps courant sur le processeur (ligne 14). Parmi toutes les tâches qui attendent une contribution de cette tâche, toutes celles dont c'était la dernière contribution en attente sont placées dans la listes des tâches prêtes de leurs processeurs candidats (lignes 15 à 19) selon l'ordre défini par σ ; pour cela on a besoin de calculer, pour chacun des processeurs candidats, la date à laquelle une nouvelle tâche prête aurait reçu toutes ses contributions si elle était distribuée sur ce processeur. Comme le montre la figure 4.5 le schéma de communication des contributions pour une tâche est fonction de la distribution des tâches dont elle dépend. Ainsi en fonction du schéma de contribution et de la modélisation en temps des coûts de communication, on déduit la date à laquelle une tâche est prête pour un de ses processeurs candidats en prenant la date à laquelle la dernière des contributions qu'elle attend pourra être reçue sur ce processeur.

L'algorithme de distribution et d'ordonnancement se termine lorsque toutes les tâches ont été ainsi distribuées.

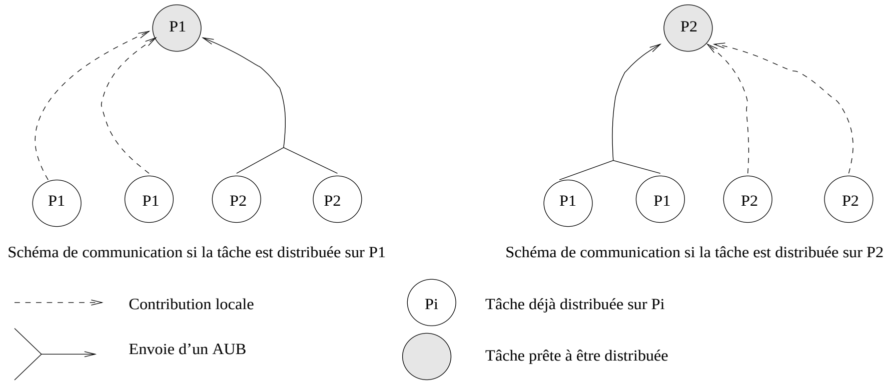


FIG. 4.5 – Schéma de communication des contributions pour une tâche.

L'ordonnancement des tâches de calculs d'un processeur p_l est donc donné par le vecteur K_l. Nous rendons le schéma de communication compatible avec le schéma de calcul en ordonnançant les contributions en fonction de la priorité de leur tâche destinataire. Ainsi sur un processeur donné p, l'ensemble S_q de tous les agrégats de contributions (AUB) destinés au processeur q est ordonné selon la priorité de la tâche destinataire des AUB sur ce processeur q. Nous notons par $S_q[i]$ l'AUB de rang i dans la file d'envoi ; de cette manière la propriété $\mathcal{P}$ que nous avions introduite dans le paragraphe 3.3 pour déterminer les priorités d'envoi des agrégats et qui assure la compatibilité de l'ordre de reception des messages avec l'ordre des calculs est définie par :

$$\mathcal{P}(S_q[i]) \text{ est vraie} \Leftrightarrow \begin{cases} S_q[i] \text{ est complétement agrégé}et \\ \forall j < i, S_q[j] \text{ est complétement agrégé et prêt à être envoyé.} \end{cases}$$

Cette propriété signifie qu'une contribution ne pourra être envoyée que lorsque toutes les contributions destinées à des tâches plus prioritaires sur le processeur destinataire auront été envoyées. Outre l'intérêt d'envoyer les contributions dans l'ordre dans lequel elles seront consommées sur le processeur p_q, cela permet de compacter un ensemble de contributions qui se débloquent (voir algorithme de la phase d'envoi sur la figure 3.5) en un seul message.

Pour illustrer le bien-fondé de nos algorithmes, nous montrons sur les figures 4.6, 4.7, 4.8, le diagramme de Gantt des tâches de calcul et le diagramme des communications prédits par la phase de distribution (ceux du haut) et tracés au cours de la factorisation parallèle (ceux du bas) pour quelques matrices tests ; les exécutions ont été réalisées ici sur SP2. La différence entre les diagrammes de communication prédit et mesuré est due au fait que pour les prédictions, on prend en compte les temps d'arrivée des messages alors que pour les mesures réelles on prend en compte la date à laquelle une contribution arrivée était traitée par l'algorithme. On peut voir sur ces traces que notre algorithme de distribution et d'ordonnancement fournit un schéma d'exécution parallèle efficace conforme à la prédiction, et avec une bonne localité des communications.

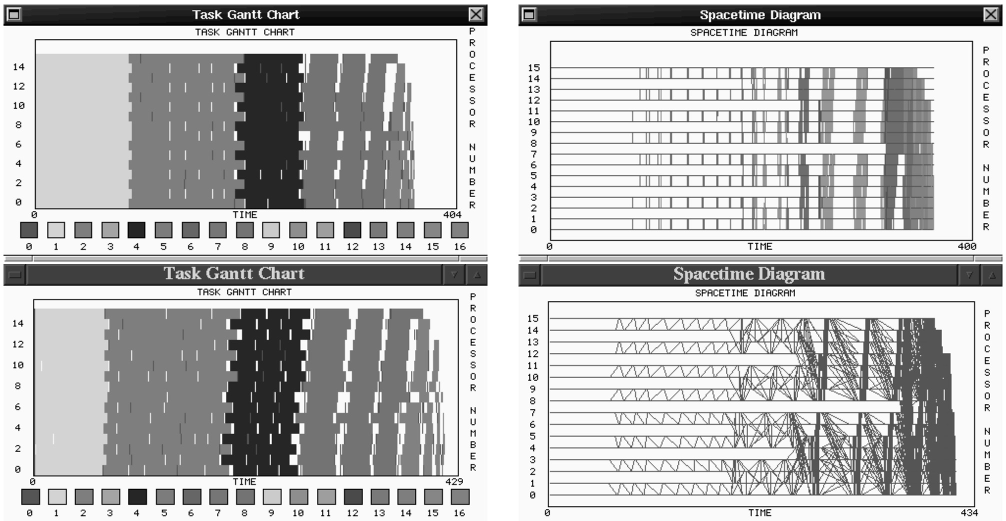


FIG. 4.6 – Diagrammes pour la CUBE 39.

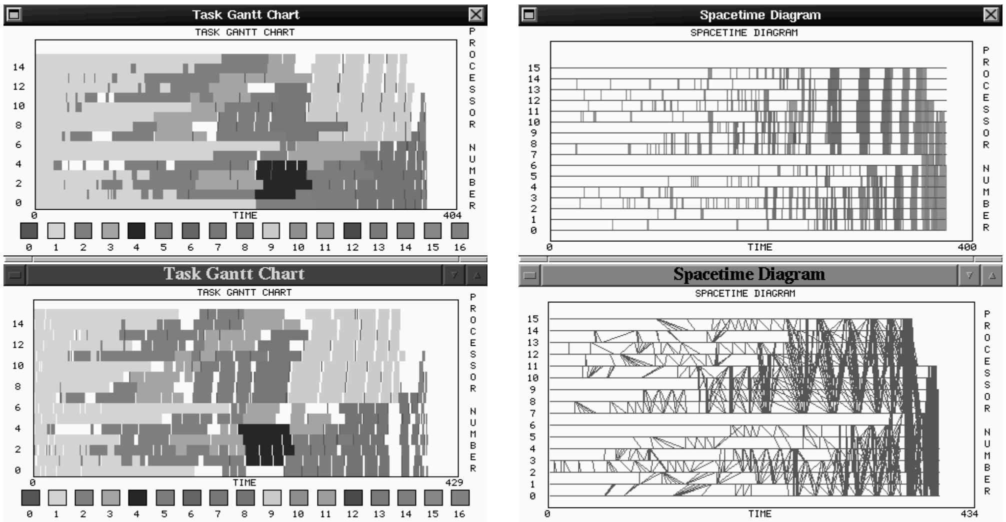


FIG. 4.7 – Diagrammes pour la BMW3.

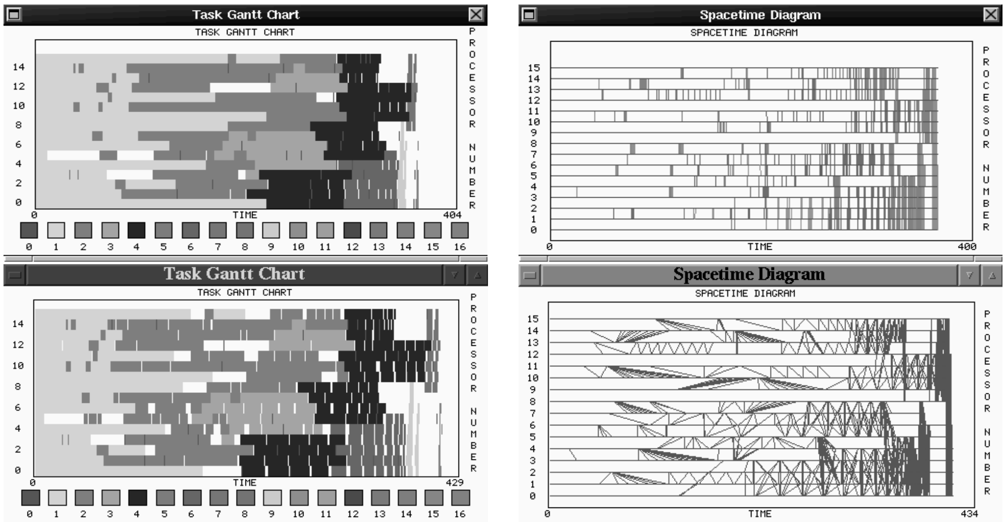


Fig. 4.8 – Diagrammes pour la CRANKSEG1.

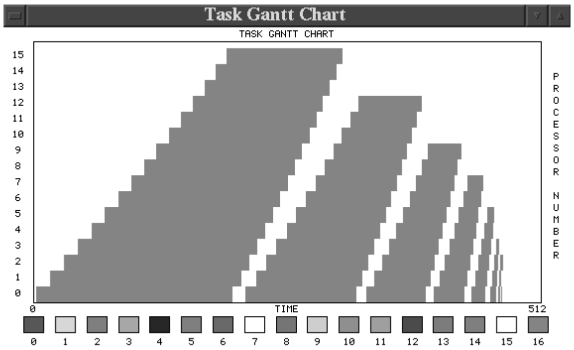


Fig. 4.9 – Diagramme de Gantt prédit pour une matrice pleine 4096×4096 sur 32 processeurs avec une distribution 1D.

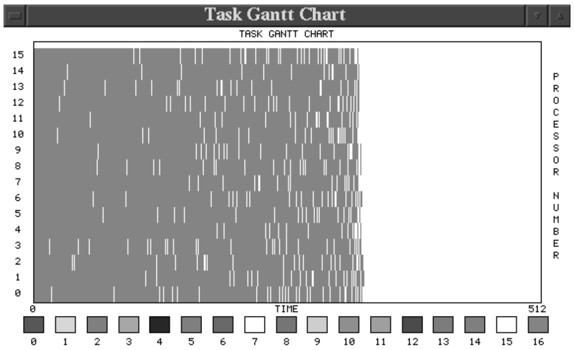


Fig. 4.10 – Diagramme de Gantt prédit pour une matrice pleine 4096×4096 sur 32 processeurs avec une distribution 2D.

Les deux figures 4.9 et 4.10 montrent les diagrammes de Gantt de la simulation dans le cas d'un système plein avec des distributions 1D et 2D. On voit que l'algorithme d'ordonnancement se comporte ici très bien ; dans ce cas, il met en évidence la supériorité de la distribution 2D. La figure 4.9 illustre aussi la tendance naturelle de l'algorithme de distribution à utiliser un nombre de processeurs, parmi l'ensemble des processeurs candidats, en correspondance avec la somme de travail disponible.

4.4 Validation expérimentale sur SP2

4.4.1 Conditions expérimentales

Tous les algorithmes décrits dans cette thèse ont été intrégrés dans la bibliothèque PaStiX [46,47,49,50], et nous avons utilisé la version 3.4 du logiciel Scotch [64,65] de placement statique et de renumérotation de matrice creuse.
La description de tous nos cas tests ainsi que des machines qui ont servies aux expérimentations parallèles ont été données dans les paragraphes 3.4 et 3.5. Les valeurs présentées dans le paragraphe 4.4.2 ont été obtenues sur le SP2 du CINES (qui n'est plus en service actuellement). A l'époque où ces résultats ont été obtenus, nous n'avions que la version $L.D.L^t$ du solveur parallèle.

La valeur moyenne en nombre de processeurs que nous avions trouvée sur le SP2 pour passer d'une distribution 1D à une distribution 2D était de 12 (voir paragraphe 4.1). La taille de blocage que nous utilisions était de 60 à la fois pour la distribution 1D et la distribution 2D.

4.4.2 Résultat de la factorisation parallèle

La table 4.3 présente les performances en temps de la factorisation parallèle avec notre stratégie de distribution/ordonnancement. Dans cette table 4.3, le symbol (-) sans résultat correspond à des cas où l'espace mémoire n'était pas suffisant pour que toutes les données distribuées de la matrice tiennent dans la mémoire vive sans provoquer de *swap* ce qui fausse bien sûr la mesure. A travers ces temps, nous cherchons à mesurer les meilleures performances possibles que l'on peut obtenir avec notre stratégie ; c'est pourquoi, dans le cadre d'une technique d'agrégation totale complétement prédite, nous allouons *de manière statique* en mémoire *tous les blocs d'agrégation AUBs* au début de l'exécution, ce qui sera un peu moins coûteux que de le faire au fur-et-à mesure au cours du calcul.

La puissance reportée (en Gigaflops) dans le tableau est la puissance qui exprime le rapport entre le nombre d'opérations que nécessite la factorisation séquentielle scalaire sans amalgamation et le temps de factorisation réel. Du fait de l'amalgamation (voir paragraphe 1.2), des termes de la matrice en blocs peuvent contenir des termes nuls ; cette puissance est donc inférieure à la puissance réelle exploitée sur les processeurs au cours de la factorisation, mais elle exprime bien les performances par rapport à une factorisation séquentielle scalaire de référence.

La table 4.3 montre que la *scalabilité* est plutôt bonne dans l'ensemble. Sur l'ensemble des cas, les mesures de performances varient entre 1.08 et 3.24 Gigaflops sur 16 processeurs, et entre 1.63 et 11.62 sur 64 processeurs, la meilleure scalabilité étant, comme on pouvait s'y attendre, obtenue par les plus "gros" cas (voir paragraphe 3.5 pour les caractéristiques). Pour ces problèmes assez gros, le niveau de Gigaflop atteint la moitié de la puissance maximum qui peut être réalisée effectivement lors d'un `GEMM` (primitives BLAS3 de multiplication de 2 matrices pleines) ; ceci est un résultat plus qu'honorable pour une factorisation en creux.

TAB. 4.3 – Performances de la factorisation parallèle sur IBM SP2 : le **temps** est en secondes et la puissance en Gigaflops.

Nom	Nombre de processeurs					
	2	4	8	16	32	64
GRID1023	-	-	**14.57** (1.43)	**8.61** (2.42)	**5.56** (3.74)	**3.83** (5.44)
CUBE39	**43.58** (0.51)	**24.20** (0.93)	**13.20** (1.70)	**8.88** (2.52)	**6.32** (3.55)	**5.03** (4.46)
CUBE47	-	**73.31** (0.95)	**38.23** (1.82)	**24.08** (2.89)	**17.15** (4.06)	**12.59** (5.53)
BCSSTK32	**4.03** (0.29)	**2.46** (0.47)	**1.41** (0.82)	**1.08** (1.08)	**0.78** (1.49)	**0.71** (1.63)
BBMAT	**28.17** (0.44)	**14.73** (0.85)	**8.55** (1.46)	**5.24** (2.39)	**3.39** (3.69)	**2.72** (4.60)
TOOTH	**20.01** (0.31)	**10.70** (0.59)	**6.60** (0.95)	**4.01** (1.56)	**2.77** (2.26)	**2.49** (2.52)
OCEAN	**48.37** (0.27)	**18.39** (0.71)	**9.78** (1.33)	**6.26** (2.08)	**4.12** (3.16)	**4.17** (3.12)
M14B	-	-	**46.95** (1.30)	**25.84** (2.37)	**14.60** (4.19)	**9.21** (6.64)
OILPAN	**7.28** (0.41)	**3.81** (0.78)	**2.23** (1.34)	**1.51** (1.98)	**1.03** (2.90)	**0.92** (3.24)
QUER	-	**4.46** (0.74)	**2.77** (1.18)	**1.90** (1.73)	**1.35** (2.43)	**1.04** (3.16)
INVEXTR1	**11.59** (0.32)	**6.31** (0.60)	**4.03** (0.93)	**2.52** (1.49)	**1.76** (2.14)	**1.57** (2.40)
SHIP001	**20.98** (0.43)	**10.91** (0.83)	**6.13** (1.47)	**3.78** (2.39)	**2.34** (3.86)	**1.96** (4.60)
X104	**31.53** (0.54)	**19.69** (0.87)	**11.02** (1.55)	**6.32** (2.71)	**5.30** (3.23)	**3.85** (4.44)
MT1	**37.92** (0.56)	**20.35** (1.04)	**12.91** (1.63)	**6.96** (3.03)	**4.33** (4.87)	**3.51** (6.01)
BMW3_2	-	**34.02** (0.88)	**20.78** (1.45)	**11.52** (2.61)	**7.90** (3.81)	**6.67** (4.51)
MIXTANK	**17.32** (0.42)	**10.02** (0.73)	**6.33** (1.16)	**3.80** (1.93)	**3.32** (2.20)	**3.10** (2.36)
BMWCRA_1	-	-	**30.97** (1.84)	**18.75** (3.04)	**9.55** (5.97)	**5.94** (9.60)
CRANKSG1	-	**33.54** (0.90)	**19.20** (1.57)	**10.54** (2.85)	**6.82** (4.41)	**5.34** (5.63)
SHIPSEC8	-	**43.40** (0.85)	**27.71** (1.33)	**15.40** (2.39)	**11.93** (3.09)	**9.23** (3.99)
CRANKSG2	-	**47.99** (0.96)	**25.33** (1.82)	**14.39** (3.20)	**8.45** (5.45)	**6.04** (7.62)
SHIPSEC5	-	**79.12** (0.88)	**40.36** (1.72)	**21.46** (3.24)	**15.36** (4.53)	**11.76** (5.91)
SHIP003	-	-	**45.83** (1.75)	**26.86** (2.98)	**16.69** (4.80)	**12.03** (6.66)
THREAD	**78.04** (0.50)	**41.14** (0.94)	**22.85** (1.70)	**13.45** (2.89)	**11.79** (3.29)	**6.70** (5.80)
COLOGB30	**9.88** (0.44)	**5.25** (0.83)	**3.23** (1.35)	**2.06** (2.12)	**1.69** (2.58)	**1.34** (3.25)
COLOGB75	-	**18.72** (0.82)	**8.99** (1.70)	**5.69** (2.69)	**3.89** (3.94)	**3.14** (4.88)
COUP1500T	-	-	-	-	-	**33.01** (11.51)
COUP2000T	-	-	-	-	-	**43.12** (11.62)

4.4.3 Comparaison avec PSPASES

Nous avons jugé intéressant de comparer les résultats en temps de notre factorisation guidée par les stratégies de distribution et d'ordonnancement qui ont été implémentée dans la bibliothèque PASTIX [46,49] , à celles de PSPASES [43,57] qui est basée sur une approche multifrontale (voir la description de PSPASES dans le chapitre 2). Nous avons utilisé pour ces comparaisons la version 1.0.3 de PSPASES qui utilise METIS version 4.0 pour l'ordonnancement. Dans les deux cas, la taille de blocage a été fixée à 60 et la bilbiothèque IBM ESSL est utilisée. Les expérimentations portent sur une collection de matrice au format RSA ; les valeurs de la table 4.4 donne les caractéristiques de nos cas tests en utilisant un algorithme de factorisation symbolique par colonne. Les résultats de la table 4.5 montrent les temps de factorisation obtenus par PaStiX et PSPASES. Lorsqu'il y a un symbole - , cela signifie que les temps que nous obtenions étaient faussés par l'utilisation du "swap mémoire" (cas ne passant pas en mémoire).

Une remarque destinée à faciliter la comparaison est qu'à l'époque de ces expérimentations, PSPASES utilisait une factorisation de Cholesky ($L.L^t$) qui permet une meilleur utilisation du câche dans les calculs que la factorisation $L.D.L^t$ qui était alors utilisée par PaStiX. Ainsi par exemple, pour une matrice pleine 1024×1024, la factorisation $L.L^t$ de ESSL prend 1.07s alors que la factorisation $L.D.L^t$ de la même bibliothèque ESSL prend 1.27s.

Les résultats obtenus sont assez favorables à PaStiX qui obtient de meilleurs temps de factorisation sur presque tous les cas jusqu'à 32 processeurs. Sur les plus gros cas, la comparaison continue à être favorable à PaStiX jusque 64 processeurs. Sur les cas plus petits, les résultats sont assez équivalents. Toutefois PSPASES semble avoir une meilleure scalabilité d'ensemble sur les cas les plus petits.

Une remarque importante concerne le fait que PSPASES semble avoir besoin de beaucoup plus de mémoire que PaStiX. En effet, sur le SP2 qui était limité à 256 Mo de mémoire par processeur nous avons beaucoup de cas où l'exécution de PSPASES échoue par manque d'espace mémoire ou conduit à des valeurs de temps faussée par du "swap mémoire". Par exemple, le cas THREAD passe normalement sur 2 nœuds avec PaStiX alors que PSPASES n'obtient une exécution normale qu'à partir de 16 processeurs. Des études sur les besoins mémoires cconcernant l'approche multifrontale et l'approche supernodale seront menées ultérieurement.

TAB. 4.4 – Description de cas tests. $\mathrm{NNZ_A}$ est le nombre de termes extra-diagonaux dans une partie triangulaire de la matrice A, $\mathrm{NNZ_L}$ est le nombre de termes extra-diagonaux dans la matrice factorisée L et OPC est le nombre d'opérations requises pour la factorisation.

Name	Columns	$\mathrm{NNZ_A}$	$\mathrm{NNZ_L}$(Scotch)	OPC(Scotch)	$\mathrm{NNZ_L}$(Metis)	OPC(Metis)
OILPAN	73752	1761718	8.912e+06	2.985e+09	9.065e+06	2.751e+09
QUER	59122	1403689	9.119e+06	3.281e+09	9.586e+06	3.448e+09
SMDOOR	162610	3873534	2.542e+07	1.531e+10	2.404e+07	1.237e+10
SHIP001	34920	2304655	1.428e+07	9.034e+09	1.481e+07	9.462e+09
X104	108384	5029620	2.634e+07	1.713e+10	2.728e+07	1.412e+10
MT1	97578	4827996	3.115e+07	2.109e+10	3.455e+07	2.269e+10
BMWCRA_1	148770	5247616	6.597e+07	5.702e+10	6.981e+07	6.124e+10
SHIPSEC5	179860	4966618	5.650e+07	6.952e+10	5.256e+07	5.509e+10
SHIP003	121728	3982153	5.873e+07	8.008e+10	5.910e+07	7.587e+10
THREAD	29736	2220156	2.404e+07	3.884e+10	2.430e+07	3.583e+10

Tab. 4.5 – Performances de la factorisation parallèle sur IBM SP2 : le **temps** est en secondes et la puissance en Gigaflops. Pour chaque matrice, la première ligne donne les temps de PaStiXet la seconde ceux de PSPASES.

Nom	Nombre de processeurs					
	2	4	8	16	32	64
OILPAN	**7.28** (0.41)	**3.81** (0.78)	**2.23** (1.34)	**1.51** (1.98)	**1.03** (2.90)	**0.92** (3.24)
	-	-	**2.79** (0.99)	**1.73** (1.59)	**1.25** (2.20)	**0.93** (2.96)
QUER	-	**4.46** (0.74)	**2.77** (1.18)	**1.90** (1.73)	**1.35** (2.43)	**1.04** (3.16)
	-	-	-	**2.01** (1.72)	**1.30** (2.65)	**0.96** (3.59)
SMDOOR	**29.54** (0.52)	**15.52** (0.99)	**9.10** (1.68)	**5.25** (2.91)	**4.07** (3.76)	**2.91** (5.25)
	-	-	-	**7.32** (1.69)	**3.91** (3.16)	**2.58** (4.79)
SHIP001	**20.98** (0.43)	**10.91** (0.83)	**6.13** (1.47)	**3.78** (2.39)	**2.34** (3.86)	**1.96** (4.60)
	-	-	**8.11** (1.17)	**4.48** (2.11)	**2.98** (3.17)	**2.14** (4.42)
X104	**31.53** (0.54)	**19.69** (0.87)	**11.02** (1.55)	**6.32** (2.71)	**5.30** (3.23)	**3.85** (4.44)
	-	-	-	-	**4.92** (2.87)	**3.18** (4.44)
MT1	**37.92** (0.56)	**20.35** (1.04)	**12.91** (1.63)	**6.96** (3.03)	**4.33** (4.87)	**3.51** (6.01)
	-	-	-	-	**5.70** (3.98)	**3.59** (6.32)
BMWCRA_1	-	-	**30.97** (1.84)	**18.75** (3.04)	**9.55** (5.97)	**5.94** (9.60)
	-	-	-	-	**24.87** (2.49)	**13.48** (4.61)
SHIPSEC5	-	**79.12** (0.88)	**40.36** (1.72)	**21.46** (3.24)	**15.36** (4.53)	**11.76** (5.91)
	-	-	-	-	-	**11.81** (4.66)
SHIP003	-	-	**45.83** (1.75)	**26.86** (2.98)	**16.69** (4.80)	**12.03** (6.66)
	-	-	-	-	-	**14.08** (5.39)
THREAD	**78.04** (0.50)	**41.14** (0.94)	**22.85** (1.70)	**13.45** (2.89)	**11.79** (3.29)	**6.70** (5.80)
	-	-	-	-	**11.24** (3.19)	**6.41** (5.59)

4.5 Adaptation aux architectures de type SMP

L'adaptation de nos algorithmes à une architecture SMP tient essentiellement dans la modélisation des communications. La première modification par rapport à une architecture classique à une seule hiérarchie mémoire consiste à prendre en compte, pour le calcul du coût d'une communication, le fait que les processeurs appartiennent ou non au même nœud SMP.

Comme nous l'avons vu dans le paragraphe 3.4 pour le cas du SP3, un autre phénomène qui peut intervenir dans une architecture à base de nœuds SMP est que le débit et la latence varient selon le nombre de processeurs communiquant en même temps dans un nœud SMP. Notre algorithme de distribution ne nous permet pas de connaître le nombre de communications simultanées à un moment donné de la factorisation. Nous avons donc introduit un modèle de communication qui utilise des valeurs de latence et de bande passante qui varient selon le nombre de processeurs candidats qui participent au calcul d'un super-nœud de l'arbre d'élimination. Ainsi pour un super-nœud de l'arbre d'élimination, on considère que si m processeurs candidats participent alors il y a, en moyenne, $m/2$ communications simultanées.

4.5.1 Validation expérimentale du modèle SMP

Nous avons tout d'abord validé l'intérêt d'une modélisation dédiée aux machines à nœuds SMP communiquant via MPI.

TAB. 4.6 – Comparaison des temps de factorisation obtenu sur IBM SP3 en utilisant ou non le modèle SMP. Les temps sont en secondes : temps sans modèle SMP (temps avec modèle SMP).

Nom	Nombre de processeurs			
	16	32	64	128
OILPAN	.48 (.44)	.38 (.37)	.33 (.32)	.29 (.29)
BMWCRA1	5.09 (4.84)	2.93 (2.90)	1.91 (1.83)	1.55 (1.48)
CUBE47	7.24 (7.08)	5.13 (4.94)	4.09 (3.91)	3.38 (3.34)
SHIP003	8.08 (7.20)	5.01 (4.82)	3.48 (3.49)	2.88 (2.84)
COUP1500T	29.50 (28.71)	16.15 (15.88)	8.06 (7.94)	4.99 (4.92)
COUP2000T	42.78 (40.52)	20.21 (19.67)	10.31 (9.93)	6.20 (5.80)
COUP3000T	-	31.26 (29.48)	17.43 (16.36)	8.68 (8.67)

Les résultats de ces quelques tests de tailles et de natures variées (voir paragraphe 3.5) montrent que le gain obtenu par le modèle SMP que nous avons défini atteint ici au plus une différence de 10% sur les plus grosses matrices, et la tendance est que plus il y a de processeurs, moins l'écart entre les temps d'une distribution utilisant une modélisation avec ou sans SMP est grand. Ce résultat qui peut paraître décevant s'explique naturellement par le fait que, hormis les problèmes classiques de grain de calcul trop faible à partir d'un certain nombre de processeurs, le nombre (16) de processeurs par nœud SMP est ici assez faible et qu'il y a rapidement "prédominance" du modèle de type réseau sur celui de type mémoire partagée quand le nombre total de processeurs augmente. Il sera intéressant de remesurer ces écarts sur la machine de l'IDRIS à base de Power4 qui possède des nœuds SMP de 32 processeurs.

4.6 Validation expérimentale sur SP3

4.6.1 Conditions expérimentales

Les résultats que nous présentons dans le paragraphe suivant on été obtenus par PaStiX pour une factorisation $L.L^t$ sur l'IBM SP3 du CINES. La renumérotation utilisée est, comme pour les expérimentations sur le SP2, celle fournit par Scotch.

La distribution utilise le modèle de communication pour le SMP que nous avons décrit dans le paragraphe 4.5. Le nombre de processeurs candidats à partir duquel "on bascule" d'une distribution 1D à une distribution 2D est de 12, et le facteur de blocage est de 60 en distribution 1D et de 120 en distribution 2D (voir paragraphe 4.1).

Les expérimentations ne commencent qu'à partir de 16 processeurs (1 nœud SMP de la machine), car en dessous de cette limite la configuration de la machine du CINES autorise d'autres applications à partager le nœud ce qui a tendance à fausser les mesures. Comme pour les temps sur IBM SP2, nous utilisons une *allocation statique* des AUBs pour obtenir les meilleurs temps possibles.

4.6.2 Résultat de la factorisation parallèle

TAB. 4.7 – Performances de la factorisation sur IBM SP3 : le **temps** est en secondes et la puissance en Gigaflops.

Name	Number of processors			
	16	32	64	128
GRID1023	**2.59** (8.03)	**1.72** (12.11)	**1.18** (17.58)	**.94** (21.98)
CUBE39	**2.59** (8.62)	**1.87** (11.91)	**1.58** (14.15)	**1.58** (14.11)
CUBE47	**7.08** (9.83)	**4.94** (14.07)	**3.91** (17.80)	**3.34** (20.84)
BCSSTK32	**.30** (3.76)	**.25** (4.54)	**.24** (4.71)	**.25** (4.58)
TOOTH	**1.23** (5.06)	**.87** (7.18)	**.84** (7.45)	**.82** (7.59)
OCEAN	**2.41** (5.38)	**1.41** (9.19)	**1.04** (12.39)	**.97** (13.37)
M14B	**7.00** (8.72)	**4.25** (14.37)	**3.03** (20.15)	**2.70** (22.57)
OILPAN	**.44** (6.66)	**.37** (8.05)	**.32** (9.28)	**.29** (10.29)
QUER	**.52** (6.27)	**.42** (7.76)	**.39** (8.21)	**.37** (8.66)
INVEXT	**.76** (4.89)	**.54** (6.88)	**.52** (7.18)	**.56** (6.61)
SHIP001	**1.06** (8.50)	**.70** (12.74)	**.59** (15.28)	**.55** (16.19)
X104	**1.83** (9.35)	**1.38** (12.33)	**1.16** (14.72)	**1.22** (14.03)
MT1	**1.91** (11.03)	**1.24** (16.96)	**1.04** (20.14)	**.94** (22.42)
BMW3	**3.30** (9.11)	**2.23** (13.47)	**1.69** (17.70)	**1.46** (20.48)
MIXTANK	**1.20** (6.07)	**1.14** (6.38)	**1.12** (6.49)	**1.13** (6.42)
BMWCRA1	**4.84** (11.76)	**2.90** (19.29)	**1.83** (29.72)	**1.48** (37.31)
CRANKSEG1	**3.08** (9.73)	**1.91** (15.70)	**1.58** (19.02)	**1.29** (23.17)
SHIPSEC8	**4.56** (8.07)	**3.64** (10.10)	**2.73** (13.48)	**2.50** (14.71)
CRANKSEG2	**3.97** (11.58)	**2.45** (18.73)	**1.93** (23.78)	**1.59** (28.93)
SHIPSEC5	**6.26** (11.09)	**4.43** (15.68)	**3.33** (20.82)	**3.03** (22.87)
SHIP003	**7.20** (10.28)	**4.82** (16.59)	**3.49** (22.89)	**2.84** (27.78)
THREAD	**4.16** (9.33)	**3.12** (12.41)	**2.77** (13.98)	**2.40** (16.14)
COL30	**.57** (7.54)	**.45** (9.50)	**.39** (10.92)	**.38** (11.24)
COL75	**1.72** (8.88)	**1.11** (13.73)	**.87** (17.50)	**.86** (17.79)
COUP1500T	**28.71** (13.22)	**15.88** (23.91)	**7.94** (47.81)	**4.92** (77.15)
COUP2000T	**40.52** (12.36)	**19.67** (25.46)	**9.93** (50.43)	**5.80** (86.32)
COUP3000T	-	**29.48** (26.06)	**16.36** (46.96)	**8.67** (88.59)
COUP5000T	-	-	**26.49** (49.50)	**14.15** (92.57)
COUP8000T	-	-	**42.65** (49.00)	**22.49** (92.93)

On peut voir sur le tableau 4.7 que l'on a une scalabilité d'ensemble raisonnable, mais que d'une manière plus marquée par rapport aux résultats SP2, seuls les très gros cas (COUPOLE) obtiennent des temps montrant une scalabilité vraiment satisfaisante jusqu'à 128 processeurs. Ceci est dû au fait que les cas ne sont plus "assez gros" pour ce type de machine qui est moins équilibrée en pratique que le SP2 du point de vue du ratio calcul/communication. En effet, plus le nombre de nœuds SMP est important, plus le nombre de super-nœuds impliquant des communications extra-nœuds et des communications simultanées est important (voir figure 4.11) ; or comme nous l'avons vu dans le paragraphe 3.4, les performances en communication peuvent alors s'affecter très vite dans ces conditions.

Par contre il est intéressant de noter que pour la plupart des cas, on arrive à retrouver un rapport autour de 3 entre les temps sur SP3 et SP2. Ceci est assez

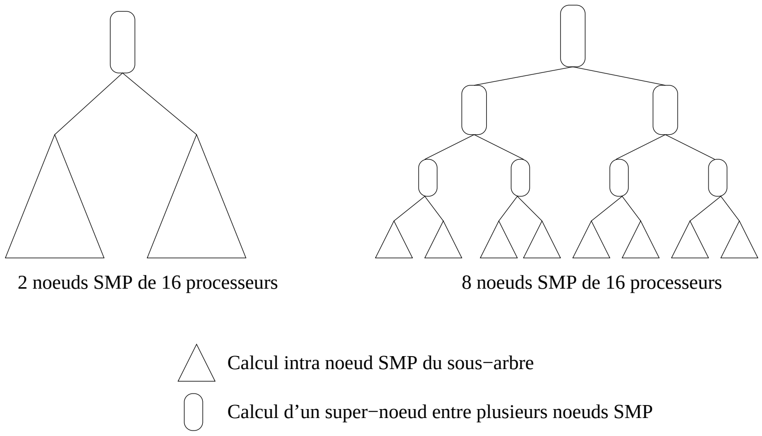


FIG. 4.11 – Proportion de calcul intra et extra-nœud quand le nombre de processeurs augmente.

proche du rapport entre la puissance de crête des processeurs de ces deux machines (480 MFlops/1,5 GFlops), ce qui montre bien le bon comportement global de nos algorithmes de distribution/ordonnancement. Dans certains cas, ce rapport peut être supérieur à 3 ceci étant dû à ce que l'algorithme de notre solveur en version $L.L^t$ utilise mieux les effets caches qu'en $L.D.L^t$.

4.7 Généralité de l'approche

L'approche et les algorithmes que nous avons présentés dans ce chapitre peuvent se généraliser à d'autres cas que la factorisation Cholesky-Crout sur machine à mémoire distribuée et à base de nœuds SMP.

La première généralisation possible concerne le type de factorisation. En effet les techniques que nous avons présentées peuvent très bien s'adapter pour les factorisations dont le coût ne dépend pas dynamiquement de l'exécution. Si on a besoins de faire du pivotage dynamique, comme c'est généralement le cas pour la factorisation de type $L.U$, alors le coût de ce pivotage ne pourra pas être connu lors de la phase de prétraitement. Toutefois dans le cas de $L.U$, la méthode superLU [61], qui effectue un pivotage statique basé sur les valeurs numériques initiales de la matrice, peut régler au moins partiellement ce problème du pivotage dynamique. Les adaptations à cet autre type de factorisation concernerait essentiellement la définition des tâches de calcul.

La seconde généralisation de notre méthode de distribution concerne le type d'architecture cible. Nous n'avons testé nos méthodes que sur des machines homogènes (IBM SP2, SP3) avec un ou plusieurs niveaux de communication. Il est tout à fait envisageable d'adapter notre modélisation à une machine hétérogène type grappe de PCs par exemple. Dans ce cas, outre une modélisation appropriée du réseau et des performances de calcul de chaque machine, il faudrait apporter quelques modifications aux algorithmes de repartitionnement et de distribution :

- dans la phase de repartitionnement : la répartition des processeurs candidats pourrait se faire selon une méthode de type "sac à dos" [33] pour trouver le meilleur équilibrage des processeurs candidats sur les sous-arbres en fonction du coût des sous-arbres et de la puissance des processeurs ;
- dans la phase de distribution, il faudrait tenir compte du fait que le coût d'une tâche de calcul diffère selon le processeur lors de la simulation.

4.8 Coût de nos algorithmes

Notre méthode de distribution et d'ordonnancement des calculs est plus coûteuse en temps qu'une distribution classique de type *proportionnal mapping* par exemple. La phase asymptotiquement la plus coûteuse est la phase de distribution/ordonnancement (voir figure 4.4) : dans l'algorithme de cette phase, c'est l'étape 2 qui comporte deux boucles imbriquées qui est la plus coûteuse.

Si on note :

- P le nombre de processeurs ;
- n le nombre total de tâches ;
- d_i le nombre de tâches qui deviennent prêtes après l'exécution de la tâche t_i ;
- d le degré maximal d'une tâche dans le graphe des tâches,

alors le nombre d'opérations (on majore le nombre de processeurs candidats par P) est asymptotiquement en :

$$\Theta(\sum_{i=1}^{n}(d_i.P.d)).$$

Comme $\sum_{i=1}^{n} d_i = n$, la complexité asymptotique de la phase de distribution et d'ordonnancement est

$$\Theta(n.P.d)).$$

Cette formule est confirmée (linéarité par rapport à P) à partir de 32 processeurs par le tableau 4.8 qui présente le temps de distribution d'une matrice pleine 6000×6000 en distribution 2D (il y a ici 171700 tâches de calcul quel que soit le nombre de processeurs) pour différent nombre de processeurs (sur une station de travail avec un processeur PowerPC à 350 MHz) :

Tab. 4.8 – Temps de distribution d'une matrice pleine 6000×6000 en 2D

matrice pleine	Nombre de processeurs							
	2	4	8	16	32	64	128	256
6000×6000	5.36	5.76	6.72	8.71	12.54	20.2	44.0	90.19

Le tableau 4.9 donne maintenant le total des temps de repartitionnement et de distribution pour nos cas tests. Ces temps ont été obtenus en faisant tourner ces algorithmes sur un nœud SP2 :

Tab. 4.9 – Temps séquentiels pour la distribution et pour la factorisation symbolique (FS_Time).

Nom	Nombre de processeurs						
	2	4	8	16	32	64	FS_Time
GRID1023	1.15	1.12	1.77	2.20	3.27	10.27	3.31
CUBE39	0.28	0.38	0.47	0.80	1.63	3.11	0.58
CUBE47	0.94	1.17	1.39	2.70	4.72	8.92	1.35
BCSSTK32	0.13	0.14	0.16	0.19	0.28	0.52	0.55
TOOTH	1.82	1.94	2.16	2.50	3.41	7.22	0.96
OCEAN	3.89	4.11	4.24	4.86	6.43	12.55	1.29
M14B	5.89	6.73	7.52	8.42	10.06	16.32	2.43
OILPAN	0.14	0.14	0.14	0.19	0.31	0.92	0.88
QUER	0.10	0.10	0.10	0.14	0.29	0.62	0.69
INVEXTR1	0.87	0.96	1.17	1.39	1.69	2.40	0.62
SHIP001	0.05	0.05	0.08	0.11	0.19	0.49	0.88
X104	0.13	0.19	0.22	0.31	0.78	2.15	1.95
MT1	0.08	0.12	0.18	0.26	0.84	1.46	1.80
BMW3_2	0.57	0.73	0.83	1.00	1.80	5.13	3.12
MIXTANK	0.57	0.85	0.97	1.16	1.70	3.72	0.61
BMWCRA_1	0.33	0.39	0.58	0.87	1.08	1.56	2.59
CRANKSG1	0.09	0.13	0.26	0.41	0.89	2.02	1.99
SHIPSEC8	0.51	0.75	1.02	1.43	2.85	7.88	1.62
CRANKSG2	0.11	0.17	0.31	0.52	0.71	1.91	2.55
SHIPSEC5	0.88	0.94	1.38	1.76	3.28	8.00	2.43
SHIP003	0.70	0.95	1.32	1.85	3.36	8.53	2.05
THREAD	0.11	0.51	0.56	0.71	2.35	6.20	0.87
COLOGB30	0.03	0.05	0.06	0.80	0.17	0.33	0.10
COLOGB75	0.08	0.11	0.14	0.20	0.29	0.88	0.24
COUP1500T	1.46	1.55	1.69	1.99	2.65	5.19	1.17
COUP2000T	2.10	2.13	2.27	2.70	3.57	6.77	2.05

L'important dans ces tableaux est de constater que les temps de nos algorithmes de distribution/ordonnancement sont très raisonnables en particulier si on les compare avec ceux de la factorisation symbolique. Le surcoût occasionné par nos algorithmes peut être amorti dans beaucoup d'applications industrielles qui utilisent la même structure de matrice (qui peut ne dépendre par exemple que de la topologie d'un maillage d'éléments finis) pour résoudre un grand nombre de fois des systèmes où seuls les cœfficients numériques changent. Un exemple d'une telle application où a été appliqué PaStiX est décrit dans [38, 40].

Chapitre 5

Optimisation de la mémoire

Dans les chapitres précédents, nous avons présenté nos stratégies de distribution et d'ordonnancement, et nous avons montré qu'elles permettaient d'obtenir de bonnes performances en temps. Outre les problèmes de performances en temps, la faisabilité effective de la résolution des systèmes de grandes tailles est déterminante pour les applications industrielles. Un des points forts de notre méthode est que l'ordonnancement des tâches de calcul nous permet de connaître exactement l'ordre dans lequel les données seront accédées ou allouées en mémoire. Comme nous allons le voir dans ce chapitre, cette connaissance *a priori* de l'évolution de la mémoire nous permet de mettre en place des mécanismes optimisés pour la gestion de cette mémoire.

5.1 Réduction du surcoût mémoire dû à l'agrégation totale des blocs de contributions

Un problème important qui intervient dans les applications industrielles de grandes tailles est celui de la complexité en espace mémoire, en particulier à cause du surcoût mémoire dû aux structures supplémentaires qui interviennent dans la gestion des données distribuées et des communications. Dans le cas de la factorisation supernodale $L.L^t$ ou $L.D.L^t$ avec agrégation totale des contributions, ce surcoût consiste essentiellement dans l'allocation des blocs agrégés de contributions. En effet, un bloc agrégé AUB_{ij} sur un processeur (voir paragraphe 3.3) consiste en un bloc recouvrant de toutes les contributions de ce processeur pour le bloc (i, j) appartenant à un autre processeur. Ainsi un AUB doit être alloué dès le calcul de sa première contribution, et reste en mémoire jusqu'à ce que sa dernière contribution lui ait été ajoutée et qu'il soit effectivement envoyé. La somme en mémoire de tous les AUBs alloués simultanément peut devenir très importante par rapport à la mémoire nécessaire pour stocker les données locales de la matrice distribuée et cette somme dépend naturellement du nombre total de processeurs. La table 5.1 montre le pourcentage de mémoire nécessaire pour le stockage des AUBs au cours du calcul par rapport à la mémoire nécessaire pour le stockage des cœfficients de la matrice. Comme on peut le voir ce pourcentage est très important, surtout pour les cas issus de maillage en 3D (CUBE47, SHIP003, THREAD), et a tendance à augmenter quand le nombre de processeurs augmente.

TAB. 5.1 – Pourcentage de mémoire nécessaire pour allouer les AUBs par rapport à celle nécessaire pour les cœfficients de la matrice.

	Nombre de processeurs			
Nom	16	32	64	128
THREAD	**264.0**	**325.5**	**366.0**	**172.2**
SHIP003	**95.0**	**129.5**	**205.9**	**330.9**
CUBE47	**103.7**	**233.5**	**332.8**	**344.9**
BMWCRA1	**9.1**	**23.2**	**41.4**	**55.0**

Une solution pour réduire la taille de cette mémoire supplémentaire consiste à réduire le nombre d'AUBs alloués simultanément en *envoyant certains d'entre eux avant qu'ils ne soient totalement agrégés*. Ainsi un AUB partiellement agrégé pourra être envoyé et libérera de la mémoire jusqu'à ce qu'il soit réalloué lors du calcul de sa prochaine contribution. Cette méthode est appelée *agrégation locale partielle*.

La technique d'agrégation partielle est nécessairement pénalisante en temps par rapport à l'agrégation totale : elle provoque plus d'envois de messages, de réallocations dynamiques et oblige les processeurs à recevoir et ajouter en plusieurs fois les AUBs destinés à leurs blocs locaux. Il convient donc de minimiser le nombre d'AUBs partiels, selon la contrainte mémoire, pour que le temps de factorisation parallèle ne soit pas trop augmenté. Nous proposons une solution qui, *avant la factorisation parallèle*, et en fonction du vecteur d'ordonnancement des tâches de calcul et de l'ordre d'envoi des AUBs (selon le schéma d'agrégation totale), modifie pour chaque processeur le schéma d'envoi de ses AUBs afin d'anticiper l'envoi de certains d'entre eux (le plus petit nombre possible) partiellement agrégés.

Grâce à l'ordonnancement des tâches de calcul calculé dans la phase de distribution et d'ordonnancement, il est très aisé de connaître l'ordre des ajoûts de contributions dans les AUBs. De cette façon, et comme l'illustre la figure 5.1, on peut déduire la taille exacte de mémoire déjà allouée au fur-et-à-mesure des calculs de contributions. Cette figure montre comment les allocations et désallocations des AUBs peuvent être suivies à l'aide de l'ordre d'exécution des tâches `BMOD` en distribution 2D et `COMP_1D` en distribution 1D (voir paragraphe 3.3). En effet, lorsque

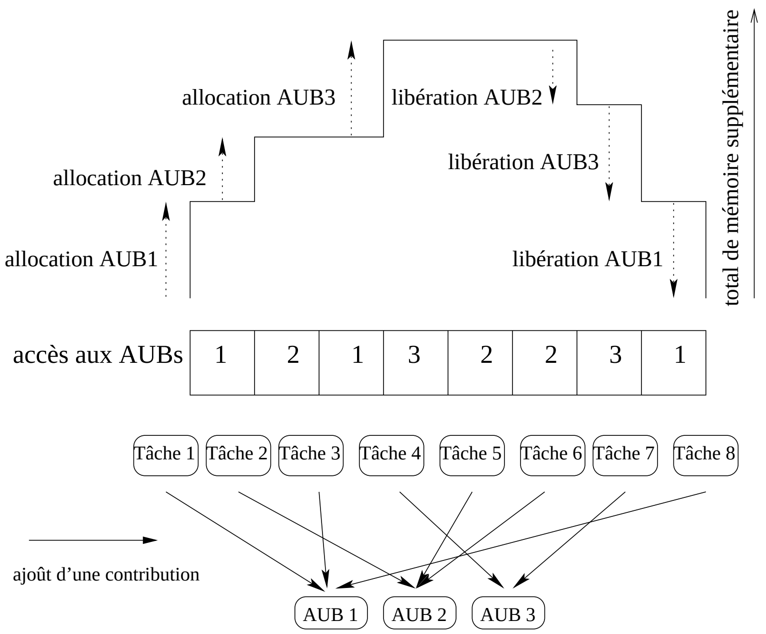


FIG. 5.1 – Calcul du surcoût mémoire pour les AUBs à partir de l'ordonnancement des tâches.

une tâche `BMOD` (ou `COMP_1D`) ajoute une contribution pour la première fois dans un AUB, cela signifie qu'il doit être alloué en mémoire et, lorsque sa dernière contribution est ajoutée et qu'il est envoyé, cela signifie alors qu'il doit libéré cette mémoire. Par exemple, sur la figure 5.1, on peut voir que le pic de mémoire allouée pour les AUBs est atteint quand la tâche 4 a besoin d'allouer l'AUB 3. Ainsi, étant donnée une contrainte en taille mémoire, il est possible à partir de l'ordonnancement des tâches de calcul et des envois des AUBs, de savoir si cette contrainte sera respectée ou non ; on peut identifier alors quelle allocation d'AUB provoquera éventuellement un dépassement pour cette contrainte. L'algorithme que nous utilisons pour détecter les dépassements mémoires que peuvent causer l'allocation d'un AUB, et qui permet alors de modifier le schéma de communication pour anticiper l'envoi des AUBs partiellement agrégés est donné dans la figure 5.2.

Pour décrire cet algorithme, nous utiliserons les notations suivantes :
- r : nombre total d'ajoûts dans les AUBs au cours de la factorisation sur un processeur (par exemple sur la figure 5.1 r vaut 8) ;
- ajoût_AUB[i] : indice de l'AUB qui reçoit le $i^{\text{ème}}$ ajoût de contribution ;
- $\mathcal{A}$: l'ensemble des AUBs présents en mémoire ;
- taille(b) : taille en octets de l'AUB b ;
- mémoire_occupée : taille de la mémoire occupée par des AUBs ;

- limite_mémoire : limite de taille mémoire que l'on ne veut pas dépasser ;
- agrégation_finie(b) : renvoie *vraie* si toutes les contributions de l'AUB b lui ont été ajoutées ;
- processeur_destinataire(b) : processeur auquel est destiné l'AUB b ;
- S_q : ensemble des AUBs dont le processeur destinataire est le processeur q (voir paragraphe 3.3) ;
- $\mathcal{P}(b)$: propriété qui est vraie lorsque tous les AUBs destinés à processeur_destinataire(b) et de priorité inférieure à celle de b ont été envoyés (voir paragraphe 3.3 et 4.3) ;
- premier(S_q) : le premier AUB non totalement agrégé dans la file d'émission vers le processeur q.

```
1.  𝒜 ← ∅ ;
2.  mémoire_occupée ← 0 ;
3.  Pour i = 1 à r faire
4.    b ← ajoût_AUB[i] ;
5.    Si (b ∉ 𝒜) alors
6.      𝒜 ← 𝒜 ∪ {b} ;
7.      mémoire_occupée ← mémoire_occupée + taille(b)
8.      Tant que mémoire_occupée > limite_mémoire faire
9.        Soit c ∈ 𝒜, tel que c ≠ b et c = l'AUB
          de 𝒜 qui sera réaccédé le plus tard ;
10.       𝒜 ← 𝒜 \ {c} ;
11.       q ← processeur_destinataire(c) ;
12.       c devient un AUB partiel ;
        On choisit la priorité minimimale qui assure que l'AUB c aura été émis
          avant que l'AUB b ne soit accédé
13.       priorité[c] ← priorité[premier(S_q)] − 1 ;
14.       mémoire_occupée ← mémoire_occupée−taille(c) ;
15.   Si (agrégation_finie(b)= vraie) alors
16.     Pour tous les AUBs c ∈ S_q tels que 𝒫(c) est vraie faire
17.       mémoire_occupée ← mémoire_occupée−taille(c) ;
18.       𝒜 ← 𝒜 \ {c} ;
```

FIG. 5.2 – Algorithme de création des AUBs partiels pour une émission anticipée en fonction d'une limite mémoire.

En additionnant au fur-et-à-mesure des ajoûts de contributions pour des AUBs qui seront alloués lors de la factorisation parallèle, on peut trouver un AUB b qui provoque un dépassement de la limite mémoire (ligne 8 de l'algorithme) ; nous choisissons alors d'envoyer suffisament d'AUBs partiellement agrégés parmi ceux déjà alloués (lignes 9 à 14 de l'algorithme) pour que la limite mémoire soit respectée. Un choix judicieux de ces AUBs partiellement agrégés consiste à prendre ceux pour lesquels une nouvelle contribution sera calculée le plus tard possible (ligne 9). En effet, en choisissant ceux qui seront réaccédés le plus tard, on repousse au plus loin le moment où l'on sera obligé de les réallouer et ainsi on préserve le plus longtemps possible la place mémoire que leur désallocation a permis d'économiser. Ce critère simple permet de minimiser le nombre d'AUBs à utiliser, et donc de dégrader le moins possible le schéma de communication originel. Par exemple dans la figure 5.1, si l'allocation de l'AUB 3 provoque un dépassement de la limite mémoire, alors un bon choix d'AUB partiel est l'AUB 1 qui n'est réaccédé qu'à la fin du calcul. La priorité d'un AUB que nous avons définie dans le paragraphe 4.3 correspond à son rang dans la file d'envoi vers son processeur destinataire. Un AUB ne peut être envoyé à son destinataire que lorsqu'il est totalement agrégé et que tous les AUBs ayant une priorité plus basse dans sa file ont été envoyés (i.e lorsque $\mathcal{P}$ est *vraie*). Lorsque l'on décide qu'un AUB partiellement agrégé doit être envoyé, son envoi doit être fait suffisament tôt pour que la mémoire qu'il occupe soit libérée. C'est pourquoi on lui donne la priorité *minimale* qui assure qu'il sera déjà envoyé au moment où l'AUB b qui provoque un dépassement mémoire sera alloué (ligne 13).

La figure 5.3 présente pour le cas test BMWCRA1 et par processeur, l'évolution de la mémoire allouée pour les AUBs au cours d'une factorisation sur 32 processeurs ; on utilise différentes contraintes de réduction mémoire (par exemple 50% signifie que la mémoire nécessaire aux AUBs a été réduite de 50 % par rapport à une agrégation totale). On peut voir sur cette figure qu'en agrégation totale (courbe 0% de réduction), le pic du surcoût mémoire est atteint ($\approx$ 5.5 Mo) vers le 3500$^{\text{ème}}$ ajoût d'une contribution dans un AUB. Les courbes 25 %, 50 %. montrent le comportement de l'évolution de la mémoire obtenu lorsqu'on applique l'algorithme que nous avons décrit. On peut remarquer que cet algorithme permet de réduire la mémoire au plus juste pour respecter la limite mémoire imposée.

La table 5.2 et la figure 5.4 présentent pour 4 cas tests différents la pénalité en temps pour différents niveaux de réduction de la mémoire allouée aux AUBs. La table 5.2 montre l'intérêt de l'agrégation totale ; lorsque l'agrégation partielle est poussée jusqu'à un comportement proche du "fan-out" alors la dégradation en temps devient très importante. Ainsi pour la pluspart des cas, lorsque l'on contraint les AUBs à une limite mémoire réduite de plus de 80% par rapport à l'agrégation totale, on observe une dégradation du temps de la factorisation supérieure à 100 % du temps obtenu en agrégation totale. Par contre, comme on peut le voir sur ces exemples, une réduction de 50 % du surcoût mémoire entraîne un surcoût en temps de factorisation qui varie sur 128 processeurs entre 2,69% et 28,4 % par

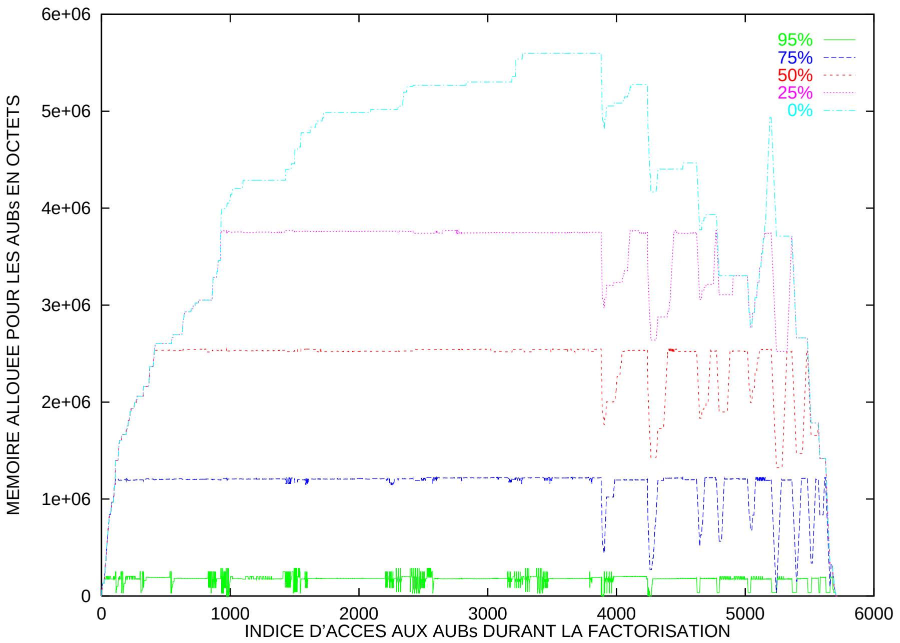


FIG. 5.3 – Courbe de réduction mémoire par agrégation partielle pour la BMWCRA1 sur 32 processeurs.

rapport au temps de factorisation en agrégation totale (ligne 0 %) ; cette limite apparait donc comme un bon compromis entre réduction du surcoût mémoire et augmentation du temps de factorisation. Dans ce contexte, notre régulation de la mémoire supplémentaire conduit globalement à une bonne scalabilité mémoire.

THREAD	Nombre de processeurs			
Réduction	16	32	64	128
0 %	**4.16**	**3.12**	**2.77**	**2.40**
25 %	**4.80**	**3.36**	**3.00**	**2.53**
50 %	**6.05**	**3.66**	**3.14**	**2.63**
75 %	**7.56**	**5.16**	**3.74**	**2.98**
95 %	**9.75**	**8.39**	**5.78**	**4.40**

SHIP003	Nombre de processeurs			
Réduction	16	32	64	128
0 %	**7.20**	**4.82**	**3.49**	**2.84**
25 %	**7.73**	**5.03**	**3.68**	**2.87**
50 %	**7.93**	**5.09**	**3.77**	**2.93**
75 %	**10.21**	**6.74**	**4.90**	**3.51**
95 %	**18.96**	**13.52**	**9.94**	**8.05**

BMWCRA1	Nombre de processeurs			
Réduction	16	32	64	128
0 %	**4.84**	**2.90**	**1.83**	**1.48**
25 %	**5.06**	**3.11**	**2.07**	**1.69**
50 %	**5.47**	**3.64**	**2.25**	**1.90**
75 %	**7.36**	**4.76**	**3.18**	**2.52**
95 %	**11.85**	**6.70**	**4.34**	**3.51**

CUBE47	Nombre de processeurs			
Réduction	16	32	64	128
0 %	**7.08**	**4.94**	**3.91**	**3.34**
25 %	**7.47**	**5.27**	**4.05**	**3.38**
50 %	**8.63**	**6.65**	**4.11**	**3.43**
75 %	**14.09**	**11.64**	**5.54**	**4.23**
95 %	**24.21**	**18.28**	**9.35**	**8.97**

Tab. 5.2 – Influence de la réduction mémoire par agrégation partielle sur le temps de factorisation parallèle sur IBM SP3.

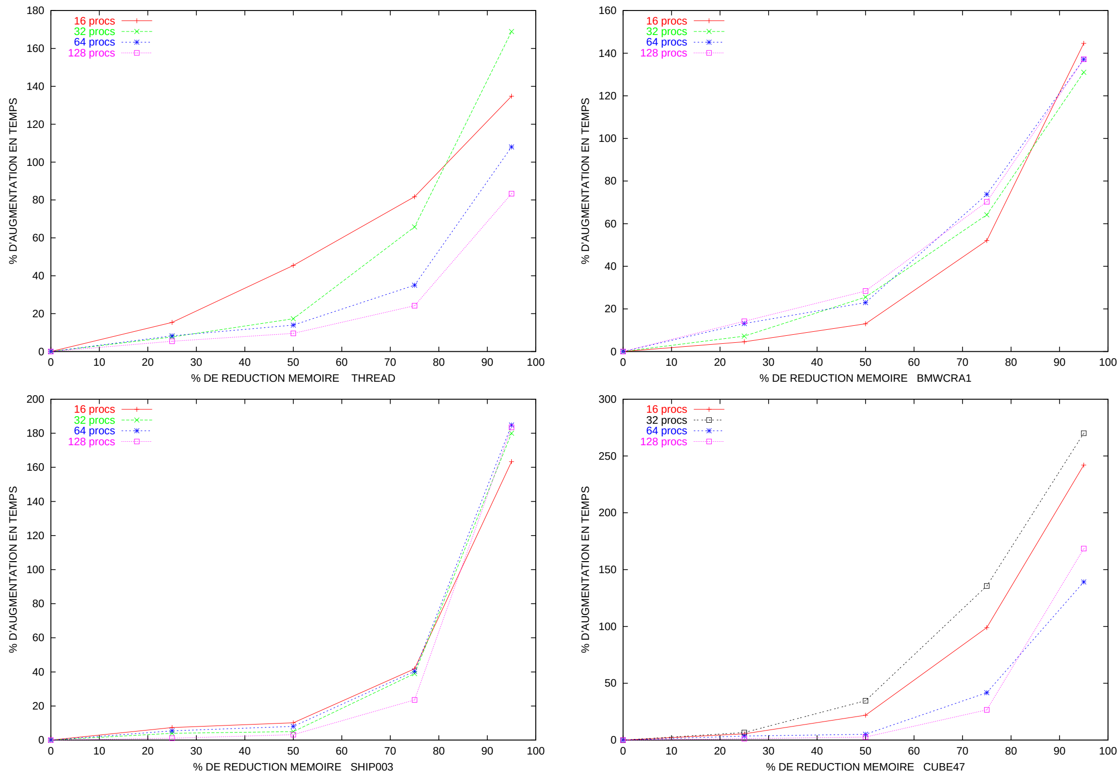


Fig. 5.4 – % d'augmentation du temps de factorisation / % de réduction mémoire

5.2 Etude préliminaire concernant une version Out-of-Core

Dans le paragraphe précédent, nous avons présenté une technique pour contrôler le surcoût mémoire induit par notre algorithme parallèle de factorisation avec agrégation des contributions. Le but était de contrôler la scalabilité mémoire via un mécanisme optimisé d'agrégation partielle en visant des temps de calcul du même ordre que pour une factorisation avec agrégation totale des contributions. Cependant, les grands challenges numériques actuels concernent des résolutions de systèmes qui comportent des dizaines de millions d'inconnues. La mémoire physique des machines n'est alors toujours pas suffisante pour des systèmes d'une telle grandeur. Pour pouvoir les traiter, il faut nécessairement mettre en place un mécanisme "Out-of-Core".

Le mécanisme "Out-of-Core" consiste à n'avoir en mémoire vive que les données nécessaires au traitement des calculs courants. Lorsqu'un calcul est terminé, les données qui lui sont associées sont écrites sur disque libérant ainsi de la mémoire pour recharger des données qui serviront aux calculs ultérieurs. Le mécanisme le plus élémentaire "Out-of-Core" consiste à laisser "swaper" le système d'exploitation. Mais généralement cette technique de swap donne des résultats catastrophiques, car elle utilise un découpage uniforme de la mémoire en pages et un algorithme universel (basé par exemple sur un critère *Less Recently Used* [79]) pour choisir les pages à remettre sur disque.

Pour que le mécanisme "Out-of-Core" soit efficace, il faut qu'il soit adapté autant que faire se peut à l'application. La technique "Out-of-Core" s'adapte naturellement à notre algorithmique : en effet, comme dans le cas de l'agrégation partielle, tous les accès aux données peuvent naturellement être anticipés par l'ordonnancement des tâches de calcul. Dans notre cas, le temps de lecture d'un bloc sera généralement plus coûteux que le temps de calcul associé à ce bloc. La performance de la factorisation "Out-of-Core" dépendra donc essentiellement du volume de données lues sur disque et du débit en lecture et en écriture asynchrone des disques. L'objectif de cette étude est qu'avec un mécanisme performant de lecture/écriture asynchrone les temps de lecture des blocs sur disque puissent être raisonnablement recouverts par les temps de calcul.

Dans cette étude préliminaire, nous nous sommes limités au cas d'une distribution 1D de la matrice ; ce choix se justifie par le fait que la granularité des accès au disque doit être suffisante pour pouvoir profiter de toute la bande passante dans les échanges entre les disques et la mémoire. Ainsi dans une version 1D, et sur les cas de grande taille, la taille d'un bloc-colonne sera de l'ordre du mégaoctets en double précision, alors qu'un bloc, 120×120 par exemple, ne représente qu'à peu près 115 KiloOctets toujours en double précision.

Comme nous l'avons vu au paragraphe 5.1, les blocs de contributions agrégées peuvent représenter une part non négligeable des données allouées en mémoire. Cependant, le mécanisme d'agrégation partiel est incompatible avec un solveur "Out-of-Core" efficace ; en effet, les AUBs partiellement agrégés sont généralement émis par anticipation pour des tâches de calcul qui ne seront pas traitées tout de suite sur le processeur récepteur. Un tel mécanisme aurait donc pour effet d'obliger le chargement en mémoire de blocs-colonnes dont on n'a pas besoin pour les calculs imminents, et générerait donc beaucoup de va-et-vient entre la mémoire et le disque. Dans toute la suite, on considèrera donc, dans le contexte "Out of Core", uniquement la stratégie d'agrégation totale.

Le mécanisme "Out-of-Core" doit donc prendre en compte à la fois les accès aux blocs-colonnes locaux de la matrice, et les accès aux blocs de contributions agrégées. Ainsi nous désignerons indifféremmemt par *bloc mémoire* les AUBs et les blocs-colonnes de la matrice. A partir de l'ordonnancement des tâches de calcul et de communication, l'ordre d'accès à tous les blocs mémoires est donc connu. Ces données accédées au cours du calcul d'un bloc-colonne k sont :

- les blocs-colonnes locaux appartenant à $BCol(k)$ (voir paragraphe 1.2) qui reçoivent les contributions calculées au cours de la tâche de calcul `COMP_1D(`k`)` ;
- les AUBs dans lesquels sont ajoutées les contributions du bloc-colonne k pour les blocs-colonnes non locaux.

La stratégie qui consisterait à ne remettre sur disque que les blocs-colonnes dont on ne se sert plus dans la suite du calcul (i.e. les blocs-colonnes déjà factorisés) peut se révéler dans beaucoup de cas insuffisante. En effet, l'allocation simultanée des AUBs et des blocs-colonnes appartenant à $BCol(k)$ pour un bloc-colonne k déjà factorisé, peut rapidement saturer la mémoire s'il n'y a pas de renvoi sur disque. Pour minimiser le nombre de va-et-vient des blocs mémoires entre le disque et la mémoire vive, il convient alors de choisir parmi les blocs présents en mémoire ceux qui seront réaccédés *le plus tard dans le calcul.*

La figure 5.5 donne l'algorithme de choix des blocs à remettre sur disque. Pour décrire notre algorithme, nous utiliserons les notations suivantes :

- r : nombre d'accès à des blocs mémoires au cours de la factorisation sur un processeur ;
- bloc_accédé[i] : indice du bloc mémoire accédé lors du $i^{\text{ème}}$ accès mémoire ;
- $\mathcal{A}$: ensemble des blocs mémoires présents en mémoire vive ;

- taille(b) : taille en octets du bloc mémoire b ;
- mémoire_occupée : taille de la mémoire occupée par des blocs mémoires ;
- limite_mémoire : limite de taille mémoire que l'on ne veut pas dépasser ;
- dernier_accès[b] : variable indiquant l'indice de l'accès le plus récent au bloc mémoire b ;
- remettre_sur_disque(b, i) : indique que le bloc mémoire b devra être remis sur disque après le $i^{\text{ème}}$ accès mémoire lors de la factorisation parallèle ;
- calcul_fini(b) : renvoie *vraie* si le bloc b ne reservira plus dans les calculs à venir (il ne sera donc plus réaccédé).

$\mathcal{A} \leftarrow \emptyset$;
mémoire_occupée $\leftarrow 0$;
Pour $i = 1$ à r **faire**
  $b \leftarrow$ bloc_accédé[i] ;
  Si ($b \notin \mathcal{A}$) **alors**
    mémoire_occupée $\leftarrow$ mémoire_occupée$+$ taille(b) ;
    Tant que mémoire_occupée $>$ limite_mémoire **faire**
      Soit $c \in \mathcal{A}$, tel que $c \neq b$ et c est le bloc
      mémoire de $\mathcal{A}$ qui sera réaccédé le plus tard ;
      $\mathcal{A} \leftarrow \mathcal{A} \setminus \{c\}$;
      mémoire_occupée $\leftarrow$ mémoire_occupée$-$ taille(c) ;
      remettre_sur_disque(c, dernier_accès[c]) ;
  dernier_accès[b] $\leftarrow i$;
  Si (calcul_fini(b)= *vraie*) **alors**
    mémoire_occupée $\leftarrow$ mémoire_occupée$-$ taille(b) ;
    remettre_sur_disque(b, i) ;
    $\mathcal{A} \leftarrow \mathcal{A} \setminus \{b\}$;
  Sinon
    $\mathcal{A} \leftarrow \mathcal{A} \cup \{b\}$;
    dernier_accès[b] $\leftarrow i$;

FIG. 5.5 – Algorithme du choix des blocs mémoires à remettre sur disque en fonction d'une limite mémoire.

Ainsi, avant le début de la factorisation, chaque processeur exécute l'algorithme décrit à la figure 5.5. Cette algorithme utilise le vecteur bloc_accédé[.] qui se déduit directement du vecteur d'ordonnancement des tâches de calcul et d'envoi des AUBs. L'algorithme consiste, en prévoyant la taille de la mémoire occupée tout au long des accès mémoire, à choisir parmi les blocs présents en mémoire ceux qui devront lors de la factorisation être remis sur disque pour respecter la limite mémoire choisie.

Ces blocs sont choisis en prenant ceux qui seront réaccédés (et auront donc besoin d'être rechargés en mémoire) le plus tard possible (lignes 7 à 9 de l'algorithme). Ce critère permet de minimiser, en fonction de la limite mémoire, le volume de données écrites et lues sur le disque.

Le tableau 5.3 montre maintenant la quantité de données lues sur disque en fonction de la limite mémoire que l'on se fixe et ce pour une factorisation parallèle sur 32 processeurs. Les valeurs en mégaoctets sont par processeur. La colonne "Pic sans OOC" donne le pic de mémoire vive atteint lors de la factorisation avec agrégation totale des contributions et sans "Out-of-Core" ; les valeurs de cette colonne représentent donc la quantité de mémoire minimum nécessaire pour factoriser la matrice sans "Out-of-Core". La colonne "Pic avec OOC " correspond à une factorisation "Out-of-Core" qui ne lit les données locales de la matrice *qu'une fois*. La valeur la plus à droite dans cette colonne correspond à la taille de la matrice locale. Dans les colonnes "50 %", "25 %" et "10 %", la valeur à gauche correspond au pic mémoire de la factorisation avec "Out-of-Core" (colonne "Pic avec OOC") réduit du pourcentage indiqué en haut de la colonne ; la valeur de droite correspond au volume de données lues sur le disque (nécessairement plus que la totalité de la matrice locale puisqu'il faut mettre en œuvre les allers-retours sur disque calculés par notre algorithme) .

Une première constatation (naturelle) que l'on peut faire est que la quantité de mémoire nécessaire par rapport au volume des lectures nécessaires sur le disque est très variable selon le type de la matrice : dans le cas d'une matrice issue d'un (vrai) maillage 3D volumique, comme par exemple la CUBE47 ou l'AUDI, le rapport est beaucoup plus important que dans le cas des matrices issues de maillages 2D ou 2D$\frac{1}{2}$ "épais", comme par exemple les maillages COUPOLE. Ceci est dû à deux phénomènes qui interviennent de façon plus prononcée sur les matrices de maillage 3D volumique. Tout d'abord la masse d'AUBs est beaucoup plus importante pour ces matrices (sur la AUDI la mémoire nécessaire pour une exécution est presque 2 fois supérieure au volume des données locales de la matrice). D'autre part, les premiers séparateurs d'un graphe 3D volumique sont de grande taille (asymptotiquement en $\mathcal{O}(n^{\frac{2}{3}})$; ainsi la taille du premier séparateur correspond, pour notre renumérotation (voir paragraphe 3.1), au nombre d'inconnues dans le dernier bloc plein de la matrice (c'est la raçine de l'arbre d'élimination avant redécoupage). Lorsque l'on a besoin d'ajouter des contributions dans des blocs-colonnes appartenant à ce super-nœud redécoupé, il est nécessaire d'y accéder en mémoire. Ainsi la taille de bloc plein et le couplage des autres blocs-colonnes avec ces blocs pleins favorisent, pour ces cas 3D volumique, une plus grande consommation de mémoire.

On constate ainsi sur le tableau 5.3, que pour diviser la taille mémoire nécessaire par environ 3.4 (622.8 / 183.8) pour l'AUDI il faut lire environ 13.4 fois la taille de la matrice locale (4070.1 / 303.5) sur chaque processeur. Par contre sur le cas COUP3000T, pour diviser la taille mémoire nécessaire par 100, il faut seulement lire environs 21 fois la taille de la matrice locale sur un processeur.

TAB. 5.3 – Evolution (mémoire nécessaire / volume de lecture sur disque) pour une factorisation $L.L^t$ sur 32 processeurs.

Nom	Pic sans OOC	Pic avec OOC		50 %		25 %		10 %	
CUBE47	46.1	38.3	12.1	19.1	74.3	9.6	139.5	3.8	379.3
SHIPSEC5	34.6	24.1	14.1	12.0	106.17	6.0	372.3	2.4	442.5
SHIP003	34.9	23.0	14.7	11.5	100.5	5.8	407.1	2.3	557.4
BMWCRA1	27.1	12.2	16.5	6.1	235.5	3.05	364.6	1.2	472.6
COUP2000T	207.3	24.5	172.2	12.3	873.3	6.1	3175.4	2.5	4312.1
COUP3000T	298.7	29.7	258.3	14.9	2226.7	7.4	3592.8	3.0	6264.4
AUDI	622.8	367.3	303.5	183.8	4070.1	91.9	10165.1	36.73	14756.5

La figure 5.6 montre sur le cas CUBE_47, pour différentes limites mémoires et en appliquant le mécanisme de va-et-vient que nous venons de décrire, les courbes d'occupation de la mémoire vive obtenues lors d'une factorisation sur 32 processeurs (les courbes de mémoire sont obtenues pour un processeur), au fur-et-à mesure des accès aux blocs-colonnes et aux AUBs. La courbe "illimité" correspond au pic mémoire obtenu pour une factorisation "Out-of-Core" sans limite mémoire. On voit sur cette courbe que le pic mémoire est atteint vers le 15000$^{\text{ème}}$ accès à un bloc mémoire et vaut environ 38Mo. Les autres courbes correspondent à l'occupation de la mémoire vive lorsqu'on fixe une limite mémoire qui oblige à utiliser le va-et-vient calculé au plus juste par l'algorithme 5.5. On peut voir sur la courbe avec limite à 5Mo que pour ne pas dépasser la limite mémoire imposée, on est obligé de commencer à remettre des blocs mémoires sur disque très tôt dans le calcul.

5.3 Bilan sur les aspects mémoires

Nous avons vu dans ce chapitre deux moyens de réguler la consommation de mémoire au cours de la factorisation parallèle. Le mécanisme d'agrégation partielle permet de contrôler les surcoûts mémoires dûs à l'agrégation totale des contributions et ainsi de préserver une certaine scalabilité mémoire. Le mécanisme "Out-of-Core", dont nous avons fait ici l'étude préliminaire, devrait permettre de résoudre des systèmes de tailles très importantes. Dans ces cas, c'est l'ordonnancement statique des calculs qui permet d'anticiper l'évolution de la mémoire au cours de la

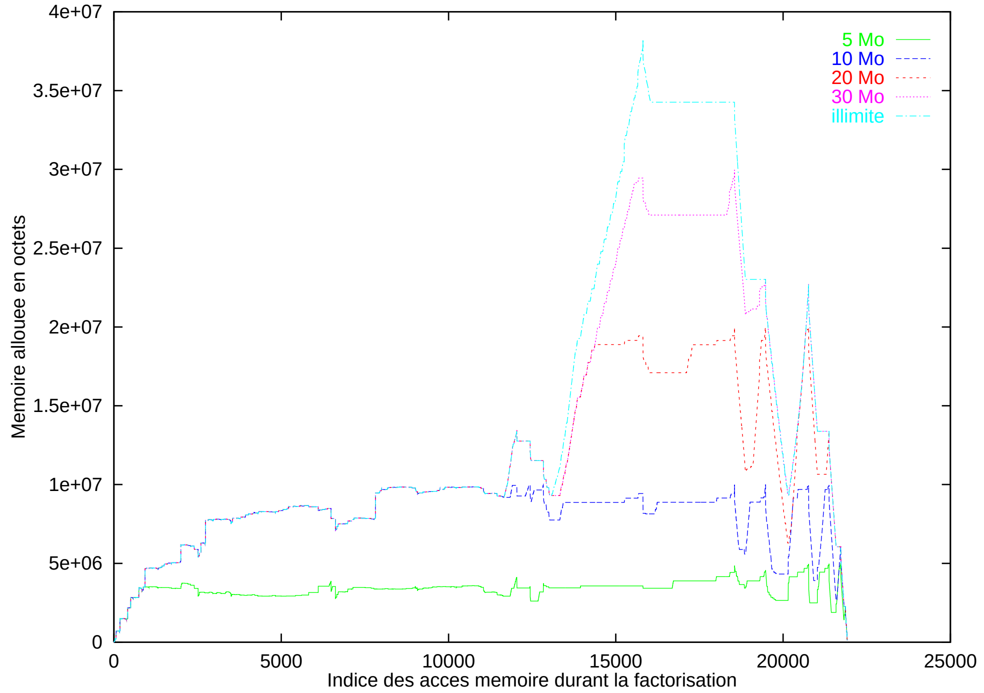


Fig. 5.6 – Courbes mémoires pour différentes limites mémoires lors de la factorisation "Out-of-Core" de la CUBE_47 sur 32 processeurs.

factorisation et ainsi de choisir de manière optimale les blocs agrégés de contributions ou les blocs-colonnes qui doivent temporairement libérer la mémoire. La mise en œuvre de nos stratégies "Out-of-Core" devrait pouvoir se faire dès que la solution logicielle que nous avons choisie sera utilisable effectivement sur l'IBM SP3 du CINES. En effet, nous avons besoin pour les lectures et écritures asynchrones de l'interface AIO (*Asynchronous Input Output*) qui n'est pas installée par défaut sur cette machine et son activation nécessite une installation spécifique sur tous les processeurs.

Conclusion et perspectives

Bilan des résultats obtenus dans cette thèse.

Les travaux qui ont été présentés dans cette thèse avaient pour objectif d'étudier la distribution des données pour la résolution de matrices creuses en parallèle par méthode directe.

Cette étude a conduit à une stratégie de régulation statique des calculs et des communications découplée en deux phases :

- la phase de repartitionnement découpe les "gros" blocs-colonnes et construit à partir de ce découpage l'ensemble des tâches de calcul. Pour chacune des tâches de calcul un ensemble de processeurs *candidats* à sa possession est créé.
- la phase de distribution et d'ordonnancement a pour rôle de distribuer les tâches de calcul (et donc les blocs associés) et de les ordonnancer. Pour cela, elle s'appuie sur une simulation en temps de la factorisation permettant, lorsqu'une tâche de calcul devient prête au sens de cette simulation, de choisir *lequel de ses processeurs candidats permettra de la traiter au plus vite.*

Ce découplage nous permet d'obtenir une stratégie de distribution et d'ordonnancement qui permet de prendre en compte de manière très fine les phénomènes qui interviennent lors de la factorisation parallèle, et les caractéristiques de calcul et de communication de la plate-fome d'exécution. Les algorithmes que nous avons présentés dans cette thèse sont assez généraux et peuvent s'appliquer à toute méthode de factorisation dont le coût des tâches de calcul est prédictible avant l'exécution.

Nous avons appliqué ces techniques à la conception du solveur parallèle PaStiX qui utilise la factorisation de Cholesky-Crout. Nous avons ainsi développé un solveur basé sur une factorisation avec une approche supernodale qui utilise une agrégation totale des contributions. L'algorithmique de régulation statique des calculs et des communications permet de piloter une factorisation parallèle utilisant une distribution mixte 1D (par blocs-colonnes) et 2D (par blocs). De plus, nous avons développé une modélisation qui permet d'adapter les stratégies de distribution et d'ordonnancement à une architecture basée sur des nœuds SMP. Nous avons ainsi validé la pertinence des stratégies de distribution par des expérimentations sur une large collection de cas tests industriels de grande taille (jusqu'à 5 millions d'inconnues).

Un aspect important de notre méthode réside dans le fait qu'elle permet de contrôler la scalabilité mémoire en modérant le surcoût mémoire dû à l'agrégation totale des contributions. Le mécanisme mis alors en œuvre est une émission anticipée de contributions partiellement agrégées. Nous avons vu comment, grâce au schéma de calcul et de communication pré-calculé lors de la phase de distribution, nous pouvions choisir de manière optimale les agrégats dont l'émission doit être anticipée au cours du calcul. Ceci nous permet d'obtenir, pour une réduction de 50 % du surcoût mémoire, des temps raisonnablement du même ordre que ceux obtenus en agrégation totale.

Nous avons conclu cette thèse par une étude préliminaire sur les possibilités qu'offre notre algorithmique dans un cadre "Out-of-Core". Cette étude montre que grâce à la connaissance a priori des accès aux différentes structures de données, il est possible d'anticiper le chargement des données par des lectures asynchrones sur disque au fur-et-à-mesure des calculs. Lorsque le mécanisme "Out-of-Core" le plus simple (consistant à charger de manière asynchrone et anticipée des données locales de la matrice et à les écrire sur disque lorsqu'elles ne sont plus utiles) n'est plus suffisant, nous utilisons un mécanisme optimisé de va-et-vient des données allouées en mémoire vive. Ce mécanisme prend en compte à la fois les blocs agrégés de contributions et les blocs de cœfficients de la matrice. Nous devrions ainsi pouvoir mettre en œuvre très rapidement une version "Out-of-Core" de PaStiX dès que l'on disposera effectivement d'entrées-sorties asynchrones sur le SP3 (utilisant l'interface AIO).

Perspectives et futurs travaux

Les méthodes algorithmiques que nous avons présentées sont en constante évolution afin de traiter de nouveaux problèmes numériques ou pour s'adapter à de nouvelles architectures. Ainsi, dans le cadre des architectures à base de nœuds SMP, nous étudions de nouvelles possibilités pour mieux exploiter la structure hiérarchique de ces machines. Nous étudions notamment la possibilité d'utiliser une programmation hybride utilisant les mécanismes de communication optimisés dans les nœuds SMP tels que les "one-sided-communications" ou l'utilisation de *threads* en mémoire partagée. Une fois que nous aurons mis en œuvre ces possibilités, nous pourrons comparer notre solveur avec une bibliothèque comme WSSMP [41] par exemple. Dans un futur proche, nous aurons aussi accès à la machine de l'IDRIS à base de nœud SMP contenant des processeurs Power 4 ; nous pourrons ainsi continuer nos expérimentations et réaliser d'éventuelles adaptations sur cette nouvelle machine.

Comme nous l'avons dit dans le paragraphe 4.4, il serait aussi intéressant de faire une comparaison approfondie en temps et en utilisation de la mémoire entre l'approche supernodale utilisé par PaStiX et l'approche multifrontale. Ainsi nous pensons effectuer cette comparaison entre PaStiX et MUMPS [7,8] sur ces points. A plus long terme, nous envisageons d'appliquer nos techniques de régulation statique des calculs et des communications à d'autres types de factorisations comme la factorisation $Q.R$ ou la factorisation $L.U$ à pivotage statique.

La dernière perspective de nos travaux consiste en la mise en place de nouveaux solveurs pour "faire sauter" le verrou de la limite en taille des solveurs parallèles directs actuels (on vise bien au delà des 10 millions d'inconnues pour des problèmes tridimensionnels).

Il s'agit de travailler sur des solveurs par blocs (pour profiter toujours de l'efficacité des primitives BLAS) *hybrides* basés sur un couplage fin entre méthodes directes et méthodes itératives. A partir de numérotations de type dissections emboîtées, l'objectif est de calculer en parallèle un préconditionneur de type Cholesky incomplet par blocs qui sera ensuite intégré dans des versions parallèles par blocs d'algorithmes de type gradient conjugué ou GMRES. Des premières études réalisées à la NASA et allant dans ce sens sont prometteuses [52, 53].

L'originalité est dans le calcul du préconditionneur qui sera issu d'une *factorisation symbolique par bloc incomplète* dans laquelle on pourra *de manière adaptative* choisir le niveau de remplissage. L'objectif a priori sera de calculer le vrai remplissage dans tous les sous domaines (cela se traduira donc par l'application en parallèle d'une méthode directe dans ces sous domaines), puis de calculer un remplissage approché pour le reste de la matrice, zone où interviendra plus spécifiquement le processus itératif. L'étude globale portera plus généralement sur les points suivants.

- L'impact de la numérotation sur la qualité du préconditionneur : on sait déjà que la dissection emboîtée a tendance naturellement à limiter la longueur des chaînes de remplissage dans l'élimination de Gauss ;
- L'étude du facteur de blocage (taille des blocs pour la partition de la matrice) et celle du seuil de basculement entre la zone "avec remplissage exact" et la zone "avec remplissage approché" : il faut mixer le critère d'économie mémoire qui est fondamental, et celui de la qualité du préconditionneur parallèle ;
- La vitesse de convergence du gradient conjugué ou du GMRES ainsi préconditionné ;
- L'étude fondamentale des relations entre la qualité du préconditionnement et la structure du graphe valué associé à la matrice du préconditionneur [19].

Il sera donc intéressant d'utiliser toutes les techniques développées dans cette thèse pour déterminer une distribution des données et des calculs pour calculer en parallèle sur diverses plates-formes la matrice par blocs du préconditionneur.

L'ensemble de cette étude se fera en particulier dans le cadre d'un Projet NSF/INRIA mis en place en 2001 et mettant en commun les compétences de spécialistes américains (Yousef Saad/Minneapolis, Randall Bramley/Indiana, Esmond Ng/Berkeley, Maria Sosonkina/Minneapolis, John Gilbert/Xerocs) et français (Patrick Amestoy/ENSEEIHT-IRIT, Frédéric Desprez/INRIA, Bernard Philippe/IRISA, Jean Roman/LaBRI).

Bibliographie

[1] BLACS, PBLAS, PESSL and ScaLAPACK libraries on the IBM SP. http ://www.citeseer.nj.nec.com/.

[2] MPI : A message passing interface standard. *Int. Journal of SuperComputer Applications*, 1994.

[3] Scalapack users' guide : Preliminary draft : 18 may, 1996. http ://www.netlib.org/scalapack/scalapack.

[4] MPI-2 : Extensions to the message passing interface. 1997, July 12. http ://www.mpi-forum.org/docs/.

[5] P. Amestoy, T. Davis, and I. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. and Appl.*, pages 886–905, 1996.

[6] P. Amestoy and I. S. Duff. Memory management issues in sparse multifrontal methods on multiprocessors. *Int. J. Supercomputer Applics*, 7 :64–82, 1993.

[7] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. In *Comput. Methods Appl. Mech. Eng.*, pages 501–520. 2000.

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

[9] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and X. S. Li. Analysis and comparison of two general sparse solvers for distributed memory computers. *ACM Transactions on Mathematical Software*, 2001. to appear.

[10] P. R. Amestoy, I. S. Duff, and C. Puglisi. Multifrontal QR factorization in a multiprocessor environment. *Int. Journal of Num. Linear Alg. and Appl.*, 3(4) :275–300, 1996.

[11] A. Ashcraft. The fan-both family of column-based distributed cholesky factorization algorithms. In Alan George, John R. Gilbert, and Joseph W.-H. Liu editors, editors, *Graph Theory and Sparse Matrix Computations*. Springer-Verlag, New York, NY, 1993.

[12] A. 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.

[13] A. 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, 1990.

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

[15] C. Ashcraft, S. C. Eisenstat, and J. W.-H. Liu. A fan-in algorithm for distributed sparse numerical factorization. *SIAM J. Sci. Stat. Comput.*, 11(3) :593–599, 1990.

[16] C. Ashcraft, S. C. Eisenstat, J. W.-H. Liu, B. W. Peyton, and A. Sherman. A compute-ahead implementation of the fan-in sparse distributed factorization scheme. Technical report, Departement of Computer Science, York University, September 1994.

[17] C. Ashcraft, S. C. Eisenstat, J. W.-H. Liu, and A. Sherman. A comparison of three column based distributed sparse factorization schemes. In *Proc. Fifth SIAM Conf. on Parallel Processing for Scientific Computing*, 1991.

[18] C. Ashcraft and R. G. Grimes. SPOOLES : An objected oriented sparse matrix library. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, March 1999.

[19] M. Bern, J.R. Gilbert, B. Hendrickson, N. Nguyen, and S. Toledo. Support - graph preconditionners. Submitted to SIAM J. Matrix Anal. Appl.

[20] L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley. Scalapack users' guide, 1997.

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

[22] J. M. Conroy, S. G. Kratzer, and R. F. Lucas. Multifrontal sparse solvers in message passing and data parallel environments - a comparitive study. In *Proceedings of PARCO*, 1993.

[23] J.W. Demmel, J.R. Gilbert, X.S. Li, and J.W. Liu. A supernodal approach to sparse partial pivoting. *Tech. Rep. CSL-94-14, Xerox Palo Alto Research Center*, 1995.

[24] J.W. Demmel, J.R. Gilbert, X.S. Li, and J.W. Liu. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. 1997.

[25] R. Clint Whaleyand Jack J. Dongarra. Automatically tuned linear algebra software. Technical Report UT-CS-97-366, 1997.

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

[27] I. S. Duff. Sparse numerical linear algebra : direct methods and preconditioning. Technical Report TR/PA/96/22, CERFACS, 1996.

[28] I. S. Duff and J. Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. 20(4) :889–901, 1999.

[29] I. S. Duff and J. Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. Technical Report RAL-TR-1999-030, 1999. To appear in *SIAM Journal on Matrix Analysis and Applications*.

[30] 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.

[31] 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.

[32] 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 editors, editors, *Parallélisme et applications irrégulières*, pages 135–147. Hermès, 1995.

[33] Coffman, Jr. E. G., M. R. Garey, and D. S. Johnson. Approximation algorithms for bin-packing - an updated survey. 1984.

[34] K. A. Gallivan and al. Parallel algorithms for matrix computations. *SIAM, Philadelphia, PA*, 1990.

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

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

[37] A. George and J. W.-H. Liu. *Computer solution of large sparse positive definite systems.* Prentice Hall, 1981.

[38] D. Goudin, P. Hénon, F. Pellegrini, P. Ramet, J. Roman, and J.-J. Pesqué. Parallel sparse linear algebra and application to structural mechanics. *Numerical Algorithms, Baltzer Science Publisher*, 24 :371–391, 2000.

[39] D. Goudin and J. Roman. A scalable parallel assembly for irregular meshes based on a block distribution for a parallel block direct solver. In *Proceedings of PARA'2000,* LNCS 1947. Springer Verlag, 2000.

[40] David Goudin. *Mise en œuvre d'une Bibliothèque d'Outils pour la Résolution Parallèle Hautes Performances par Méthode Directe de Grands Systèmes Linéaires Creux et application à un Code de Mécanique des Structures.* PhD thesis, LaBRI, Université Bordeaux 1, Talence, France, November 2000.

[41] A. Gupta, M. Joshi, and V. Kumar. Wssmp : Watson symmetric sparse matrix package, 1997.

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

[43] 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, 1994.

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

[45] 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, 1993.

[46] P. Hénon, P. Ramet, and J. Roman. PaStiX : A Parallel Direct Solver for Sparse SPD Matrices based on Efficient Static Scheduling and Memory Managment. Tenth SIAM Conference on Parallel Processing for Scientific Computing, Portsmouth, Virginie, USA, 2001.

[47] P. Hénon, P. Ramet, and J. Roman. A Mapping and Scheduling Algorithm for Parallel Sparse Fan-In Numerical Factorization. In *Proceedings of EuroPAR'99*, number 1685 in Lecture Notes in Computer Science, pages 1059–1067. Springer Verlag, September 1999.

[48] P. Hénon, P. Ramet, and J. Roman. Pastix : A high-performance parallel direct solver for sparse symmetric definite systems. In *Actes of PMAA'2K, Neuchatel, Suisse*, August 2000.

[49] P. Hénon, P. Ramet, and J. Roman. PaStiX : A Parallel Sparse Direct Solver Based on a Static Scheduling for Mixed 1D/2D Block Distributions. In *Proceedings of Irregular'2000*, number 1800 in Lecture Notes in Computer Science, pages 519–525. Springer Verlag, May 2000.

[50] P. Hénon, P. Ramet, and J. Roman. PaStiX : A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. *Parallel Computing, à paraître*, 2002.

[51] HSL. A collection of Fortran codes for large scale scientific computation, 2000. http ://www.cse.clrc.ac.uk/Activity/HSL.

[52] D. Hysom and A. Pothen. Efficient parallel computation of ILU(k) preconditioners. Technical Report 2000 – 23, NASA, may 2000.

[53] D. Hysom and A. Pothen. Parallel ILU ordering and convergence relationships : numerical experiments. Technical Report 2000 – 24, NASA, may 2000.

[54] S. Hammarling J. Dongarra, J. Du Croz and R. J. Hanson. A extended set of basic linear algebra subroutines. *ACM Trans. Mathematical Software*, 1988.

[55] J. Du Croz J.J. Dongarra, I.S. Duff and S. Hammarling. A set of level 3 basic linear algebra subprogramms. *ACM Trans. Mathematical Software*, 1990.

[56] M. John and R. Rice. A grid-based subtree-subcube assignment strategy for solving partial differential equations on hypercubes, 1992.

[57] 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.

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

[59] G. Karypis and 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.

[60] X. S. Li. *Sparse Gaussian Elimination on High Performance Computers.* PhD thesis, University of California at Berkeley, 1996.

[61] X.S. Li and J.W. Demmel. A scalable sparse direct solver using static pivoting. *In Proceedings of the Ninth SIAM Conference on Parallel Processing for ScientificComputing, San Antonio, Texas*, March 1999.

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

[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*, number 1225 in Lecture Notes in Computer Science, pages 370–378. Springer Verlag, 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*, number 1586 in Lecture Notes in Computer Science, pages 986–995. Springer Verlag, April 1999. Extended paper appeared in Concurrency : Practice and Experience, 12 :69-84, 2000.

[66] A. Pothen and C. Sun. A mapping algorithm for parallel sparse Cholesky factorization. *SIAM J. Sci. Comput.*, 14(5) :1253–1257, September 1993.

[67] Pierre Ramet. *Optimisation de la Communication et de la Distribution des Données pour des Solveurs Parallèles Directs en Algèbre Linéaire Dense et Creuse.* PhD thesis, LaBRI, Université Bordeaux 1, Talence, France, jan 2000.

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

[69] 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.

[70] Donald J. Rose. In Ronald C.Read, editor, *Graph Theory and Computing*, pages 183–217. Academic Press, 1972.

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

[72] 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.

[73] 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.

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

[75] E. Rothberg and R. Schreiber. Improved load distribution in parallel sparse cholesky factorization. *Supercomputing'94*, 1994.

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

[77] Olaf Schenk, Klaus Gartner, Wolfgang Fichtner, and Andreas Stricker. Pardiso : a high-performance serial and parallel sparse linear solver in semiconductor device simulation. *Future Generation Computer System*, 2000.

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

[79] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, New Jersey, 1992.

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

[81] David W. Walker. Software libraries for linear algebra computations on high performance computers. *SIAM Review*, 37(2) :151–170, 1995.

Annexes

Annexe A

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*, Numerical Algorithms volume 24, pages 371-391, 2000.

[2] P. Hénon, P. Ramet et J. Roman. *PaStiX : A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems.* Parallel Computing, à paraître.

Congrès internationaux avec comité de sélection et avec actes :

[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.* Actes de Irregular'2000, Cancun, Mexique, mai 2000, LNCS 1800, pages 519-525. Springer Verlag.

[5] P. Hénon, P. Ramet et J. Roman. *PaStiX : A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems.* PMAA'2000, Neuchâtel, Suisse, août 2000.

[6] D. Goudin, P. Hénon, F. Pellegrini, P. Ramet, J. Roman et J.-J. Pesqué. *Description of the EMILIO Software Processing Chain and Application to Structural Mechanics.* PMAA'2000, Neuchâtel, Suisse, août 2000.

[7] P. Hénon, P. Ramet et J. Roman. *PaStiX : A Parallel Direct Solver for Sparse SPD Matrices based on Efficient Static Scheduling and Memory Managment.* SIAM Conference PPSC'2001, Portsmouth, Virginie, USA, mars 2001.

Congrès nationaux avec comité de sélection et avec actes :

[8] 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.

[9] D. Goudin, P. Hénon, F. Pellegrini, P. Ramet, J. Roman et J.-J. Pesqué. *Algèbre Linéaire Creuse Hautes Performances : Application à la Mécanique des Structures.* iHPerf'2000, Aussois, décembre 2000.

[10] P. Hénon, P. Ramet. *PaStiX : Un solveur parallèle direct pour des matrices creuses symétriques définies positives basé sur un ordonnancement statique performant et sur une gestion mémoire efficace.* RenPar'2001, Paris, France, avril 2001.

Workshops internationaux sur invitation :

[11] D. Goudin, P. Hénon, F. Pellegrini, P. Ramet, J. Roman et J.-J. Pesqué. *Parallel Sparse Linear Algebra and Application to Structural Mechanics.* ACTC'2000, Paris, France, mai 2000.