n^0 d'ordre : 97
n^0 bibliothèque : 98ENSL0097

THESE

présentée par

Stéphane DOMAS

pour obtenir le titre de

DOCTEUR de L'ÉCOLE NORMALE SUPÉRIEURE de LYON

Spécialité : **Informatique**

Contribution à l'Ecriture et à l'Extension d'une Bibliothèque d'Algèbre Linéaire Parallèle

Date de soutenance: 23 Octobre 1998

Composition du jury:

Rapporteurs	Pierre	**MANNEBACK**
	Jean	**ROMAN**
Examinateurs	Olivier	**COULAUD**
	Yves	**ROBERT**
Directeurs de thèse	Frédéric	**DESPREZ**
	Bernard	**TOURANCHEAU**

Thèse préparée au sein du
Laboratoire de l'Informatique du Parallélisme
URA 1398 du CNRS, ENS Lyon.

Je remercie profondément les membres du Jury :

- Pierre Manneback et Jean Roman, pour avoir accepté d'être mes rapporteurs, pour leur remarques sur le manuscrit, pour leur symphatie et leur soutient malgré le timing très serré et les problèmes "administratifs".
- Olivier Coulaud, pour sa présence en temps qu'examinateur.
- Yves Robert, pour son écoute et ses conseils, pour ses "piques" du coin café, pour savoir rester proche des thésards.
- Frédéric Desprez et Bernard Tourancheau, pour m'avoir soutenu tout au long de cette thèse, pour avoir compris et respecté mon indépendance de recherche, pour tous leur conseils et leur expérience librement partagée, pour m'avoir remis sur les rails quand je doutais.

A tous va ma reconnaissance.

Pour plagier honteusement une des chansons ayant peuplé mon univers enfantin, je crois qu'il est temps d'entonner le rituel : “voici venu, le temps, des remerciements...”. Bien entendu, quoi de plus dur que d'ordonnancer tout le beau monde qui a contribué à l'aboutissement de ces travaux. La NP-complétude de ce problème ne fait aucun doute même s'il n'y a aucune preuve formelle, au grand dam des fondamentalistes. Les moyens d'organiser le protocole sont cependant nombreux. Pour en citer quelques-uns : aléatoire, par ordre de parenté, classification automatique... Il est d'ailleurs étrange de constater la ressemblance étroite entre le sujet de la thèse et la structure des remerciements ! Passons rapidement sur ces possibilités car le choix d'une de ces méthodes est lui-aussi cornellien : choix aléatoire, choix par ordre de parenté, choix par classification automatique... Etrangement récursif, n'est-il pas ?

Aussi, rejetons une fois pour toute ce fatras inextricable de choix et prenons une attitude totalement objective, c'est-à-dire la mienne ! Puisque j'ai notifié mon enfance, remontons donc aux sources de cette passion et cheminons le long de cette périlleuse ascension vers les sommets brumeux qu'on atteint, généralement très facilement après le pot de thèse !

La première personne que je tiens à remercier est mon arrière-arrière-grande tante. Quel rapport avec l'informatique me direz-vous ? Aucun bien entendu, sinon le fait que c'est le début d'une longue liste d'enseignants, dont j'espère être le cinquième maillon. Enseignement auquel je ne me destinais absolument pas, mais dont la thèse m'a fait découvrir l'intérêt.

La personne suivante est Mr Sinclair pour avoir sorti en 1980 une petite boite noire relativement plate, qu'il a fallu renvoyer trois fois avant qu'elle fonctionne. Tels les Incas, émerveillés devant la splendeur des Conquistadors, mon père et moi nous sommes très vite retrouvés esclaves de cette chose au clavier mou et au cerveau gros comme un petit pois (1Ko de mémoire vive, ça fait pas beaucoup). Des nuits entières ont passé avant que nous ne maîtrisions cette machine infernale, en la scotchant définitivement à une planche de bois. Une sorte de crucifixion. Il faut dire qu'elle avait la désagréable habitude d'afficher son mécontentement quand on la secouait trop fort ! Nous avons donc survécu à cette peste, mais définitivement marqués par le virus de l'informatique qui avait germé dans nos cerveau.

Pour cette même époque, je remercie mon père d'avoir, malgré ou peut être grâce à des circonstances que je tairai, acheté cet inutile et ridicule bout de plastique noir qui ne savait afficher que “/0 error” la première fois, rien la deuxième, et “>” la troisième et bonne fois. Présenté de manière uniquement pécuniaire, cet acte n'en reflète absolument pas l'origine. Elle est plutôt dans l'irrémédiable attrait de mon père pour les gadgets, mêlé avec cet avant-gardisme soixante-huitard et bien entendu sa passion pour l'enseignement et dans mon cas, l'éducation. C'est grâce à ces qualités que je suis maintenant en passe de devenir docteur en informatique. Il existe beaucoup d'autres raisons (dont certaines uniquement pécuniaires !), mais elles se passent de commentaires car nous les connaissons au plus profond de nous, et il n'y a pas vraiment de mots pour les exprimer.

Toujours dans l'ordre chronologique, je pense à ma mère qui m'a fait découvrir la grande ville. Lyon, ses cinémas, son agitation, son caractère accariatre (la ville bien entendu !)... Même si cela a été une époque difficile, elle m'a toujours supporté quelque soit la définition qu'on mette sur ce mot. Pour toutes les choses que je t'ai demandé que tu m'as donné, pour toutes celles que tu m'as donné sans que je les demande, merci...

Avançant toujours dans le temps, je ne remercie ni Mr Collège ni vraiment Mr Lycée. On ne peut pas dire qu'ils aient vraiment servi dans l'édification de mon avenir sauf si l'on considère que toute erreur est bonne. Il semble à présent que je ne suis pas l'ingénieur vers lequel ils me destinaient.

Passons donc directement à Mme Université où les choses s'accélèrent. Un remerciement spécial pour Cyril (Gavoille pour ceux qui n'avaient pas compris !) qui m'a précédé dans la voie qui mène à l'ENS. Il m'a notamment donné envie de faire une licence/maîtrise d'informatique et de faire mon stage de maîtrise à l'ENS. Il pèse donc un grand poids dans les choix que j'ai suivi pour aboutir devant cette minable SPARC 1 sur laquelle je suis en train de rédiger ces remerciements. Un remerciement spécial également pour tous les profs de licence, de maîtrise, de D.E.A. Spécial car même les “mauvais” n'ont pas réussi à me dégoutter de cette formation : le séchage de leur cours m'a permis de faire plein de choses passionnantes ! Spécial car tous m'ont conforté dans l'idée que la richesse vient de la diversité et que l'informatique est suffisamment vaste pour que jamais je ne m'ennuie. Je ne cite aucun nom, ils se reconnaîtront.

Revenons au stage de maîtrise puisque c'est ainsi que j'ai connu Fred et Bernard. Je commence par remercier le deuxième pour avoir débarqué dans la salle de formation avec Michel, discuté environ 1/2 heure sur la factorisation LU, repartir en me laissant persister dans mes erreurs, et tout cela sans que je sache ni qui il était, ni qui était le gars avec une cravate à coté de lui. Drôle d'impression mais depuis j'ai appris que l'erreur est souvent nécessaire, que l'on doit écouter sinon suivre les conseils de ceux qui ont l'expérience d'un domaine et que le gars avec la cravate était le directeur du labo. Ouuppss! Je le remercie donc car il m'a fait confiance malgré mon caractère indépendant, et su me guider par quelques mots qui n'avaient l'air de rien mais qui ont impliqué certains réajustements dans ma vision de la recherche. Pour ces mêmes qualités, je remercie Fred qui a par contre eu le mérite de se présenter, même si je n'ai pas compris son nom la première fois ! Merci pour s'être décarcassé lorsque les choses n'allaient pas dans le bon sens, pour les pâtes au pamplemousse, pour les 8kms de footing par 40 degrés sous le soleil de Knoxville, enfin bref, pour tous ces choses un peu décalées de la réalité et du train-train quotidien. Et comme pourrait le dire (narquoisement) Yves “tout bon directeur de thèse se doit d'assurer que les travaux de son étudiant soient publiés et présentés dans une conférence.”. Sur ce, il ajoute, en temps de crise budgétaire : “Internationale pour le directeur et nationale pour l'étudiant.”. Merci donc à tous les deux pour m'avoir permis de présenter nos articles (voire une fois, celui de Makan !) en divers points du globe. Certains n'ont pas cette chance.

Par exemple, Françoise, qui malgré tous les problèmes et le peu de moyens qu'elle avait parfois à disposition a su persister et trouver sa voie. Dur, et mal vu peut être, pour une mathématicienne que de travailler en étroite collaboration avec des informaticiens, parallé-liste de surcroît. Merci donc pour ta gentillesse et surtout pour la confiance que tu as placé en nous.

Avançons jusqu'à une époque plus récente, le début de cette année 98. Alors, trois pèlerins partirent sur les routes tortueuses de la compilation automatique afin de rencontrer le pur joyau céleste au dessus de la tanière d'un enfant très difficile. Le premier s'appelait Jakaleksandri Gherber, descendant des grandes steppes noiraudes et terrilesques de la plaine Nancyéhenne. Le deuxième s'appelait Kyril Randriamrus, fier descendant d'un Consul romain et d'une Phytie Grecque, quoique certains disent... Le troisième étant le narrateur Esteban d'Omas, c'est-à-dire moi-même. Quand enfin leur quête pris fin, ils trouvèrent le grand Gourou Frildéric qui leur dit, baptisez-le. Aussi le prénomèrent-ils Alasca car Jésus était déjà pris ! En plus, c'était au début de l'année et un froid terrible sévissait dans les couloirs de la chaumière que ses pauvres habitants appelaient LIP, pour des raisons sans doute obscures et mystiques. Puis vint le temps de l'éveil pour cette jeune création. Elle fut longue et rude comme l'hiver (Desproges, que je remercie également pour son humour, aurait sans soute fait de cette remarque une belle histoire de bûcheron canadien) . Le premier s'occupa de prendre ses mesures régulièrement pour constater l'évolution de ses performance. Il construisit même une diabolique machine pour traiter automatiquement tous ces chiffres. Le deuxième lui apprit à lire et écrire la langue des Dieux, le Fortran. Le troisième lui appris l'altruisme et le leadership afin de toujours jauger correctement ce qui l'entourait mais aussi de savoir redistribuer ce qu'il amassait. Malheureusement, après six mois de dur labeur, Alasca était comme trois parties distinctes. Alors le grand Gourou Frildéric vint de nouveau les voir et dit : “assemblez-moi tout ça vite fait car il va maintenant passer devant le tribunal des Dieux, sur le Mont Eurotopse”. Ainsi fut fait, et c'est pourquoi je remercie mes deux collaborateurs pour leur ténacité et leur bons conseils. La piscine, le ping-pong... avec eux ne sont qu'accessoires mais des rumeurs courent comme quoi ils auraient été indispensables à la création d'Alasca.

Plus récemment encore, je ne remercie pas ma station de travail qui m'a plaquée comme un malpropre pendant 3 semaines. Il a fallu que je C.R.I. de nombreuses fois pour qu'elle revienne. D'ailleurs, elle est revenue et c'est pourquoi elle s'appelle Ultrafuel et pas Alice.

Mais voici le moment où les bonnes choses prennent fin, et c'est le dur temps de la rédaction qui m'a surpris en ce début d'été. Je ne remercie donc pas Juillet et Août bien qu'ils se soient montrés relativement coopératifs en ne s'écoulant pas trop vite.

Et nous voici début septembre où le coeur plein d'appréhension j'envoie le manuscrit à mes rapporteurs. Quel accueil vont-ils réserver à ce paquet de feuilles, rédigées à la sueur de mes doigts. A ma grande honte, ils n'ont disposé que de 20 jours, après décision administrative,

pour rédiger leur rapport. Merci donc à vous, Pierre et Jean, pour avoir accepté malgré ces contraintes.

Les vingt jours qui suivent sont dignes d'un film à rebondissements. "On reporte" semble être le mot du mois mais après de multiples crises cardiaques et sueurs froides, tout s'arrange. Un grand merci a Fred, Jean-michel et tous ceux qui ont su arrondir les angles ou bien se traîner à genoux sur la moquette rugueuse du second étage (champagne pour vous !).

Enfin arrive le moment de la soutenance. Je remercie donc les membres du jury d'en faire parti ! Peut être est-ce un peu court et laconique mais ça représente beaucoup de choses pour moi que ces personnes aient acceptées, sans hésitation, sans forcément bien me connaître, de venir pérenniser mes travaux.

Bien entendu, la liste des remerciements ne s'arrête pas là. En premier lieu, le LIP et ses autochtones ont toujours été un terreau propre au développement de cette thèse. Peu importe la personne, il y a toujours eu quelqu'un avec qui discuter recherche, problème de mission, administratif, films, musique, poids des spaghettis, droit fiscal du Vatican... Merci au surréalisme des discussions du coin café pour la bonne humeur que cela apporte lors de trop longues semaines passées à déboguer. En second premier lieu, tous mes amis, (la liste est longue et ils se reconnaîtront) pour le simple fait d'exister et de partager ces tranches de vie si diverses. Merci pour la richesse d'horizons que vous me procurez. Merci enfin à ces auteurs merveilleux que sont Herbert, Vance, Zelazny, Simmons, Pratchett, Powers et bien d'autres encore, pour garder la porte ouverte sur l'imaginaire et le rêve sans lequel la recherche, de quoi que ce soit, n'est rien et n'a aucun but.

A vous tous,

MERCI.

Table des matières

Introduction **1**

1 Bibliothèques et contexte **7**

1.1 ScaLAPACK et ses composants 7

1.2 Contexte 11

2 BLAS 3 pipelinés **13**

2.1 Exemple typique 14

2.1.1 Comment pipeliner des tâches `DGEMM` 15

2.1.2 Choisir un découpage pour les calculs 16

2.1.3 Choisir un découpage pour les communications 17

2.1.4 Exemple de solution pour le découpage 17

2.1.5 Influence du temps d'exécution de chaque tâche 19

2.1.6 Conclusion 20

2.2 Cadre de travail pour le pipeline de BLAS 3 21

2.2.1 Le graphe de tâche 21

2.2.2 Considérations sur les BLAS de niveau 3 22

2.3 Recherche de l'ensemble des meilleurs découpages 23

2.4 Calcul de la taille de découpage optimale 24

2.4.1 Temps d'exécution théorique d'un BLAS de niveau 3 26

2.4.2 Temps d'exécution d'un BLAS de niveau 3 découpé en L blocs. . . . 27

2.4.3 Temps d'exécution d'un pipeline 30
2.5 Résultats expérimentaux 33
2.6 Conclusion 39

3 Factorisation *LU* **41**
3.1 De l'intérêt de la factorisation *LU* 41
3.2 Travaux précédents 42
3.3 Description des problèmes 43
3.4 La factorisation *LU* par blocs en parallèle 43
3.5 Modèle théorique 46
3.5.1 Temps d'exécution théorique 46
3.5.2 Taille de bloc optimale 48
3.5.3 Résultats sur Intel Paragon 48
3.5.4 Résultats sur une IBM SP2 49
3.5.5 Commentaires 50
3.6 Optimisations 50
3.6.1 Recouvrement de la diffusion 50
3.6.2 Recouvrement et pipelinage de l'échange de lignes 51
3.6.3 Résultats expérimentaux 54
3.7 Conclusion 56

4 Calcul des redistributions **59**
4.1 Un modèle pour la redistribution de matrices 60
4.1.1 Des études et des problèmes liés à la redistribution 60
4.1.2 Calcul de la table des messages 61
4.1.3 Ordonnancement des messages 64
4.1.4 Conclusion 74
4.2 Détermination des redistributions dans un programme F77 / ScaLAPACK . 75

4.2.1 Motivation et problématique 75
4.2.2 Un algorithme de génération des redistributions 76
4.2.3 Résultats 82
4.2.4 Conclusion 84

5 Le calcul de valeur propres 85

5.1 Introduction 85
5.2 Méthode de Yau et Lu 85
5.2.1 Implémentation parallèle de l'accélération polynomiale 86
5.2.2 La multiplication en parallèle de matrices symétriques commutantes . 88
5.2.3 Validation de notre approche par comparaison du noyau de calcul de Yau et Lu avec ScaLAPACK 94
5.3 Méthode de Jacobi 95
5.3.1 L'algorithme séquentiel 95
5.3.2 Travaux relatifs à Jacobi en parallèle 96
5.3.3 Un algorithme parallèle par blocs 96
5.3.4 Temps d'exécution théorique de l'algorithme parallèle 103
5.3.5 Discussion sur l'algorithme parallèle 108
5.4 Comparaison de Yau et Lu, et Jacobi 108

6 Conclusion et Perspectives 111

Annexe 115

Liste des publications personnelles 123

Bibliographie 125

Introduction

"...On ne peut pas être à la fois au four et au moulin."
P. Desproges, à l'Olympia.

Parallélisme de tache, parallélisme de données ou encore passage de messages, autant de paradigmes de programmation pour les machines parallèles. Dans le domaine du calcul scientifique, la troisième solution est privilégiée puisqu'elle apporte la performance. Malheureusement, cela demande une expertise aussi bien dans l'architecture des machines cibles que dans l'algorithmique distribuée. De plus, les codes sont souvent peu portables puisqu'ils sont optimisés pour telle ou telle machine. Pour diminuer la complexité de programmation et gagner en portabilité, on fait de plus en plus appel aux bibliothèques de calcul et de communication (notamment MPI et PVM [52, 31]). Comme la plupart des applications scientifiques ont des noyaux utilisant intensivement l'algèbre linéaire, ces bibliothèques en implémentent les algorithmes classiques. Cependant, une bonne utilisation de ces bibliothèques n'est pas triviale. En effet, beaucoup de paramètres influent sur la performance des routines et certaines de leurs règles d'utilisation sont particulièrement absconses au néophyte.

D'un autre côté, la programmation avec parallélisme de donnée s'avère particulièrement simple malgré les faiblesses des compilateurs paralléliseurs tels que HPF [27](High Performance Fortran). Malheureusement, dans le cas du calcul scientifique, les performances obtenues sont réduites même si le code est parfaitement portable sans avoir besoin d'utiliser des bibliothèques spéciales. Pour gagner en efficacité, on étudie de plus en plus les possibilités d'interfaçage des ces bibliothèques avec les compilateurs type HPF.

Enfin, le parallélisme de tâche se situe plus ou moins entre les deux approches. Quand il s'agit de calcul scientifique, le style d'implémentation le plus courant est celui du maitre-esclave. Le maître distribue le calcul aux esclaves. Ces derniers renvoient le résultat au maître dès qu'ils ont fini. Cette organisation permet d'avoir un schéma de communication simple et qui donc s'implémente facilement. Cependant, dans le cas d'algorithmes d'algèbre linéaire, les performances ne sont pas satisfaisantes. Les tâches peuvent également être ordonnancées ce qui implique un schéma de communication complexe. Dans ce cas, on se rapproche souvent des performances obtenues avec une programmation par passage de messages. A cela, on peut ajouter des optimisations telles que le recouvrement calcul/communication ou le pipeline pour obtenir la même performance que les bibliothèques.

De ces trois points de vue, nous retenons quatre axes principaux de recherche:

- Les performances des bibliothèques sont soumises à de nombreux paramètres tels que la distribution des données, le nombre de processeurs.... Peut-on calculer automatiquement les meilleurs paramètres quelque soit la machine?
- La portabilité diminue la performance. Peut-on gagner en performance en limitant la portabilité?
- L'utilisation des bibliothèques est fastidieuse mais elles sont performantes par rapport à un code produit par un compilateur paralléliseur. Comment mixer les deux approches en générant automatiquement les initialisations nécessaires à l'utilisation des bibliothèques? De même, comment intégrer au compilateur les moyens d'utiliser efficacement les bibliothèques avec un calcul automatique des paramètres?
- Gérer les communications entre tâches est parfois complexe surtout si l'on ajoute des optimisations telles que le pipeline. Dans le cadre bien précis de l'algèbre linéaire, est-il possible de développer une bibliothèque qui gère ces communications de manière transparente pour l'utilisateur?

En fait, nous nous sommes intéressés principalement à la bibliothèque ScaLAPACK ou ses composants connexes tels les PBLAS. Les quatre axes principaux de la thèse reposent donc sur un problème de fond: comment adapter ou optimiser ScaLAPACK et ses composants. Quant à la forme, notre cadre de travail est un projet commun d'aide à la parallélisation. L'approche globale consiste à partir d'un code Fortran 77 contenant des appels séquentiels BLAS, LAPACK ou bien parallèles PBLAS, ScaLAPACK, et de le transformer en un code entièrement parallèle. Pour cela, il faut transformer les appels séquentiels en appels parallèles et surtout gérer la distribution des matrices. Nous savons qu'il n'est pas facile pour un non expert de jongler avec la dizaine de routines nécessaires avant l'utilisation des PBLAS. La solution est d'insérer automatiquement des routines gérant ces problèmes d'initialisation lors de la transformation en code parallèle. Il faut notamment des routines gérant la distribution de chaque matrice. En HPF, cela se fait grâce à une directive qui spécifie la distribution. Nous allons voir au cours des différents chapitres comment construire des modèles théoriques des routines de ScaLAPACK et leur utilité pour calculer automatiquement la meilleure distribution pour chaque matrice utilisée par ces routines.

Cette thèse comporte cinq chapitres :

1. Présentation de ScaLAPACK et de ses composants, et un tour d'horizon des notions utilisées dans la thèse.

2. Etude sur la gestion des communications pipelinées entre différentes opérations matricielles BLAS 3, enchaînées sur plusieurs processeurs.

3. Etude et optimisation de la factorisation LU de ScaLAPACK.

4. Etude de la redistribution de matrice distribuées de manière cyclique par bloc et calcul automatique des redistributions dans un code Fortran appelant ScaLAPACK.

5. Etude sur le calcul de valeurs propres en parallèle.

Chapitre 1 - Présentation de ScaLAPACK et notions

Nous donnons dans ce chapitre une présentation rapide de la bibliothèque ScaLAPACK et de ses composants internes (BLAS, BLACS, LAPACK) ou connexes (PBLAS). Nous donnons également quelques contraintes dues à l'utilisation de ces bibliothèques, notamment sur la distribution des matrices.

Nous présentons ensuite quelques notions utilisées dans cette thèse. Nous expliquons entre autres les types de communications, le pipeline, le recouvrement/calcul communication...

Chapitre 2 - BLAS 3 pipelinés

Ce chapitre est consacré à l'optimisation du temps de calcul total de plusieurs opérations matricielles (BLAS de niveau 3) effectuées sur différents processeurs. Chaque opération matricielle est considérée comme une tâche qui nécessite les résultats du calcul d'une autre tâche pour s'exécuter. L'optimisation consiste à découper les opérations matricielles en plusieurs sous-tâches et à pipeliner les résultats partiels. On peut par exemple décomposer un produit matriciel en une suite de produits matrice-vecteur. On utilise également des communications asynchrones pour recouvrir le temps de l'envoi par un calcul partiel. Le but de l'étude est de déterminer automatiquement la taille des résultats partiels et surtout la manière de découper les opérations matricielles. Premièrement, nous donnons un algorithme qui permet de trouver l'ensemble des meilleurs découpages. Nous donnons ensuite une modélisation du temps d'exécution des routines BLAS de niveau 3. Enfin, nous nous servons de ce modèle pour déterminer le meilleur découpage ainsi que la taille optimale de découpage.

Chapitre 3 - La factorisation LU

Ce chapitre est divisé en deux parties. La première présente un modèle théorique du temps d'exécution de la factorisation LU de ScaLAPACK. La méthode utilisée pour construire le modèle n'est pas spécifique à cette routine et peut être appliquée à d'autres routines des bibliothèques ScaLAPACK ou PBLAS. La méthode exploite notamment la modélisation théorique des BLAS proposée dans le chapitre précédent. Le but est d'avoir une expression mathématique du temps d'exécution en fonction des paramètres de distribution de la matrice à factoriser. Nous en déduisons notamment la taille de bloc optimale théorique pour distribuer la matrice (cf. chapitre 1). Une comparaison avec des tailles expérimentales vient valider le modèle et la méthode de construction.

La deuxième partie utilise le modèle théorique pour trouver les phases de l'algorithme intéressantes à optimiser. Nous présentons deux optimisations possibles basées sur le recouvrement calcul/communication et le pipeline.

Chapitre 4 - Calcul des redistributions

Dans le chapitre traitant de la factorisation LU, nous montrons qu'il est possible de déterminer la meilleur distribution d'une matrice pour n'importe quelle routine PBLAS ou ScaLAPACK. Dans un code comportant plusieurs appels aux routines ScaLAPACK ou PBLAS, cela implique des redistributions puisque chaque routine peut employer une distribution différente. Mais est-il vraiment intéressant de redistribuer entre chaque routine?

La première partie du chapitre est consacrée à l'évaluation du temps de la redistribution d'une matrice distribuée par blocs cycliques. Nous montrons qu'il est impossible d'obtenir un modèle mathématique comme pour les routines ScaLAPACK mais que l'on peut déterminer ce temps de manière calculatoire. Pour cela, nous donnons une méthode efficace pour construire la table des messages échangés au cours de la redistribution. Nous présentons ensuite un algorithme d'ordonnancement de ces messages. Il construit un schéma de communication le plus court possible et donc permet d'évaluer le temps de la redistribution. Nous donnons enfin des résultats sur le comportement de l'algorithme.

La deuxième partie présente un algorithme qui génère un ensemble de redistributions pour un code F77 appelant des routines PBLAS ou ScaLAPACK. Cet ensemble est celui qui minimise le temps total d'exécution du programme. L'algorithme génère d'abord une solution initiale qu'il optimise en essayant de supprimer certaines redistributions. L'algorithme répond donc à la question posée ci-dessus.

Chapitre 5 - Calcul de valeurs propres

Ce dernier chapitre est consacré au calcul des valeurs propres d'une matrice symétrique dense. C'est en effet un problème récurrent notamment dans les calculs de simulation électromagnétique, hydrodynamique... La première partie du chapitre est consacré à la version parallèle de l'algorithme de Yau et Lu. Cet algorithme utilise un grand nombre de produits matriciels. Il est donc intéressant d'utiliser les routines PBLAS afin de paralléliser le code. On s'aperçoit cependant qu'il est possible d'optimiser le code en disposant d'une routine de multiplication matricielle adaptée. En effet, les routines PBLAS sont dans ce cas trop générales et deux fois trop de calculs sont effectués. Nous avons donc étudié l'implémentation la plus performante possible de cette routine tout en utilisant les contraintes des PBLAS. Nous donnons une comparaison de notre implémentation avec la routine de multiplication matricielle générale des PBLAS. Nous donnons ensuite une comparaison du code optimisé avec une routine de ScaLAPACK calculant les valeurs propres d'une matrice.

La deuxième partie est consacrée à l'algorithme de Jacobi. Cet algorithme est facilement parallélisable mais il n'est pas trivial de prendre en compte la symétrie, ni d'obtenir des calculs à gros grain. Nous proposons une version parallèle de l'algorithme, utilisant la symétrie et effectuant les mises à jour avec des BLAS de niveau 3. Nous donnons un temps théorique d'exécution et le comparons à celui donné dans un autre papier. Nous montrons que notre algorithme effectue moins de calculs mais plus de communications.

Enfin, après une conclusion, nous présentons des perspectives de travaux dans ce domaine.

CHAPITRE 1 Bibliothèques et contexte

"- Tu as là une jolie bibliothèque.
- Merci, dit-elle."
Corwin et Flora.
R. Zelazny, Les 9 Princes d'Ambre

1.1 ScaLAPACK et ses composants

La performance est cruciale pour les calculs scientifiques. Le parallélisme est un moyen d'obtenir cette performance mais souvent au prix d'une longue et difficile implémentation. L'usage de bibliothèques telles que ScaLAPACK facilite la programmation. Encore faut-il en maîtriser tous ses composants.

L'organisation de ScaLAPACK ("Scalable Linear Algebra Package") et de ses composants est donnée dans la figure 1.1. La principale caractéristique de ces bibliothèques est qu'elles sont parfaitement portables mais restent performantes quelque soit la machine. Elles sont donc des candidats intéressants pour paralléliser une application quelque soit le paradigme de programmation employé. Elles proposent entre autre des routines parallèles effectuant la multiplication matricielle, la factorisation LU ou encore le calcul de valeurs propres... Voyons plus en détail chaque composant.

La bibliothèque de communication se dénomme BLACS ("Basic Linear Algebra Communication Subroutines" : [26]). C'est un ensemble de primitives de communication permettant d'envoyer des matrices carrées ou trapézoïdales sur une grille bidimensionnelle de processeurs. Il est possible d'effectuer des envois point à point, des diffusions, ou encore des opérations globales comme une somme sur plusieurs processeurs. Des versions de cette bibliothèque existent pour MPI et PVM mais aussi pour des machines particulières telles que l'IBM SP2 ou l'Intel Paragon... Le surcoût d'utilisation est pratiquement nul ce qui fait des BLACS une bibliothèque performante, portable et parfaitement adaptée au calcul matriciel intensif.

La première bibliothèque de calcul séquentiel se dénomme BLAS ("Basic Linear Algebra

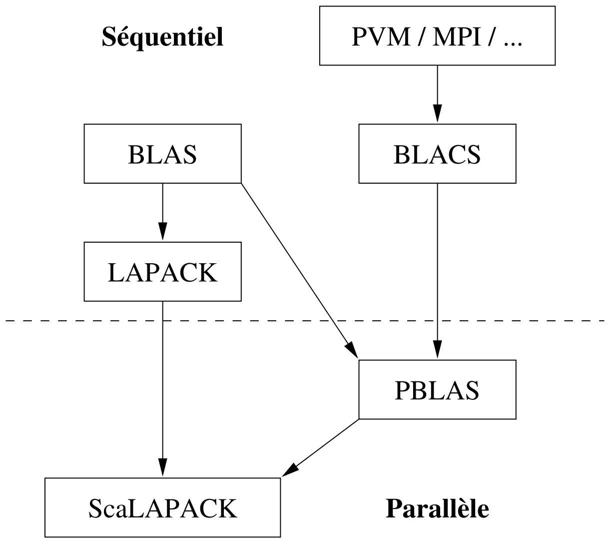

FIG. 1.1 - *Organisation de ScaLAPACK.*

Subroutines"). C'est un ensemble de routines de calcul réparties en trois niveaux. Le premier niveau a été défini dans [41]. Il effectue des opérations basiques sur les vecteurs. Par exemple, `_DOT` effectue le produit scalaire de deux vecteurs ($s \leftarrow x^T y$), `_SCAL` fait le produit d'un réel et d'un vecteur ($x \leftarrow \alpha x$). Le deuxième niveau, défini dans [25], effectue des opération matrices-vecteurs. Par exemple, `_GEMV` fait le produit matrice-vecteur ($y \leftarrow \beta y + \alpha Ax$). Le troisième niveau, défini dans [24], effectue des opérations matrices-matrices. Ces routines ont un bon ratio entre nombre de calculs et accès mémoire ($O(n^3)/O(n^2)$) et sont donc plus performantes que les BLAS 2. Par exemple `_TRSM` résout un système triangulaire ($B \leftarrow \alpha A^{-1}B$). Toutes ces routines sont implémentées pour des calculs en simple et double précision, pour des réels et des complexes. Les codes source de ces routines sont en Fortran 77 et peuvent donc être compilés sur n'importe quelle machine. Cependant, de nombreux constructeurs de machines (Sun, Intel, IBM...) fournissent des versions optimisées.

La deuxième bibliothèque de calcul séquentiel se dénomme LAPACK ("Linear Algebra Package" : [2]). Elle propose un nombre impressionnant de routines de calcul séquentiel implémentant des algorithmes d'algèbre linéaire classiques (calcul de valeurs propres, résolution de systèmes, factorisation LU, QR ...) pour différents types de matrices (bandes, pleines, symétriques, triangulaires...). Elle utilise intensivement les BLAS.

Les PBLAS ("Parallel Basic Linear Algebra Communication Subroutines" : [37]) sont la version parallèle des BLAS. Ces routines sont implémentées en passage de message et utilisent comme base les BLAS pour le calcul et les BLACS pour les communications. Leur utilisation nécessite un peu plus d'expertise que pour la version séquentiel puisqu'il faut savoir utiliser les BLACS et comprendre les principes de programmation utilisés dans cette bibliothèque.

Un des points cruciaux est la distribution employée pour les matrices. La figure 1.2 donne un exemple de la distribution cyclique par blocs employée dans les PBLAS. La matrice est tout d'abord divisée en blocs dont la taille $r \times s$ doit être fixée par le programmeur. La plupart des routines utilisent des blocs carrés. Ensuite, les blocs sont répartis cycliquement entre les processeurs en se conformant à la forme de la grille de processeurs. Dans l'exemple de la figure 1.2, la matrice est de taille de $8r \times 8r$ et la grille de processeurs 2×3. Tous les blocs grisés appartiennent au processeur numéro 4 (coordonnées (1,1) dans la grille). Dans la mémoire local du processeur 4, ces blocs sont concaténés pour ne pas perdre de place mémoire et surtout avoir un gros grain lors des calculs.

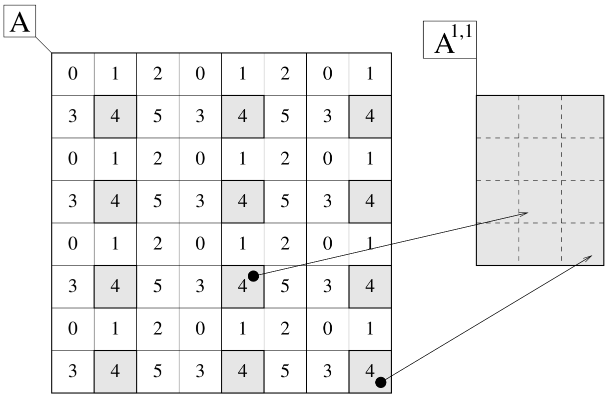

FIG. 1.2 - *Distribution cyclique par bloc d'une matrice.*

Soit (i,j) les coordonnées d'un élément de la matrice. Il se trouve sur le processeur $(\left\lfloor \frac{i}{r} \right\rceil \%P, \left\lfloor \frac{j}{s} \right\rceil \%Q)$, aux coordonnées $(\left\lfloor \frac{i}{Pr} \right\rceil .r + (i\%r), \left\lfloor \frac{j}{Qs} \right\rceil .s + (j\%Q))$ dans la matrice locale du processeur.

L'avantage d'une telle distribution est qu'elle permet de modifier l'équilibrage de charge en faisant varier la forme de la grille et la taille de bloc. On peut également jouer sur la forme de la grille pour retrouver des distributions plus standards (par colonne cyclique, par bloc, par ligne de blocs...). L'inconvénient est la complexité de développer une routine utilisant ce type de distribution. De plus, elle pose à l'utilisateur les problèmes du choix de la taille de bloc et de la forme de la grille.

Enfin, ScaLAPACK ("Scalable Linear Algebra Package": [5]) contient la version parallèle de plusieurs routines de LAPACK, notamment LU, Choleski, QR... L'implémentation est basée sur les PBLAS et LAPACK. Toutes ces routines utilisent également la distribution cyclique par blocs.

De ce tour d'horizon et de l'expérience acquise au cous de cette thèse, nous retenons 4 points essentiels sur l'utilisation de ScaLAPACK et de ses composants.

- La portabilité de ces bibliothèques est très appréciable quand il s'agit de tester un code sur différentes machines. Les BLACS ont un rôle essentiel pour obtenir cette portabilité puisque ce sont les primitives de communication qui varient le plus d'une machine à l'autre (les routines de calcul peuvent toujours être compilées même si cela limite les performances). L'ensemble des routines proposé est donc suffisamment réduit pour fonctionner sur n'importe quelle architecture. De plus, dans le cas d'une architecture utilisant PVM ou MPI, les BLACS masquent en partie les problèmes liés au démarrage des processus. Malheureusement, il reste de nombreuses routines à appeler pour initialiser la grille virtuelle, ce qui peut poser problème au néophyte. Dans certains cas, il est possible de gérer ces inconvénients de manière transparente à l'utilisateur. Dans le chapitre 4 par exemple, nous avons implémenté quelques routines qui gèrent automatiquement l'initialisation et les contextes BLACS lors de l'exécution.

- La portabilité est également assurée par l'inexistence de routines BLACS asynchrones. Ceci constitue un inconvénient majeur lorsque l'on veut optimiser un code comme nous l'avons fait dans les chapitres 2 et 3.

- La performance des routines de ScaLAPACK vient essentiellement de l'emploi des routines PBLAS. Elles permettent d'implémenter la majeure partie des algorithmes d'algèbre linéaire classiques. Malheureusement, leur petit nombre en fait parfois des routines trop "générales". Pour obtenir une meilleure performance d'un code, il faut implémenter de nouvelles routines comme nous l'avons fait dans le chapitre 5.

- La plus grosse difficulté de ScaLAPACK vient de la distribution cyclique par blocs des matrices. Que ce soit pour implémenter une nouvelle routine ScaLAPACK ou simplement utiliser cette bibliothèque, ce type de distribution impose de nombreuses contraintes. Dans ce dernier cas, l'utilisateur doit gérer lui-même la distribution de la matrice, fixer la taille des blocs, la taille et la forme de la grille de processeurs. Un mauvais choix pour la valeur de ces paramètres entraîne une perte de performance notable. Le calcul automatique de ces paramètres est donc une amélioration primordiale pour simplifier l'utilisation de ScaLAPACK. Ce sujet est abordé dans les chapitres 3 et 4.

1.2 Contexte

Dans cette thèse, nous utilisons à plusieurs reprises des modèles théoriques ou des notions propres au parallélisme. Nous en donnons un aperçu afin de clarifier par avance certaines affirmations.

Les communications

Pour la plupart des chapitres, nous avons utilisé la bibliothèque BLACS afin d'effectuer les communications. Les routines proposées sont localement bloquantes. Cela signifie qu'un processeur attend que le récepteur ait reçu son message avant de continuer son exécution. Par contre, lors d'une communication globale (diffusion, somme...), chaque processeur n'attend pas la fin de la communication globale. C'est pourquoi on parle de localement bloquant.

Dans les chapitres 2 et 3, nous utilisons des communications asynchrones pour optimiser des codes. Ce type de communication n'existe pas dans les BLACS mais on le trouve dans MPI ou dans les primitives de communication des machines parallèles à mémoire distribuée. Les communications asynchrones permettent d'envoyer un message et de redonner la main au processeur sans attendre que le récepteur ait reçu le message. Il faut donc tester dans la suite du programme si le message est bien arrivé, sinon on risque d'être confronté à des permutations de messages, de problèmes de tampons mémoire ou encore des erreurs sur les identificateurs des messages. Leur utilisation est donc largement plus délicate que pour les BLACS mais permet le recouvrement calcul/communication, technique d'optimisation propre au machine parallèles (voir ci-dessous).

En ce qui concerne la modélisation théorique des communications, nous avons employé le modèle en $\beta + L\tau$, où τ est la bande passante du réseau d'interconnexion, et β la latence. Le temps d'envoyer un message de taille L d'un processeur à un autre est donc $T = \beta + L\tau$. Ce modèle simple ne prend pas en compte la distance entre les processeurs, ni l'architecture du réseau, mais il s'est montré suffisamment précis pour tous les résultats théoriques produits dans cette thèse.

Nous ne parlons pas ici des différentes topologies de réseaux existantes, ni des techniques de routage car elles n'ont pas d'impact important sur les résultats présentés. Pour plus de détails sur ces sujet, le lecteur pourra se référer à l'ouvrage [33].

Le pipeline et le recouvrement calcul/communication

Ces deux techniques servent à optimiser un code parallèle quand on ne peut obtenir aucun gain sur le temps de calcul. Le pipeline peut être parfois employé quand des calculs effectués par différents processeurs sont dépendants. Par exemple, un processeur attend qu'on lui communique le résultat d'un calcul avant de continuer son exécution en utilisant ce résultat. Ce cas de figure arrive fréquemment dans des codes implémentés en parallélisme de tâche (cf. chapitre 2) mais également dans des codes de calculs scientifiques implémentés en parallélisme par passage de messages (cf. chapitre 3).

Le pipeline consiste à découper les tâches de calculs en sous-tâches. Au lieu d'attendre la fin de la tâche pour communiquer le résultat global, chaque sous-tâche envoie son résultat partiel. Le processeur destinataire peut donc commencer ses calculs plus tôt. Bien entendu, cette technique ajoute un paramètre crucial pour l'efficacité du code : le nombre de sous-tâches. Ce nombre dépend énormément des calculs que l'on pipeline et de la machine sur laquelle le code est exécuté. Nous proposons dans le chapitre 2 une méthode pour calculer automatiquement ce nombre mais d'autre techniques (calculatoires, évaluation dynamique...) peuvent être trouvé dans [18, 50]. Des résultats sur le pipeline de communications peuvent également être trouvés dans [29].

Le recouvrement calcul/communication repose sur l'utilisation des communications asynchrones. La technique consiste à utiliser un temps de calcul pour envoyer un message et vérifier après le calcul si le message a été bien reçu. C'est similaire à un accès direct à la mémoire (DMA) pendant que le processeur travaille. L'avantage est bien entendu de masquer une partie voire toute la communication, ce qui est très important dans un code parallèle où les communications représentent une part non négligeable du temps d'exécution. Malheureusement, les dépendances entre calculs empêchent souvent ce type d'optimisation. Cependant, dans le cas d'un pipeline, on peut avantageusement tirer parti de ce type d'optimisation puisque l'on découpe les calculs en sous-tâches indépendantes.

Pour conclure, nous donnons également au lecteur quelques références sur l'algorithmique parallèle générale ([40, 28, 11]).

CHAPITRE 2

BLAS 3 pipelinés

"Sans le mélange pour catalyser la prescience linéaire des Navigateurs de la Guilde, les voyageurs de l'espace ne peuvent franchir les parsecs qu'à la vitesse d'un escargot"
F. Herbert, L'Empereur-Dieu de Dune.

Il existe de nombreuses recherches sur l'optimisation des BLAS 3, notamment en utilisant des techniques de partitionnement de boucles et une bonne gestion de la hiérarchie mémoire [12, 30, 38]. On peut citer PHiPAC [4] et ATLAS [59] qui proposent le calcul automatique du meilleur découpage en bloc d'un produit matriciel pour une architecture donnée. Ces travaux concernent essentiellement les codes séquentiels contenant des appels aux BLAS 3 et malheureusement, dans le cas d'un code déjà parallèle, il est souvent difficile d'optimiser les performances. Ou bien on trouve un algorithme parallèle plus performant, ou bien on travaille sur le code existant en utilisant des techniques d'optimisations propres aux machines parallèles. Les communications asynchrones se rangent dans la seconde catégorie. Elles permettent à un processeur de travailler pendant qu'il communique, réduisant ainsi le surcoût des communications. Malheureusement, les dépendances entre communications et calculs limitent souvent le nombre de communications pouvant être recouvertes. Le pipeline est également une technique d'optimisation fréquemment utilisée. A l'inverse des communications asynchrones, les dépendances entre les calculs et les communications favorisent la mise en place d'un pipeline. Cela consiste à découper une tâche en sous-tâches et à communiquer le résultat produit par celles-ci, dès qu'il est disponible, au processeur destinataire. Ce dernier peut donc commencer ses calculs plus tôt. Cette technique pose cependant un problème crucial, combien doit-il y avoir de sous-tâches pour que le pipeline soit efficace ? La réponse dépend de la machine, de la bande passante et de la latence du réseau de communication et surtout de l'algorithme qui est pipeliné. Quelque fois, ce nombre peut être calculé ou approximé pour obtenir des performances optimales.

Notre but est de mixer ces deux techniques d'optimisation dans le cadre d'un programme parallèle utilisant des BLAS de niveau 3. Chaque BLAS 3 est considéré comme une tâche exécutée sur un processeur particulier. Les dépendances de données entre tâches forment un graphe qui va nous servir à déterminer automatiquement la meilleur taille de découpage pour la tâche lors du pipeline. Nos recherches prennent uniquement en compte un graphe linéaire.

Ce chapitre est organisé en 4 sections. Premièrement, nous donnons un exemple typique pour découvrir quels sont les problèmes posés par notre objectif. Ensuite, nous présentons une méthodologie pour le découpage des tâches. Nous continuons en donnant un modèle théorique du temps d'exécution d'un BLAS de niveau 3 que nous utilisons pour calculer la taille de découpage optimale. Nous donnons enfin des résultats expérimentaux sur Intel Paragon et IBM SP2.

2.1 Exemple typique

Notre exemple est un graphe de tâche, constitué par 5 produits matriciels exécutés sur 5 processeurs différents. Les dépendances entre tâches sont données dans la figure 2.1. La première flèche indique que la matrice C, après calcul, est envoyée à la tâche 2, qui s'en sert pour son produit matriciel. C'est donc une dépendance de flot de données.

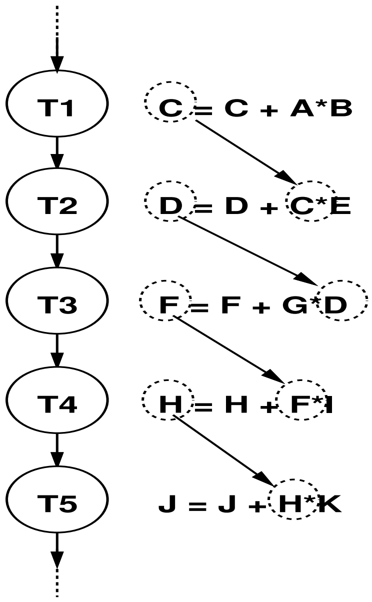

FIG. 2.1 - *Exemple typique : pipeline de 5 produits matriciels.*

Cet exemple est très simple mais il permet de constater tous les problèmes liés à la mise en place d'un pipeline optimal. Nous avons choisi le produit matriciel (`DGEMM`) comme base car il est très utilisé dans les noyaux de calcul d'algèbre linéaire et qu'il y a de multiples

Do $i = 0,\ M$	boucle i
Do $j = 0,\ N$	boucle j
Do $h = 0,\ K$	boucle h
$C_{ij} = C_{ij} + A_{ih}B_{hj}$	S_1
Enddo	
Enddo	
Enddo	

TAB. 2.1 - *Algorithme de multiplication matricielle($C = C + AB$).*

façons de le découper en plusieurs sous-produits. De plus, les autres BLAS 3 peuvent être réécrits à l'aide de cette routine [38].

Le premier point important est qu'il est impossible d'utiliser les communications asynchrones sans découper les tâches. Les dépendances du graphe ne le permettent pas. De plus, le pipeline est pratiquement obligatoire si l'on veut garder un code parallèle. En effet, le temps d'exécution parallèle sans optimisation est plus important (surcoût des communications) que le temps d'exécution séquentiel.

Deuxièmement, un produit matriciel est constitué de trois boucles imbriquées. Nous allons voir que la façon de partitionner ces boucles va déterminer l'efficacité du pipeline. Le tableau 2.1 donne l'algorithme de multiplication matricielle ($C = C + AB$), où les matrice A sont de taille $M \times K$, B de taille $K \times N$ et C de taille $M \times N$. Le nombre d'opérations flottantes est en $O(MNK)$.

2.1.1 Comment pipeliner des tâches DGEMM

Il y a plusieurs façons de découper les calculs faits par DGEMM en sous-tâches pour les pipeliner. Cela est équivalent au "strip-mining" ou au partitionnement de boucles présentés dans [49, 60]. Pour un BLAS 3, découper en sous-tâches correspond en fait à découper une ou plusieurs matrices que le BLAS 3 utilise en sous-matrices. La figure 2.2 donne trois possibilités de découpage des matrices dans le cas de la multiplication. Dans le premier et le deuxième cas, on partitionne une seule des matrices horizontalement ou verticalement afin de calculer des sous-matrices horizontales ou verticales. Ce cas de figure est idéal pour un pipeline puisqu'un processeur peut recevoir, calculer et envoyer des blocs rectangulaires. Le troisième cas est plus problématique puisque l'on calcule des blocs carrés qui sont des facettes d'une même matrice. Il vaut mieux éviter cette situation. Cependant, nous avons choisi les dépendances de notre exemple afin de tomber obligatoirement dans ce cas et montrer les différents moyens pour contourner le problème.

/* Solution (a) */	/* Solution (b) */	/* Solution (c) */
Do $i_s = 0$, M/L	*Do* $j_s = 0$, N/L	*Do* $k_s = 0$, K/L
Do $i = i_s * s$, $(i_s + 1) * s - 1$	*Do* $i = 0$, M	*Do* $i = 0$, M
Do $j = 0$, N	*Do* $j = j_s * s$, $(j_s + 1) * s - 1$	*Do* $j = 0$, N
Do $h = 0$, K	*Do* $h = 0$, K	*Do* $k = k_s * s$, $(k_s + 1) * s - 1$
$C_{ij} = C_{ij} + A_{ih} B_{hj}$	$C_{ij} = C_{ij} + A_{ih} B_{hj}$	$C_{ij} = C_{ij} + A_{ih} B_{hj}$
Enddo	*Enddo*	*Enddo*
Enddo	*Enddo*	*Enddo*
Enddo	*Enddo*	*Enddo*
Enddo	*Enddo*	*Enddo*

TAB. 2.2 - *Algorithmes de la multiplication matricielle pour trois différents découpages des calculs.*

Le tableau 2.2 donne les algorithmes correspondant, en supposant que les matrices sont découpées en L blocs. On peut bien entendu partitionner et/ou interchanger n'importe quelle boucle mais cela implique toujours le calcul et la communication de sous-matrices dont la forme est totalement inefficace pour le pipeline. Le troisième cas en est un exemple.

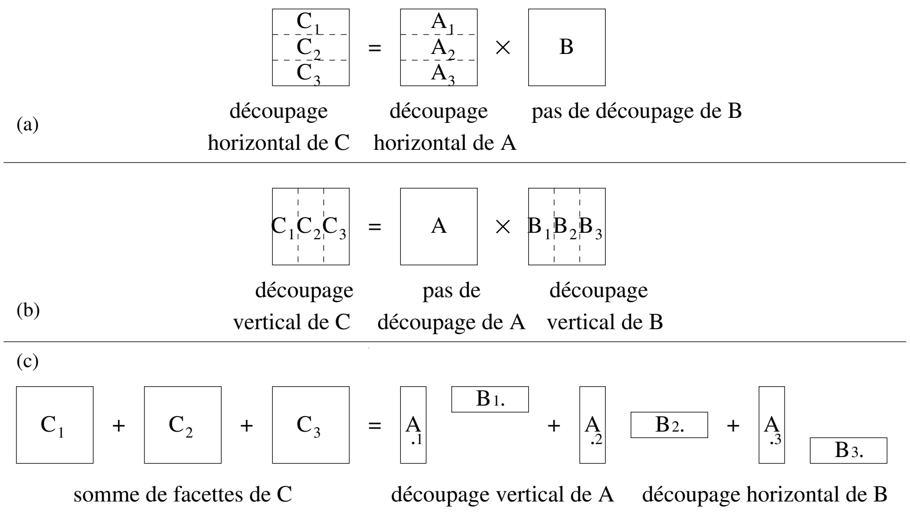

FIG. 2.2 - *Trois façons de découper un produit matriciel.*

2.1.2 Choisir un découpage pour les calculs

En tenant compte des dépendances, il faut trouver un découpage pour chaque tâche du graphe. Au lieu d'utiliser les termes découpage horizontal ou vertical, nous utilisons une notation plus intuitive. D'après le guide de référence des BLAS [24], `DGEMM` utilise trois matrices : $A_{(M \times K)}$, $B_{(K \times N)}$ and $C_{(M \times N)}$. Dans le premier cas de la figure 2.2, on remarque que

	Tâche 1	Tâche 2	Tâche 3	Tâche 4	Tâche 5
Solution 1	O/O	O/O	O/O	O/O	O/-
Solution 2	M/H	M/H	K/S	S/S	S/-
Solution 3	M/H	M/H	K/O	M/H	M/-
Solution 4	M/H	M/O	N/V	N/V	K/-

TAB. 2.3 - *Quatre solutions pour découper les tâches de l'exemple typique.*

A est en fait découpée suivant la dimension M. Dans le deuxième cas, B est découpée suivant la dimension N. Et dans le troisième cas, A et B sont découpées suivant la dimension K. Nous utilisons donc la notation : "une tâche est M-bloc (ou N-bloc, K-bloc)", pour décrire quel est le découpage des calculs utilisé pour un `DGEMM` et plus généralement celui d'un BLAS 3.

On voit également dans le troisième cas que l'on peut obtenir une matrice découpée en facettes. Puisque la matrice est la somme de ses facettes, nous utilisons la notation S-bloc pour décrire son découpage. Bien entendu, il est possible d'effectuer une multiplication matricielle avec uniquement des matrices découpées en facettes. Mais cela implique beaucoup plus que $O(MNK)$ opérations flottantes et donc doit être évité.

2.1.3 Choisir un découpage pour les communications

Dans notre exemple, nous envoyons toujours une matrice qui est le résultat d'un calcul. L'orientation (horizontale ou verticale) des blocs que l'on va envoyer à la tâche suivante est donc déterminée par le type de découpage utilisé pour le calcul. Par exemple, si la tâche 1 est M-bloc alors la tâche 2 reçoit des blocs horizontaux. Par contre, si la matrice envoyée n'est pas une matrice résultat, nous avons le choix de l'orientation.

Contrairement aux calculs, nous gardons la notion de blocs horizontaux ou verticaux pour les communications. Il y a en effet aucune dimension privilégiée lors de l'envoi d'un bloc. Nous utilisons donc les notations : "une communication est H-bloc (ou V-bloc)" en fonction de l'orientation du bloc. A cela nous ajoutons les communications S-bloc quand une matrice est envoyée par facette, et $\oslash$-bloc quand la matrice est entièrement envoyée.

2.1.4 Exemple de solution pour le découpage

Dans la table 2.3, nous donnons quatre solutions pour découper les calculs et les communications de chaque tâche, en tenant compte des dépendances du graphe. Pour chaque tâche, nous donnons à gauche le découpage des calculs et à droite le découpage des communications.

- Première solution : pas de pipeline, l'exécution est séquentielle.
- Deuxième solution : les tâches 1 et 2 sont M-bloc donc la tâche 3 reçoit des blocs horizontaux dans D. Cela implique un découpage K-bloc de la tâche 3 et un découpage S-bloc de la matrice résultat F. Les calculs suivants dans le pipeline sont également S-bloc.
- Troisième solution : elle commence comme dans la solution 2 mais la tâche 3 envoie toute la matrice F (communication $\oslash$-bloc) au lieu de faire une communication S-bloc. Ensuite, la tâche 4 commence un "nouveau" pipeline avec des calculs M-bloc.
- Quatrième solution : cette fois, c'est la tâche 2 qui envoie sa matrice résultat D en entier. La tâche 3 commence donc un "nouveau" pipeline avec des calculs N-bloc.

Prenons pour hypothèse que chaque tâche nécessite 3 secondes pour s'exécuter et que nous les divisons en 3 sous-tâches. Nous supposons également que chaque sous-tâche prend une seconde pour s'exécuter, ce qui n'est pas réellement le cas quand on découpe un BLAS 3 (cf. section 2.4.2). La figure 2.3 présente une sorte de diagramme de Gantt de l'exécution du pipeline pour chaque solution. Les rectangles représentent une tâche ou une sous-tâche et les lignes, les communications.

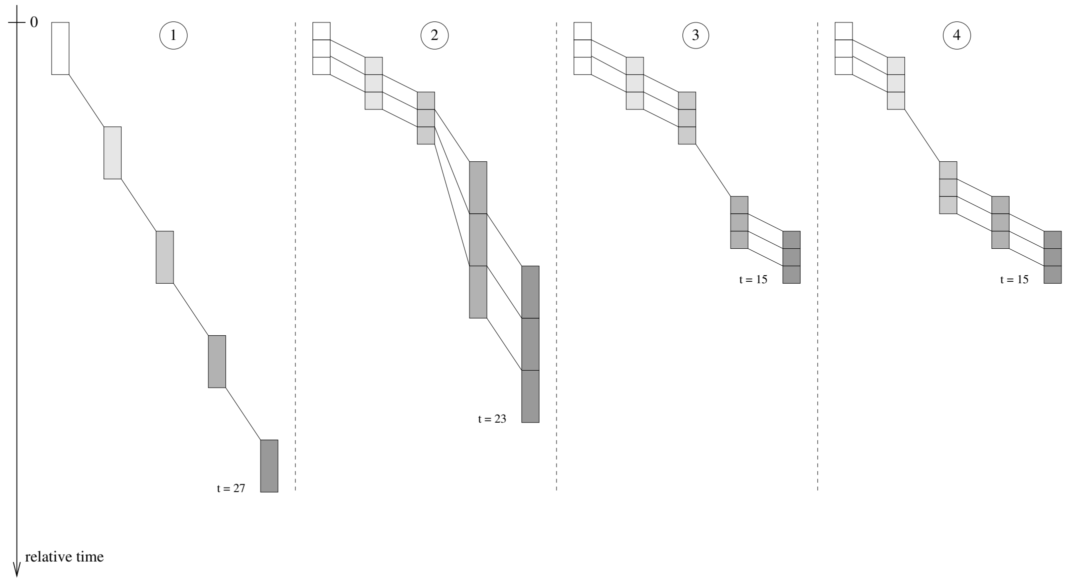

FIG. 2.3 - *Exemple typique : diagramme de Gantt pour les quatre solutions de découpage.*

- Dans la première colonne, chaque tâche finit son calcul avant d'envoyer le résultat à la tâche suivante. C'est donc la solution sans pipeline. Le temps total d'exécution est de 27 secondes.

/* Task 1 ($C = C + AB$) */	/* Task 2 ($D = D + CE$) */
Do $i_L = 0$, M/L	*Do* $i_L = 0$, M/L
	recv(sous-matrice C)
Do $i = i_L * L$, $(i_L + 1) * L - 1$	*Do* $i = i_L * L$, $(i_L + 1) * L - 1$
Do $j = 0$, N	*Do* $j = 0$, N
Do $h = 0$, K	*Do* $h = 0$, K
$C_{ij} = C_{ij} + A_{ih}B_{hj}$	$D_{ij} = D_{ij} + C_{ih}E_{hj}$
Enddo	*Enddo*
Enddo	*Enddo*
Enddo	*Enddo*
send(sous-matrice C)	
Enddo	*Enddo*
	send(matrix D)

TAB. 2.4 - *Solution 4: codes pour les tâches 1 et 2.*

- La seconde colonne correspond à la deuxième solution. On voit bien la perte due aux calculs S-bloc. La tâche 4 calcule $F.I + H \rightarrow H$. Si l'on découpe F en 3 blocs, alors mathématiquement, la tâche 4 effectue $(F_1 + F_2 + F_3).I + H \rightarrow H$, ce qui prend $3n^2 + n^3$ opérations flottantes. Mais les matrices F_i sont reçues séquentiellement (communications S-bloc). La tâche 4 calcule donc $F_1.I + F_2.I + F_3.I + H \rightarrow H$, ce qui prend $3n^2 + 3n^3$ opérations flottantes. Il en va de même pour la tâche 5, ce qui conduit à un temps total d'exécution de 23 secondes. C'est mieux que la version sans pipeline mais plus qu'une exécution séquentielle sur un seul processeur (15 secondes).
- Pour les deux dernières solutions, on obtient un gain proche de 2 par rapport à la solution sans pipeline en dépit du fait que le graphe est divisé en deux pipelines. Cependant, on ne fait pas mieux qu'une exécution séquentielle sur un seul processeur. Comme on le verra par la suite, il est possible d'obtenir un temps inférieur en découpant les tâches plus finement.

Les tableaux 2.4 et 2.5 donnent le code de chaque tâche de la solution 4. On suppose que les tâches sont découpées en L blocs. Dans chaque cas, on remarque que les trois boucles les plus internes correspondent à un produit matriciel. Elles peuvent donc être remplacées par un appel à la routine BLAS 3 `DGEMM`.

2.1.5 Influence du temps d'exécution de chaque tâche

Prenons maintenant pour hypothèse que les tâches 3 et 4 prennent 6 secondes pour s'exécuter. On garde le découpage en 3 sous-tâches. La figure 2.4 donne une comparaison du

`/* Task 3` $(F = F + GD)$ `*/`	`/* task 4` $(H = H + FI)$ `*/`	`/* task 5` $(J = J + HK)$ `*/`
`recv(matrice` D`)`		
Do $j_L = 0,\ N/L$	*Do* $j_L = 0,\ N/L$	*Do* $k_L = 0,\ K/L$
	`recv(sous-matrice` F`)`	`recv(sous-matrice` H`)`
Do $i = 0,\ M$	*Do* $i = 0,\ M$	*Do* $i = 0,\ M$
Do $j = j_L * L,\ (j_L + 1) * L - 1$	*Do* $j = j_L * L,\ (j_L + 1) * L - 1$	*Do* $j = 0,\ N$
Do $h = 0,\ K$	*Do* $h = 0,\ K$	*Do* $k = k_L * L,\ (k_L + 1) * L - 1$
$F_{ij} = F_{ij} + G_{ih}D_{hj}$	$H_{ij} = H_{ij} + F_{ih}I_{hj}$	$J_{ij} = J_{ij} + H_{ih}K_{hj}$
Enddo	*Enddo*	*Enddo*
Enddo	*Enddo*	*Enddo*
Enddo	*Enddo*	*Enddo*
`send (sous-matrice` F`)`	`send (sous-matrice` H`)`	
Enddo	*Enddo*	*Enddo*

TAB. 2.5 - *Solution 4: codes pour les tâches 3, 4 et 5.*

diagramme de Gantt des solutions 3 et 4. Dans ce cas, la solution 4 est largement plus intéressante que la 3. De plus, on atteint cette fois un temps d'exécution inférieur (20 secondes) à celui d'une exécution séquentielle sur un seul processeur (21 secondes).

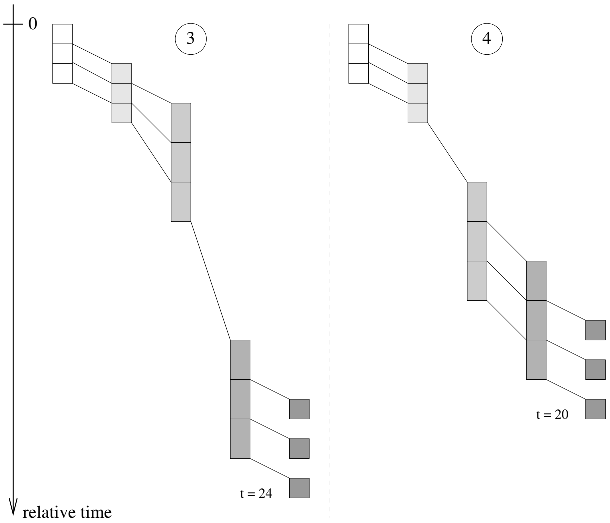

FIG. 2.4 - *Comparaison entre les solutions 3 et 4.*

La durée totale du pipeline dépend donc grandement du type de BLAS que chaque tâche exécute, de la taille des matrices qu'il utilise et bien entendu de la taille de découpage.

2.1.6 Conclusion

Nous avons présenté uniquement 4 solutions mais beaucoup d'autres existent. Il est possible de construire un arbre avec toutes les possibilités mais la plupart sont inutilisables.

Comme la deuxième solution, elles impliquent un pipeline totalement inefficace. On a vu également que les “bonnes” solutions peuvent être départagées uniquement en calculant le temps total du pipeline qu'elles impliquent.

Pour résoudre notre problème, il faut donc construire un ensemble minimal des “bonnes” solutions, trouver la ou les tailles de découpage optimales des pipelines et enfin calculer le temps total de chaque solution pour déterminer quelle est la meilleure.

2.2 Cadre de travail pour le pipeline de BLAS 3

2.2.1 Le graphe de tâche

Rappelons d'abord les hypothèses de départ de notre travail :

- Nous considérons uniquement un graphe de tâche linéaire constitué par des BLAS 3. Cela implique qu'une tâche ne peut envoyer qu'une seule matrice à une seule autre tâche.
- Chaque tâche peut être divisée en sous-tâches de calculs indépendants.
- Les communications asynchrones sont utilisées pour envoyer le résultat de la sous-tâche précédente pendant que la sous-tâche courante calcule.

Le graphe de dépendance entre tâches peut être représenté comme sur la figure 2.5. A, B et C sont les matrices utilisées par les BLAS 3 pour le calcul. B' et C' sont les matrices où est stocké le résultat du calcul. Du point de vue mémoire, B et B' (ou C et C') sont identiques. Du point de vue du pipeline, il peut exister un graphe où les matrices B ou C sont envoyées avant qu'elles ne soient écrasées par le résultat. Les opérations effectuées par le graphe de la figure 2.5 sont :

- La tâche 1 exécute un `DGEMM` et envoie la matrice résultat C' à la tâche 2.
- La tâche 2 reçoit C' dans sa propre matrice B puis calcule un `DTRSM`. Enfin, elle envoie la matrice B' à la tâche 3.
- La tâche 3 reçoit B' dans sa propre matrice A et exécute un `DGEMM`.

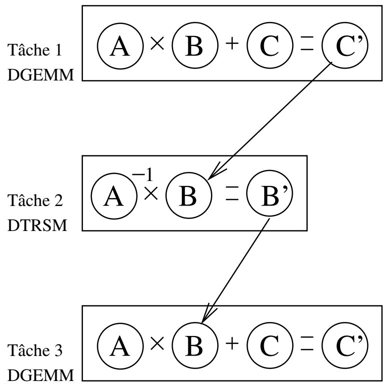

FIG. 2.5 - *Exemple 2: Trois BLAS de niveau 3 enchaînés.*

2.2.2 Considérations sur les BLAS de niveau 3

Nous avons donné dans la section précédente l'ensemble possible des découpages pour les calculs et les communications quand les tâches sont des `DGEMM`. Ainsi, le calcul d'un `DGEMM` peut être M-bloc, N-bloc, K-bloc, ou encore S-bloc.

De part la syntaxe des BLAS 3, il existe d'autres contraintes sur le choix du découpage. Par exemple `DSYMM` calcule $\alpha A.B + \beta C \rightarrow C$ ou bien $\alpha B.A + \beta C \rightarrow C$ avec A matrice symétrique. A ne peut donc pas être découpée. Le calcul d'un `DSYMM` ne peut donc être K-bloc. De plus, suivant la position de B par rapport à A, le calcul est soit M-bloc, soit N-bloc.

Un autre point important est la possibilité de transposer les matrices. Par exemple, un `DGEMM` peut calculer $\alpha A^T.B + \beta C \rightarrow C$. Du point de vue mémoire, c'est toujours la matrice non transposée qui est stockée. C'est la routine de calcul qui s'occupe d'accéder correctement à la mémoire. Si A^T est une matrice de taille $M \times K$, alors, en mémoire, elle est de taille $K \times M$. Du point de vue du pipeline, on considère que c'est A^T qui est découpée suivant sa dimension M. Le calcul est donc M-bloc.

Ces deux remarques peuvent parfois se combiner. Par exemple, `DSYR2K` calcule $\alpha A.B^T + \bar{\alpha} B.A^T + \beta C \rightarrow C$ ou bien $\alpha A^T.B + \bar{\alpha} B^T.A + \beta C \rightarrow C$. Dans le premier cas, les matrices A et B sont de taille $N \times K$ et C de taille $N \times N$. Le calcul ne peut donc être que K-bloc. Dans le deuxième cas, A et B sont de taille $K \times N$ mais elles sont tranposées en tant que membre gauche de la multiplication. On se retrouve dans le premier cas et le calcul ne peut être que K-bloc.

Le tableau 2.6 donne les possibilités de découpage (excepté S-bloc) de chaque opération BLAS 3 . La première colonne donne le nom de la routine BLAS 3. Vient ensuite l'opération

Nom	Opération	A	B	C	M-bloc	N-bloc	K-bloc
_GEMM	$\alpha A.B + \beta C \to C$	$M \times K$	$K \times N$	$M \times N$	$\times$	$\times$	$\times$
_SYMM	$\alpha A.B + \beta C \to C$	$M \times M$	$M \times N$	$M \times N$		$\times$	
_SYMM	$\alpha B.A + \beta C \to C$	$N \times N$	$M \times N$	$M \times N$	$\times$		
_SYRK	$\alpha A.A^T + \beta C \to C$	$N \times K$		$N \times N$			$\times$
_SYRK	$\alpha A^T.A + \beta C \to C$	$K \times N$		$N \times N$			$\times$
_SYR2K	$\alpha A.B^T + \bar{\alpha} B.A^T + \beta C \to C$	$N \times K$	$N \times K$	$N \times N$			$\times$
_SYR2K	$\alpha A^T.B + \bar{\alpha} B^T.A + \beta C \to C$	$K \times N$	$K \times N$	$N \times N$			$\times$
_TRSM	$\alpha A^{-1}.B + \beta C \to C$	$M \times M$	$M \times N$			$\times$	
_TRSM	$\alpha B.A^{-1} + \beta C \to C$	$N \times N$	$M \times N$		$\times$		

TAB. 2.6 - *Solutions possibles pour découper les calculs des BLAS 3.*

mathématique calculée. Les trois colonnes suivantes donnent les dimensions des matrices. Ensuite, une croix indique si tel type de découpage est possible.

D'autres routines existent mais elles ont exactement les mêmes solutions de découpage des calculs que celles présentées dans le tableau. Par exemple, _HEMM est identique à _SYMM.

2.3 Recherche de l'ensemble des meilleurs découpages

Nous avons vu dans l'exemple typique l'influence du découpage de chaque tâche sur l'efficacité du pipeline. La plupart du temps, le découpage d'une tâche dépend du découpage de la tâche précédente à cause des dépendances de données. Dans certains cas, on a le choix, et donc plusieurs solutions sont possibles pour découper l'ensemble des tâches. Nous pouvons construire un arbre avec toutes les possibilités mais la plupart sont inutiles ou inefficaces, notamment toutes celles qui contiennent des calculs ou des communications S-bloc (cf. figure 2.3). Nous proposons donc un algorithme qui construit l'ensemble des meilleures solutions. Nous employons le terme meilleur pour signifier qu'elles conduisent à un pipeline efficace, comme les solutions 3 et 4 de l'exemple typique. L'algorithme repose sur deux heuristiques que dicte l'exemple typique.

Dans la seconde solution donnée en figure 2.3, nous voyons que les calculs S-bloc sont propagés et qu'ils réduisent grandement l'efficacité du pipeline. Dans la solution 3, nous remplaçons la première communication S-bloc en une communication $\oslash$-bloc. Nous voyons que le pipeline est alors beaucoup plus efficace.

Dans la solution 4, nous voyons qu'il n'y a aucun calcul K-bloc (à part le dernier, ce qui n'a aucune influence). Mathématiquement, un calcul K-bloc donne toujours une matrice résultat communiquée S-bloc (cf. figure 2.2). La démarche suivie dans cette solution consiste à éviter de provoquer un calcul K-bloc. Si ce n'est pas possible, alors on envoie une matrice $\oslash$-bloc pour commencer un nouveau pipeline. C'est pourquoi la tâche 2 de la solution 4 envoie entièrement la matrice D à la tâche 3, sinon cette dernière aurait un calcul K-bloc (cf. solution 3).

Les deux heuristiques sont donc :

- **heuristique 1** : chaque fois qu'une communication S-bloc doit être faite, la remplacer par une communication $\oslash$-bloc. Cela revient à attendre la fin du calcul et donc briser le pipeline.
- **heuristique 2** : toujours envoyer des blocs orientés de telle façon que la tâche qui les reçoit n'ait pas de calcul K-bloc. Si ce n'est pas possible envoyer la matrice entièrement.

D'après la figure 2.3, on s'aperçoit que la deuxième heuristique n'est pas obligatoire pour obtenir un découpage efficace. En effet, la solution 3 est aussi bonne que la 4. Utiliser ou ne pas utiliser cette heuristique conduit donc à des solutions différentes. Comme notre but est de construire l'ensemble des meilleures solutions, nous ne pouvons choisir arbitrairement d'utiliser l'heuristique ou non. L'algorithme construit donc un ensemble où les deux choix sont envisagés chaque fois que la deuxième heuristique intervient. Le pseudo-code de l'algorithme est donné dans le tableau 2.6. Le découpage des calculs et des communications de la i^{eme} tâche sont notés $BL_i = [comp_i; comm_i]$ (par exemple $[M; H]$).

Le choix du découpage de la première tâche pose un problème particulier. On voit dans le tableau 2.6 que toutes les routines sauf `_GEMM` n'ont qu'un seul choix de découpage. Si la première tâche n'est pas un `_GEMM`, alors le découpage est donné par le tableau. Quand il s'agit d'un `_GEMM`, nous sautons à la tâche suivante et vérifions ses possibilités de découpage. S'il y a plusieurs choix, on passe de nouveau à la tâche suivante et ainsi de suite jusqu'à ce que l'on trouve une tâche qui n'ait qu'une seule possiblité de découpage. Ensuite, on peut remonter dans le graphe, fixer le découpage de la première tâche et lancer l'algorithme pour qu'il construise l'ensemble des meilleures solutions.

La figure 2.7 donne le résultat de l'algorithme pour l'exemple typique. On part d'un calcul M-bloc et les blocs résultats sont envoyés horizontalement. Ensuite, l'ensemble des solutions se sépare en deux. En effet, la tâche 2 effectue un calcul M-bloc et donc envoie des blocs horizontaux à la tâche 3. Cela implique des calculs K-bloc. On se trouve dans le cas où la deuxième heuristique intervient. Dans la branche de gauche, elle est utilisée et cela conduit à la solution 4. On ne l'utilise pas dans la branche de droite et cela conduit à la solution 3.

2.4 Calcul de la taille de découpage optimale

Nous avons vu dans l'exemple typique qu'il est nécessaire de connaître le temps d'exécution de chaque tâche pour déterminer quelle solution de découpage est la meilleure. Nous

$nb_{sol} = 1$ /* nombre de solutions */
$Sol = \emptyset$ /* ensemble des solutions */
For i = 1 *to* $nb_{task} - 1$ *Do*
  For j = 1 *to* nb_{sol} *Do*
    Trouve BL_i d'après la description de $task_i$ et BL_{i-1}.
    If $(comm'_i = S) \parallel (comm'_i = 0)$ *then*
      Utilise heuristique 1 $\rightarrow comm'_i = 0$.
      Concatène BL_i with Sol_j.
    Else
      Concatène BL_i with Sol_j.
      If (Heuristique 2 intervient) *Then*
        /* Créer une nouvelle solution
        Utiliser Heuristique 2 */
        $cut'_i = BL_i$
        $comm'_i = 0$
        $Sol_{nb_{sol}+1} = Sol_j$.
        Concatène cut'_i avec $Sol_{nb_{sol}+1}$.
        $nb_{sol} = nb_{sol} + 1$.
      Endif
    Endif
  Endfor
Endfor

FIG. 2.6 - *Algorithme de construction des meilleures solutions.*

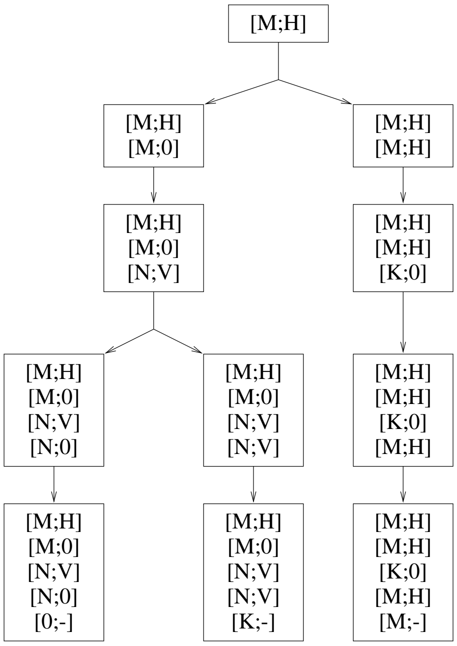

FIG. 2.7 - *Résultat de l'algorithme pour l'exemple typique.*

avons également supposé que les tâches étaient divisées en trois sous-tâches. Dans le cas des BLAS 3, la taille de découpage influe grandement sur le temps d'exécution total du pipeline. En effet, une multiplication matricielle découpée en L calculs partiels est plus longue que l'opération exécutée en une seule fois. On ne peut donc déterminer le temps d'exécution de chaque tâche et donc du pipeline que si l'on fixe la taille de découpage. Cette taille ne doit pas être trop grosse sinon les BLAS 3 perdent leur efficacité. Elle ne doit pas être trop petite sinon on perd le bénéfice du pipeline. Notre but est de calculer automatiquement la taille de découpage optimale pour chaque solution, et determiner ainsi la meilleure solution.

2.4.1 Temps d'exécution théorique d'un BLAS de niveau 3

Tous les BLAS de niveau 3 sont des routines qui effectuent des opérations matrice-matrice nécessitant $O(n^3)$ opérations flottantes et $O(n^2)$ accès mémoires. Plus précisement, le temps d'exécution dépend de la taille des matrices utilisées pour le calcul. Soient $A_{(M\times K)}$, $B_{(K\times N)}$ et $C_{(M\times N)}$ trois matrices utilisées par un BLAS de niveau 3. Alors, nous prenons pour hypothèse que le temps d'exécution de ce BLAS 3 est donné par l'expression :

$$T_{BLAS3}(M,N,K) = aMNK + bMN + cMK + dNK + eM + fN + gK + h \tag{2.1}$$

où $a\ldots h$ sont des paramètres qui dépendent de l'algorithme utilisé dans le BLAS 3 et de la machine sur lequel il est exécuté. Ces paramètres peuvent donc être trouvés par interpolations sur des tests expérimentaux. Malheureusement, nous avons remarqué que les résultats des interpolations varient suivant le protocole de test utilisé (pas d'incrémentation des taille de matrices, quelle dimension croît...). Afin, d'avoir des résultats cohérents, nous avons fixé le protocole pour notre problème spécifique, trouver la taille de découpage optimale pour le pipeline.

La première remarque concerne les différentes façons de découper les matrices d'un BLAS. Par exemple, pour `DGEMM` ($C \leftarrow \alpha AB + \beta C$), il y 3 possibilités qui sont M-bloc, N-bloc et K-bloc. Puisque dans les trois cas, les accès mémoires aux matrices sont différents (effets de cache notamment), cela conduit à des temps d'exécution différents. La deuxième remarque concerne la syntaxe des BLAS 3. Nous avons déjà signalé que les matrices pouvaient être transposées. Par exemple, pour `DGEMM`, on a quatre combinaisons possibles de transposition : $A.B$, $A^T.B$, $A.B^T$ et $A^T.B^T$. Comme l'algorithme de multiplication est différent dans chaque cas, deux calculs N-bloc peuvent avoir des temps d'exécution différents si B est transposée ou non. En résumé, cela fait 12 possibilités pour `DGEMM`. Pour `DTRSM` ($B \leftarrow A^{-1}B$), il y en a 16.

L'équation 2.1 ne tient ni compte de la dimension principale de découpage, ni de la transposition des matrices. Nous utilisons donc une nouvelle équation pour chaque possibilité. Par exemple, le temps d'exécution d'un DGEMM qui sera M-bloc dans le pipeline et dont la matrice B est transposée, est :

$$T^{M_{nt}}_{GEMM}(M,N,K) = (a_{M_{nt}}M + b_{M_{nt}})NK + (c_{M_{nt}}M + d_{M_{nt}})\sqrt{NK} + (e_{M_{nt}}M + f_{M_{nt}}) \tag{2.2}$$

L'indice $_{nt}$ indique les transpositions respectives des matrices A et B. n indique que A n'est pas transposée et t indique que B est transposée. Cette notation correspond à celle utilisée lors des appels aux BLAS 3.

Sans tenir compte des possibilités de transposition, cela donne pour les autres dimensions :

- pour un BLAS 3 qui sera N-bloc :
  $T^{N}_{BLAS3}(M,N,K) = (a_N N + b_N)MK + (c_N N + d_N)\sqrt{MK} + (e_N N + f_N)$
- pour un BLAS 3 K-bloc :
  $T^{K}_{BLAS3}(M,N,K) = (a_K K + b_K)MN + (c_K K + d_K)\sqrt{MN} + (e_K K + f_K)$

Pour chaque possibilité et chaque machine cible, nous générons un jeu de courbes que nous interpolons afin de trouver les paramètres $a \dots f$.

2.4.2 Temps d'exécution d'un BLAS de niveau 3 découpé en L blocs.

Nous avons maintenant une expression mathématique du temps d'exécution d'un BLAS 3 dans toutes les configurations possibles de découpage et de transposition des matrices. Dans le cas d'un pipeline, ce BLAS 3 n'est pas exécuté en une seule fois puisqu'il est découpé en sous-tâches de calcul. Voyons donc son temps d'exécution total lorsqu'il est réellement découpé en L calculs suivant une dimension D.

Premièrement, on fixe $1 \leq L \leq \frac{D}{2}$. Cela implique qu'un bloc ne peut être constitué d'une seule ligne ou colonne de matrice car cela équivaut à l'emploi d'un BLAS 2 et risque de ruiner l'efficacité du pipeline. Ensuite, nous voulons que les blocs aient tous une taille presque identique afin d'avoir un pipeline très régulier. Choisir une taille unique n'est pas la solution car le dernier bloc peut être soit trop gros, soit trop petit, ce qui peut nuire à l'efficacité du pipeline. Nous préférons prendre deux tailles différentes d'une seule unité. Soit $r = D \mod L$. On peut alors découper la ou les matrices en $L - r$ blocs de taille $\left\lfloor \frac{D}{L} \right\rfloor$ et r blocs de taille $\left\lceil \frac{D}{L} \right\rceil$.

Prenons par exemple $D = M$, alors le temps d'exécution d'un BLAS 3 M-bloc découpé en L blocs est:

$$\begin{aligned} T^M_{BLAS3}(M,N,K,L) &= (L-r)\Big[(a_M \left\lfloor \frac{M}{L} \right\rfloor + b_M)NK + (c_M \left\lfloor \frac{M}{L} \right\rfloor + d_M)\sqrt{NK} + (e_M \left\lfloor \frac{M}{L} \right\rfloor + f_M)\Big] + \\ & \quad r\Big[(a_M \left\lceil \frac{M}{L} \right\rceil + b_M)NK + (c_M \left\lceil \frac{M}{L} \right\rceil + d_M)\sqrt{NK} + (e_M \left\lceil \frac{M}{L} \right\rceil + f_M)\Big] \end{aligned}$$

Après simplification, cela donne:

- pour un calcul M-bloc ($M = \Big\{M_{nn}, M_{nt}, \dots\Big\}$):
  $T^M_{BLAS3}(M,N,K,L) = (a_M M + b_M L)NK + (c_M M + d_M L)\sqrt{NK} + (e_M M + f_M L)$
  $T^M_{BLAS3}(M,N,K,L) = L \times T^M_{BLAS3}(\frac{M}{L}, N, K)$
- pour un calcul N-bloc ($N = \Big\{N_{nn}, N_{nt}, \dots\Big\}$):
  $T^N_{BLAS3}(M,N,K,L) = (a_N N + b_N L)MK + (c_N N + d_N L)\sqrt{MK} + (e_N N + f_N L)$
  $T^N_{BLAS3}(M,N,K,L) = L \times T^N_{BLAS3}(M, \frac{N}{L}, K)$
- pour un calcul K-bloc ($K = \Big\{K_{nn}, K_{nt}, \dots\Big\}$):
  $T^K_{BLAS3}(M,N,K,L) = (a_K K + b_K L)MN + (c_K K + d_K L)\sqrt{MN} + (e_K K + f_K L)$
  $T^K_{BLAS3}(M,N,K,L) = L \times T^K_{BLAS3}(M, N, \frac{K}{L})$

Le temps d'exécution théorique d'un BLAS 3 découpé en L blocs est donc une fonction linéaire de L. Cependant, les résultats expérimentaux montrent que c'est partiellement vrai. La figure 2.8 donne le temps d'exécution d'un `DGEMM` M_{nn}-bloc découpé en L blocs (axe des abscisses) sur une Intel Paragon. On s'aperçoit que le temps d'exécution est linéaire par intervalles. Cette différence vient du modèle de temps d'exécution théorique d'un BLAS 3 car les paramètres $a_{M_{nt}}$, $b_{M_{nt}}$, ... sont également définis par intervalles. La figure 2.9 donne le temps d'exécution de `DGEMM` en fonction de la dimension M de la matrice A. On voit qu'il y a au moins deux intervalles de définition : $[2,11], [12,128]$. Sur ces deux intervalles, les paramètres $a_{M_{nt}}$, $b_{M_{nt}}$, ... ont donc des valeurs différentes. Cela vaut pour chaque possibilité de découpage et de transposition des matrices. Le fait qu'il y ait des intervalles est essentiellement dû à la mauvaise utilisation du cache quand les matrices sont trop petites. De même, un petit nombre d'opérations flottantes ne permet pas d'exploiter correctement le pipeline de l'unité arithmétique.

Pour la suite de notre étude, nous avons généré les paramètres des routines `DGEMM` et `DTRSM` pour les différentes possibilités de découpage et de transposition, à partir de courbes

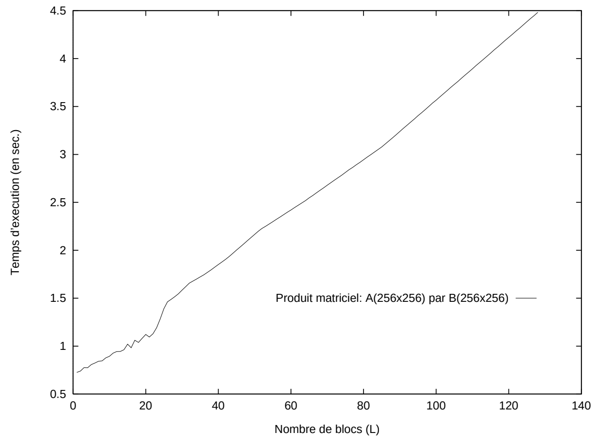

Fig. 2.8 - *Temps d'exécution d'un* DGEMM *découpé en L blocs.*

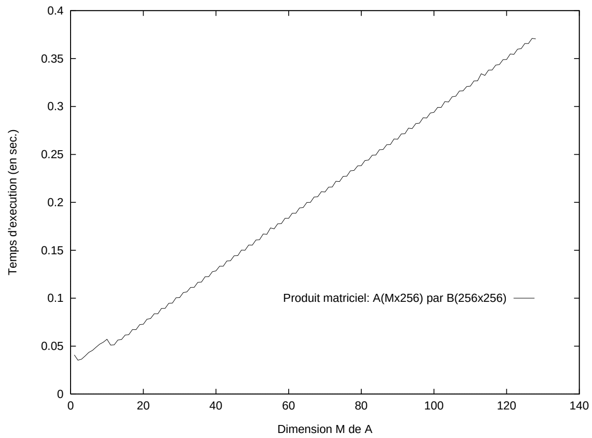

Fig. 2.9 - *Temps d'exécution de* DGEMM *en fonction de la dimension M de A.*

```
...
LstBLAS[DGEMM][M_nn].DebFin[10][0] = 36;
LstBLAS[DGEMM][M_nn].DebFin[10][1] = 39;
...
LstBLAS[DGEMM][M_nn].Coef[10][Ca] = 3.291894e-08;
LstBLAS[DGEMM][M_nn].Coef[10][Cb] = 2.798043e-06;
LstBLAS[DGEMM][M_nn].Coef[10][Cc] = 4.420302e-06;
LstBLAS[DGEMM][M_nn].Coef[10][Cd] = -3.110600e-04;
LstBLAS[DGEMM][M_nn].Coef[10][Ce] = -2.736474e-04;
LstBLAS[DGEMM][M_nn].Coef[10][Cf] = 1.593043e-02;
...
```

Tab. 2.7 - *Code C contenant les valeurs des paramètres de* DGEMM.

identiques à celle donné en figure 2.9. Les temps d'exécution bruts des routines sont donnés pour des tailles de matrices croissantes. Nous utilisons un pas unitaire d'incrémentation pour les petites tailles de matrices afin de tenir compte des effets de cache. Cela implique beaucoup d'intervalles mais augmente la précision du temps théorique d'exécution. La recherche des intervalles de définition et l'interpolation sur les courbes est calculée automatiquement par une routine développée en collaboration avec J.A. Gerber [34]. Elle produit un code C qui contient les valeurs des paramètres et les intervalles de définition. Le tableau 2.7 donne un exemple de ce qui est produit pour DGEMM. Il donne les paramètres pour un calcul M-bloc où A et B ne sont pas transposées sachant que M appartient à l'intervalle $[36, 39]$.

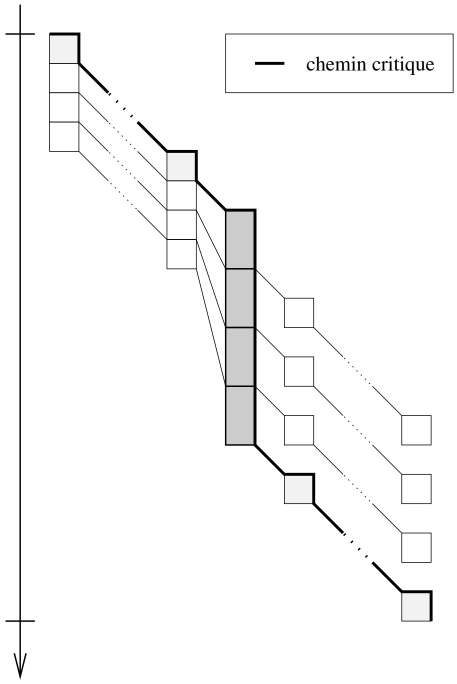

FIG. 2.10 - *Chemin critique.*

2.4.3 Temps d'exécution d'un pipeline

Le temps d'exécution d'un pipeline dépend de la stratégie de découpage de chaque tâche et de la taille de découpage (= nombre de blocs). Pour départager les meilleurs découpages, il nous faut trouver une taille de découpage optimale et ainsi déterminer le temps d'exécution du ou des pipelines. En effet, dans certains cas, le découpage est scindé en plusieurs pipelines, comme dans les solutions 3 et 4 de l'exemple typique.

Nous définissons pour cela la notion de tâche et de chemin critique. Le chemin critique est un chemin d'exécution tracé du début à la fin d'un diagramme de Gantt, qui passe par plusieurs sous-tâches, la tâche critique et plusieurs communications. Un exemple est donné dans la figure 2.10. La ligne en gras est le chemin critique. L'ensemble des blocs gris foncés forme la tâche critique. Les blocs gris clairs sont les sous-tâches traversées par le chemin critique. Nous définissons la tâche critique comme étant la tâche la plus longue du pipeline et entièrement traversée par le chemin. On suppose également que cette tâche a suffisamment de tampons pour l'envoi de messages, afin de ne jamais attendre de pouvoir envoyer entre ses sous-tâches. Pour les autres tâches, une et une seule de leur sous-tâche est traversée par le chemin. La longueur du chemin donne le temps d'exécution total du pipeline.

Il y a bien entendu de nombreux chemins possibles. La figure 2.10 donne celui que nous utilisons pour calculer la taille optimale de découpage. Il passe par les premières sous-tâches de chaque tâche avant la tâche critique, puis par la tâche critique et enfin par les dernières sous-tâches de chaque tâche. La raison d'un tel choix est la possibilité de calculer facilement la longueur de ce chemin. En effet, la tâche critique reçoit toujours les blocs pendant qu'elle calcule. Il n'y a donc aucun temps d'inactivité entre les sous-tâches, dû à un retard des données nécessaires aux calculs. Cela implique éventuellement que ses tampons de réception soient pleins, auquel cas, les tâches précédentes dans le pipeline doivent se mettre en attente de pouvoir envoyer. Il est très dur de déterminer quand ce cas de figure arrive (cela dépend du protocole de communication de la machine) et combien de temps d'inactivité cela coûte. Nous choisissons donc de faire passer le chemin par les premières sous-tâches puisque nous pouvons calculer avec certitude le moment où elles commencent. De même, les sous-tâches situées après la tâche critique sont suivies de temps d'inactivité. Au lieu de calculer la durée de ces inactivités, il est plus simple de calculer le temps pris par la dernière sous-tâche puisque nous savons calculer quand la tâche critique se termine.

Grâce au chemin critique, nous sommes en mesure de donner le temps d'exécution du pipeline en fonction de la taille de découpage. Par dérivée, on peut donc trouver la taille optimale de découpage.

Supposons qu'il y ait N tâches dans le pipeline $(t_1, \ldots, t_c, \ldots, t_N)$, avec t_c la tâche critique. Soit,

$$T_i(D_i^1, D_i^2, D_i^3) = (a_i D_i^1 + b_i) D_i^2.D_i^3 + (c_i D_i^1 + d_i)\sqrt{D_i^2.D_i^3} + (e_i D_i^1 + f_i)$$

le temps d'exécution théorique de la tâche t_i. D_i^j sont les dimensions des matrices utilisées par t_i, dont une (voire deux si $D_i^1 = K$) est découpée suivant la dimension D_i^1. Le temps de communication de E éléments est donné par $T_{com}(E) = \beta + E.\tau$ où β est la latence et τ la bande passante du réseau d'interconnexion. Soit L la taille de découpage. Alors le temps total d'exécution d'un pipeline est :

$$\begin{aligned} T_p(L) = \quad & \sum_{i=1}^{c-1}\left[\delta_{t_i} \times T_i(\tfrac{D_i^1}{L}, D_i^2, D_i^3) + T_{com}(\tfrac{D_i^1}{L} \times D_i^2)\right] + \\ & (L - 1 + \delta_{t_c}) \times T_c(\tfrac{D_c^1}{L}, D_c^2, D_c^3) + T_{com}(\tfrac{D_c^1}{L} \times D_c^2) + \\ & \sum_{i=c+1}^{N-1}\left[\delta_{t_i} \times T_i(\tfrac{D_i^1}{L}, D_i^2, D_i^3) + T_{com}(\tfrac{D_i^1}{L} \times D_i^2)\right] + \delta_{t_N} \times T_N(\tfrac{D_N^1}{L}, D_N^2, D_N^3) \end{aligned}$$

$\delta_{t_i} = 1$ si les blocs envoyés par la tâche t_i doivent être calculés avant d'être communiqués. Cela revient à dire que $\delta_{t_i} = 1$ si la tâche t_i envoie une matrice résultat et 0 si n'importe quelle autre matrice est envoyée.

Pour trouver la taille optimale de découpage, on dérive $T_p(L)$ par rapport à L :

$$
\begin{aligned}
& T'_p(L) = 0 \\
\Leftrightarrow \quad & \sum_{i=1}^{c-1} \left[\delta_{t_i} \times \frac{D_i^1}{L^2}(-a_i D_i^2 D_i^3 - c_i(D_i^2 + D_i^3) - e_i) - \tau \frac{D_i^1 \times D_i^2}{L^2} \right] + \\
& (b_c D_c^2 D_c^3 + d_c(D_c^2 + D_c^3) + f_c) + (\delta_{t_c} - 1) \times \frac{D_c^1}{L^2}(-a_c D_c^2 D_c^3 - c_c(D_c^2 + D_c^3) - e_c) - \tau \frac{D_c^1 \times D_c^2}{L^2} + \\
& \sum_{i=c+1}^{N-1} \left[\delta_{t_i} \times \frac{D_i^1}{L^2}(-a_i D_i^2 D_i^3 - c_i(D_i^2 + D_i^3) - e_i) - \tau \frac{D_i^1 \times D_i^2}{L^2} \right] + \\
& \delta_{t_N} \times \frac{D_N^1}{L^2}(-a_N D_N^2 D_N^3 - c_N(D_N^2 + D_N^3) - e_N) = 0
\end{aligned}
$$

Avec $A_i = D_i^1(a_i D_i^2 D_i^3 + c_i(D_i^2 + D_i^3) + e_i)$, $B_i = \tau \frac{D_i^1 \times D_i^2}{L^2}$ et $C = b_c D_c^2 D_c^3 + d_c(D_c^2 + D_c^3) + f_c$, on obtient :

$$
L_{opt} = \frac{1}{\sqrt{C}} \sqrt{\sum_{i=1}^{c-1} [\delta_{t_i} \times A_i + B_i] + (\delta_{t_c} - 1) \times A_c + B_c + \sum_{i=c+1}^{N-1} [\delta_{t_i} \times A_i + B_i] + \delta_{t_N} \times A_N} \tag{2.3}
$$

Nous pouvons faire quatre remarques sur ce résultat. Premièrement, L_{opt} permet d'obtenir un temps minimal d'exécution du pipeline compte tenu de notre cadre de travail (taille unique de découpage, forme des blocs...). Cette solution ne donne donc pas le temps d'exécution optimal du pipeline même si elle en est certainement proche.

Deuxièmement, il est possible que les tâches les plus longues aient le même temps d'exécution. Dans ce cas, soit on teste toutes les solutions, soit la tâche critique est celle dont la valeur de C (voir ci-dessus) est la plus grande. On obtient ainsi la valeur de L_{opt} la plus petite et on est assuré qu'une tâche n'est pas "trop" découpée ce qui pénaliserait l'efficacité du pipeline.

Troisièmement, les paramètres des modèles théoriques sont définis par intervalle pour chaque BLAS 3. Des routines différentes, avec des intervalles différents, peuvent être utilisées dans le graphe. On doit donc calculer l'intersection de ces intervalles. Cela implique également que les A_i, B_i et C et donc L_{opt} doit être calculés sur chaque intervalle donné par l'intersection. Pour chaque valeur de L_{opt}, nous testons si $\frac{D_i^1}{L_{opt}}$ est dans l'intervalle. Si plusieurs valeurs sont testées positivement, on prend la plus grande valeur de L_{opt}.

Par exemple, trois tâches sont pipelinées. $D_1^1 = 256$ et $D_2^1 = D_3^1 = 512$. Les paramètres de t_1 sont définis sur les intervalles $([2, 11], [12, \infty[)$ et ceux de t_2 et t_3 sur $(([2, 25], [26, \infty[)$. On doit donc calculer les valeurs de L_{opt} sur les intervalles $([2,11],[12,25],[26,\infty[)$. Supposons que l'on obtienne $L_{opt} = \{24, 20, 9\}$.

- $\frac{256}{24} = 10.67$ est dans $[2,11]$. $\frac{512}{24} = 21.33$ est dans $[2,25]$
- $\frac{256}{20} = 12.80$ est dans $[12,\infty[$. $\frac{512}{20} = 25.6$ n'est pas dans $[2,25]$.
- $\frac{256}{9} = 28.44$ est dans $[12,\infty[$. $\frac{512}{9} = 56.89$ est dans $[25,\infty[$.

La valeur que nous choisissons est donc $L_{opt} = 24$.

Quatrièmement, la valeur de L_{opt} n'est pas forcément la taille de découpage utilisée pour le graphe entier. On a vu dans l'exemple typique que les meilleures solutions sont constituées de deux pipelines reliés par une communication $\oslash$-bloc. Ces deux pipelines sont donc indépendants et peuvent utiliser des tailles de découpage différentes. Il faut donc calculer deux L_{opt} pour déterminer le temps d'exécution du graphe.

2.5 Résultats expérimentaux

Nous avons pris quatre exemples de graphe de tâches et testé notre méthode sur deux machines, l'Intel Paragon et l'IBM SP2. Le premier exemple est le graphe donné en figure 2.5. Le second exemple est un graphe de tâches composé de 5 `DGEMM` de taille identique. Le troisième exemple est donné en figure 2.1, c'est l'exemple typique de la première section. Il est également composé de 5 `DGEMM` mais utilisant des matrices de tailles différentes. Le dernier exemple est une multiplication matricielle parallèle utilisant le pipeline. Sa performance est comparée à la routine PBLAS `PDGEMM`.

La figure 2.11 donne une comparaison du temps d'exécution de la version pipelinée et de la non pipelinée du graphe de la figure 2.5 sur l'IBM SP2. Les temps sont donnés pour des tailles de matrices croissantes. Ce graphe comporte deux `DGEMM` et un `DTRSM`. On voit que la version pipelinée est largement plus rapide que la version non pipelinée. Notre méthode est donc valide même si le graphe n'est pas constitué de tâches identiques. Sur la figure 2.12, nous présentons le gain apporté par le pipeline. Il s'agit de la division des temps d'exécution de la figure 2.11. On voit que le gain se rapproche rapidement de 2, ce qui est excellent pour seulement 3 tâches exécutées sur 3 processeurs différents. Cela confirme l'utilité et l'efficacité du pipeline.

Nous avons bien entendu réalisé ces courbes avec une taille de découpage optimale. Le tableau 2.8 donne une comparaison de la taille théorique optimale avec l'expérimentale sur Intel Paragon. On voit que les deux tailles ne correspondent pas même si elles restent proches.

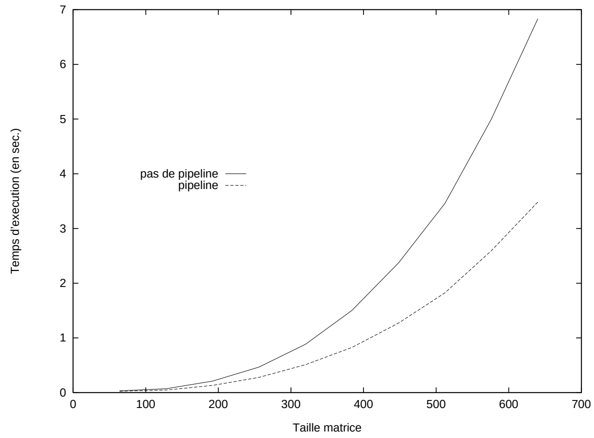

FIG. 2.11 - *Temps d'exécution du premier exemple sur IBM SP2.*

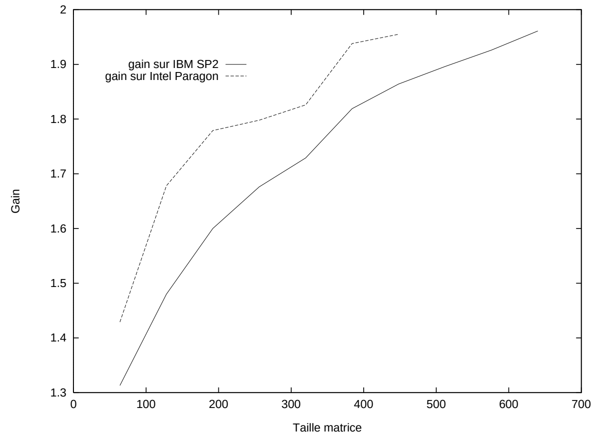

FIG. 2.12 - *Gain pour le premier exemple sur Intel Paragon et IBM SP2.*

Sur Paragon	**taille matrice**					
	64	128	192	256	320	384
Découpage optimal expérimental	6	8	6	7	8	7
Découpage optimal théorique	4	5	4	4	5	5
écart en% entre les courbes expérimentale et théorique	6,5	4,0	4,3	5,1	2,5	3,5

TAB. 2.8 - *Taille de découpage optimale pour le premier exemple sur Intel Paragon.*

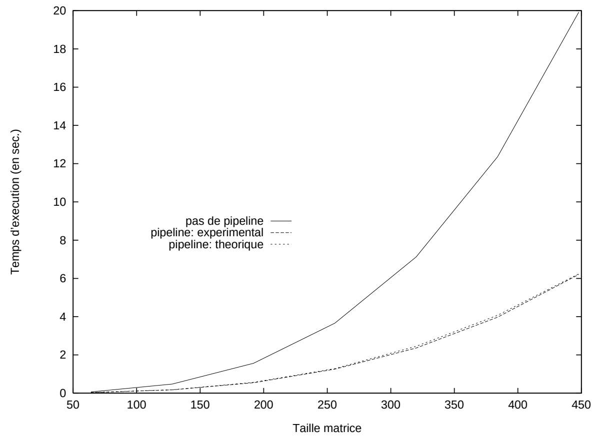

FIG. 2.13 - *Temps d'exécution du deuxième exemple sur Intel Paragon.*

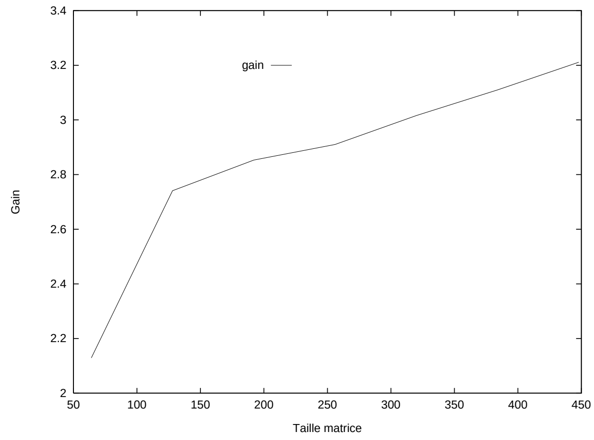

FIG. 2.14 - *Gain pour le deuxième exemple sur Intel Paragon.*

Sur Paragon	**taille matrice**						
	64	128	192	256	320	384	448
Découpage optimal expérimental	4	8	12	16	16	12	14
Découpage optimal théorique	7	9	11	12	14	15	16
Ecart en % entre les courbes expérimentale et théorique	16,7	2,6	3,5	2,3	3,6	2,4	0,6

TAB. 2.9 - *Taille de découpage optimale pour le deuxième exemple sur Intel Paragon.*

Nous avons donc exécuté le pipeline avec la taille théorique et comparé son temps d'exécution avec celui donné par la taille de découpage optimale expérimentale. Le surcoût de la courbe "théorique" est donné en pourcent dans la dernière ligne du tableau 2.8. On voit que l'écart ne dépasse jamais 7%. Cela confirme la validité de notre calcul (voir équation 2.3) et la précision des modèles théoriques des BLAS 3. A noter que la taille maximale des matrices testées n'est pas très grande mais que cela correspond pratiquement à la limite possible sur Paragon.

La figure 2.13 donne une comparaison de la version pipelinée et de la non pipelinée d'un enchaînement de 5 `DGEMM` identiques. Les temps d'exécution sont donnés pour des tailles de matrices croissantes, sur Intel Paragon. De nouveau, le pipeline réduit singulièrement le temps d'exécution du graphe. Le gain est donné en figure 2.14. Pour de petites matrices, il est en moyenne de 2,5 alors que pour de grosses matrices, il dépasse 3. Cela s'explique en partie par l'efficacité des BLAS 3 qui décroît lorsqu'il y a peu de calcul.

La table 2.9 présente le même type de résultats que le tableau 2.8. Cette fois, on remarque que les tailles de découpage théoriques correspondent un peu mieux aux expérimentales, sauf pour des matrices de taille 64. Dans ce cas, l'écart entre le temps d'exécution du pipeline avec

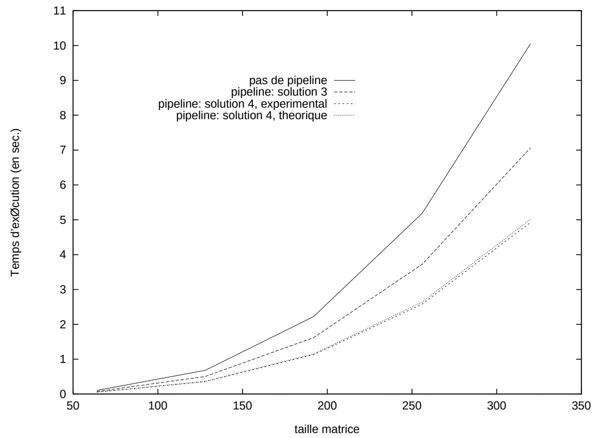

FIG. 2.15 - *Temps d'exécution du troisième exemple sur Intel Paragon.*

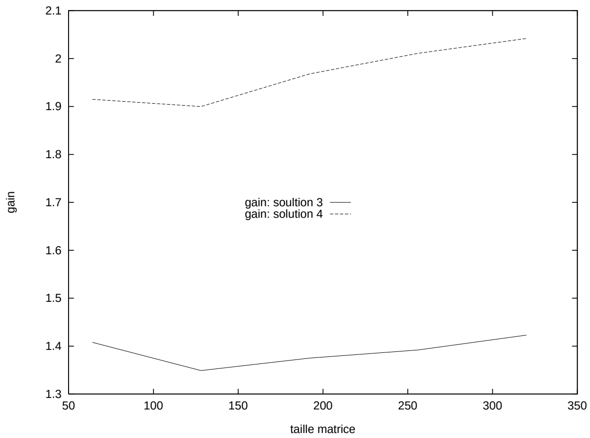

FIG. 2.16 - *Gain pour le troisième exemple sur Intel Paragon.*

une taille optimale théorique et celui donné pour une taille optimale expérimentale dépasse 16%. On remarque que la taille théorique est supérieure à la taille expérimentale. En prenant le résultat théorique, on découpe trop les calculs et les BLAS 3 perdent leur efficacité. Cela vient d'une mauvaise prise en compte des effets de cache au niveau des paramètres du modèle théorique de `DGEMM`. On peut donc remédier à ce problème en affinant le calcul des paramètres. En prenant plus d'intervalles dans les petites tailles de matrices, on modélise plus précisément le temps d'exécution et on évite de surestimer la taille de découpage optimale.

La figure 2.15 donne une comparaison des temps d'exécution de différentes solutions pour l'exemple typique de la première section voir figure 2.1. Le graphe est composé de cinq `DGEMM` utilisant des matrices de tailles différentes, parfois transposées. Les temps des solutions 3 et 4 ainsi que de la version non pipelinée sont donnés pour des tailles de matrice croissantes. Comme prévu dans la figure 2.4, la solution 4 est meilleure que la 3. D'après la figure 2.16, le gain pour la solution 4 avoisine 2 alors que le gain pour la solution 3 est en moyenne 1,4. On peut s'étonner d'un gain inférieur à celui du deuxième exemple. Cependant, dans cet exemple, la meilleure solution comporte deux pipelines reliés par une communication $\oslash$-bloc. On ne peut donc atteindre la performance de l'exemple précédent.

Du coté théorique, notre algorithme a également retenu la solution 4 comme étant la meilleure. Pour valider notre méthode calculatoire dans les cas complexes, nous comparons dans le tableau 2.10 les tailles de découpage optimales expérimentales et celles données par le calcul. A chaque fois un couple est donné puisque le graphe est divisé en deux pipelines. On s'aperçoit de nouveau que les résultats théoriques sont parfois loin de l'expérimental mais que cela n'augmente pas beaucoup le temps d'exécution total. Cela est dû au fait que le plus gros

Sur Paragon	**taille matrice**				
	64	128	192	256	320
Découpage optimal expérimental	4-16	8-22	6-32	8-43	8-54
Découpage optimal théorique	3-9	4-15	5-20	6-24	7-28
Ecart en % entre les courbes expérimentale et théorique	6,25	2,0	2,0	2,3	1,8

TAB. 2.10 - *Taille de découpage optimale pour le troisième exemple sur Intel Paragon.*

temps de calcul se trouve dans le second pipeline. Les matrices utilisées sont plus grandes et donc peuvent être découpées en de nombreux blocs. Expérimentalement, on remarque que le temps du second pipeline varie très peu pour des tailles de découpage inférieures à la valeur optimale. Donc, prendre une taille de 28 au lieu de 54 revient pratiquement au même.

Le temps d'exécution du graphe complet en utilisant les tailles de découpage théoriques est donné dans la figure 2.15. La dernière ligne du tableau 2.10 donne l'écart avec le découpage optimal expérimental. L'écart ne dépasse pas 7%, même pour de petites matrices.

Le quatrième exemple est une comparaison entre la multiplication matricielle PBLAS `PDGEMM` et une routine que nous avons implémenté, utilisant le pipeline. Notre implémentation utilise simplement une distribution par blocs et bien entendu les BLAS 3 pour les calculs séquentiels. Le fonctionnement de notre routine est à peu près similaire à l'algorithme de multiplication de matrices symétriques donné dans le chapitre 5. La différence essentielle vient de la diffusion d'une colonne de blocs au début de chaque pas de calcul. Au lieu d'utiliser la diffusion des BLACS, nous découpons le bloc à envoyer et pipelinons les communications. Chaque processeur effectue cette opération ce qui nous amène à calculer des diagonales de blocs et non des lignes de blocs. Cela implique un équilibrage de charge parfait car chaque processeur travaille avec le même nombre d'éléments à calculer. Pour résumé, à chaque étape de calcul, chaque processeur entame un nouveau pipeline sur sa ligne de processeur. Si la taille de découpage est bien choisie, les temps d'inactivité dûs aux contentions sont minimaux. La figure 2.17 montre une comparaison des temps d'exécution de notre routine et de `PDGEMM` pour des tailles de matrice croissantes, sur 16 processeurs d'une Intel Paragon. Pour `PDGEMM`, nous avons pris la taille de bloc optimale pour distribuer les matrices. Pour notre routine, nous avons utilisé une taille de découpage également optimale. On remarque que la courbe de `PDGEMM` n'est pas parfaitement cubique. Cela vient du déséquilibre de charge pour certaines tailles de matrices. Notre version n'a pas ce problème et donne de meilleurs résultats pour ces tailles particulières. Le gain de notre version par rapport à `PDGEMM` est donné en figure 2.18.

Pour comparer visuellement la différence entre les deux routines, nous présentons en

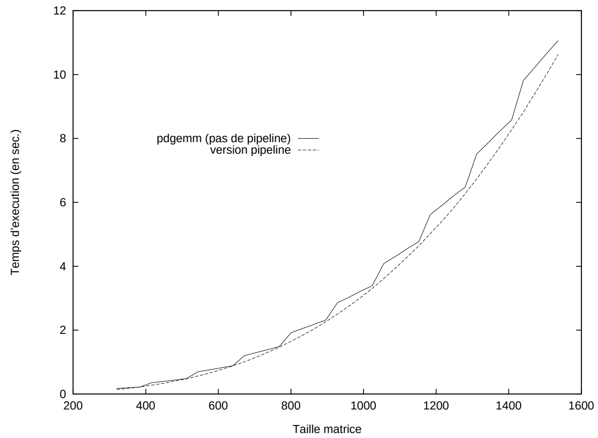

FIG. 2.17 - *Temps d'exécution de* `PDGEMM` *et de notre multiplication matricielle pipelinée sur 16 processeurs d'Intel Paragon.*

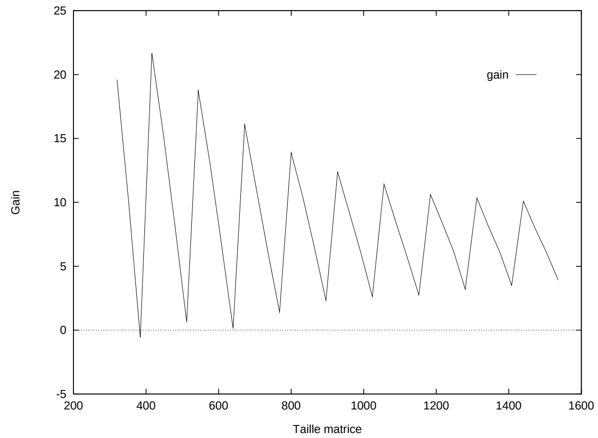

FIG. 2.18 - *Gain en % de la version pipelinée par rapport à* `PDGEMM`*.*

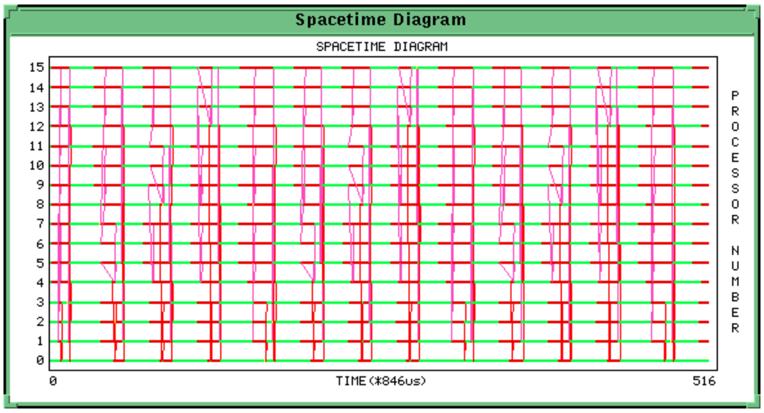

FIG. 2.19 - *Trace d'exécution de* `PDGEMM`*.*

figures 2.19 et 2.20 leur traces d'exécution pour une taille de matrice donnée. La figure 2.19 correspond à l'exécution de `PDGEMM`. Il y a donc de nombreuses diffusions et sommes globales, et les processeurs sont souvent inactifs. La figure 2.20 correspond à l'exécution de notre routine. Dans ce cas, il n'y a que 4 sommes globales (puisqu'il y a 4 diagonales de blocs) et trois phases de pipeline. Grâce aux envois asynchrones, les communications ont lieu pendant les calculs. Cela réduit l'inactivité par rapport à `PDGEMM`.

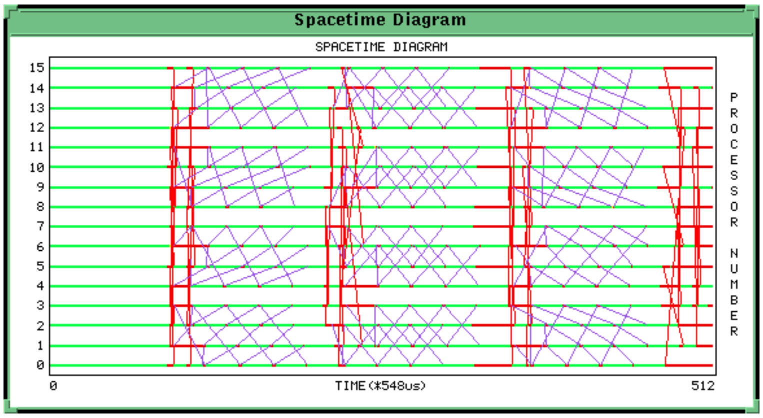

FIG. 2.20 - *Trace d'exécution de notre multiplication matricielle pipelinée.*

```
...
NewPipe(3);
InitTask(0,DGEMM,P_nn,i,i,i,1.2,A,LDA,B,LDB,1.3,C,LDC,NO,MCC,NO,NO,0,1);
InitTask(1,DTRSM,P_llnu,i,i,i,1.2,A,LDA,B,LDB,1.3,C,LDC,MB,MBB,0,0,0,2);
InitTask(2,DGEMM,P_tt,i,i,i,1.2,A,LDA,B,LDB,1.3,C,LDC,MB,NO,0,1,NO,NO);

InitPipe();
/*  RunPipe();  */
FreePipe();
```

TAB. 2.11 - *Code C pour l'initialisation et les calculs de l'exemple 1.*

2.6 Conclusion

Les résultats expérimentaux prouvent largement l'utilité et l'efficacité du pipeline dans un graphe de tâches. Utilisé conjointement avec le recouvrement calcul/communication, on peut réduire de moitié voir plus le temps d'exécution total du graphe. Le problème essentiel reste bien entendu de trouver la taille de découpage optimale. De plus, la syntaxe des BLAS 3 impose une méthodologie de découpage particulière. Suivant les contraintes posées, nous avons montré qu'il est possible de calculer automatiquement cette taille optimale et d'obtenir ainsi un temps d'exécution minimal pour le pipeline.

Actuellement, nous avons implémenté toute la chaîne de traitement du graphe, de la construction des meilleures solutions au calcul de la taille optimale de découpage. Le graphe est déclaré et défini tâche après tâche grâce à une unique fonction d'initialisation qui construit une liste chaînée. Une fonction est ensuite chargée des différents calculs produisant la solution optimale pour le pipeline. Le tableau 2.11 donne le code C qui nous a permis de déterminer la meilleure solution pour le premier exemple.

A terme, notre but est d'implémenter la routine `RunPipe()` qui utilise la meilleure solution et lance le pipeline de manière transparente à l'utilisateur. Ce dernier n'écrira aucun appel aux BLAS 3 ni aux BLACS. Il aura uniquement besoin des quelques routines données dans le tableau 2.11 pour construire un pipeline efficace.

Du point de vue pratique, beaucoup d'applications numériques pourraient bénéficier de nos travaux. Dans [7], les auteurs décrivent des mécanismes pour paralléliser des applications avec deux niveaux de parallélisme, parallélisme de tâches à gros grain au niveau externe et parallélisme de données pour le niveau interne. Leur exemple type est un algorithme de calcul de valeurs propres pour une matrice non symétrique dense [3]. Les factorisations de matrices creuses, comme l'exemple décrit dans [14], peuvent également utiliser nos résultats. Ces algorithmes enchaînent des opérations BLAS sur des blocs denses d'une matrice. Les dépendances entre les calculs sont données par la structure creuse de la matrice, donc on les connaît uniquement à l'exécution. Cela ne pose aucun problème puisque l'initialisation du graphe se fait par routine dynamiquement lors de l'exécution.

CHAPITRE 3 Factorisation LU

"Par exemple, de quel sexe était A'Tuin ? Cette question essentielle, proclament les astrozoologues avec une autorité croissante, ne trouverait de réponse qu'avec la construction d'une rampe plus grande et plus performante qui permettrait de lancer un vaisseau conçu pour l'espace."
T. Pratchett, La huitième couleur.

3.1 De l'intérêt de la factorisation LU

La factorisation LU est au cœur de nombreuses applications. L'utilité de son optimisation n'est plus à prouver vu le nombre grandissant d'applications utilisant de grandes matrices. Une implémentation optimisée peut apporter un gain conséquent sur le temps d'exécution total de l'application même si ce gain dépend énormément de la machine cible. Cependant, une des clés actuelle de la programmation parallèle est la portabilité. Il est donc important d'aboutir à un compromis entre portabilité et optimisation afin d'obtenir une implémentation efficace quelque soit la machine cible utilisée.

Après une description de différentes méthodes proposées pour implémenter et optimiser la factorisation LU, nous décrivons les problèmes posés dans le cas de la factorisation de ScaLAPACK. Ensuite, nous donnons un rappel de la factorisation LU par blocs, telle qu'implémentée dans ScaLAPACK. Puis, nous présentons une modélisation théorique permettant de répondre aux problèmes. Enfin, des résultats expérimentaux sur Paragon et SP2 sont donnés afin de corroborer les résultats théoriques et notre implémentation.

L'ensemble des ces travaux a également été publié dans [22].

3.2 Travaux précédents

De nombreuses méthodes ont été proposées pour implémenter la factorisation *LU* en parallèle. La plupart reposent sur une distribution des données par lignes ou par colonnes, cycliques ou non. Le problème de telles distributions est le déséquilibre de charge qu'elles impliquent pour les calculs et le trop grand nombre de communications avant les mises à jour. Par conséquent, des optimisations basées sur le pipeline et le recouvrement des calculs par les communications ont été proposées.

Par exemple dans [32], les auteurs proposent une distribution par ligne avec une sélection efficace du pivot maximal. Chaque processeur détient un pivot local et le pivot maximal est trouvé grâce à un arbre de diffusion inversé. Dans [10], les auteurs ajoutent à ce principe une stratégie d'équilibrage de charge pour limiter les communications dues au choix du pivot.

Cependant, on constate que ce type de distribution implique beaucoup de communications pour le pivotage puisque le pivot a $\frac{P-1}{P}$ chances d'être choisi en dehors du processeur qui détient la ligne courante. Pour palier à ce problème, Purushotham et al. utilisent dans [48] une distribution cyclique par colonne et un pipelinage des calculs. Une fois le pivot trouvé, il est envoyé de proche en proche sur l'anneau de processeurs. Chaque processeur reçoit puis envoie le pivot avant de mettre à jour une colonne. Dans [17], les auteurs optimisent ce principe en utilisant des communications asynchrones pour recouvrir l'envoi du pivot avec la mise à jour des colonnes.

L'inconvénient majeur de ces deux méthodes est qu'elles reposent sur un parallélisme à grain fin. On ne peut donc pas espérer atteindre les performances de crête des processeurs. Pour répondre à cela, la méthode proposée dans ScaLAPACK utilise une distribution des données cyclique par blocs. Les mises à jour se font sur des blocs en utilisant les routines BLAS. Cette solution apporte une meilleur efficacité des calculs même si aucune optimisation est utilisée. Cependant, elle pose d'autres problèmes que sont le choix d'une taille de bloc ainsi que la forme de la grille de processeurs. Dans [9], des heuristiques pour ces choix sont proposées. Il apparaît que la grille doit être rectangulaire afin de limiter les communications pour pivoter. Quant à la taille de bloc, elle est donnée après expérimentation. La figure 3.1 donne un exemple de l'influence de la taille de bloc sur l'efficacité de la factorisation. Pour une taille de bloc égale à 8, on obtient des performances optimales sur une Intel Paragon. Pour 64, les performances se dégradent sérieusement. Pour une taille de bloc égale à la moitié de la taille de la matrice (= distribution par blocs), cela devient catastrophique.

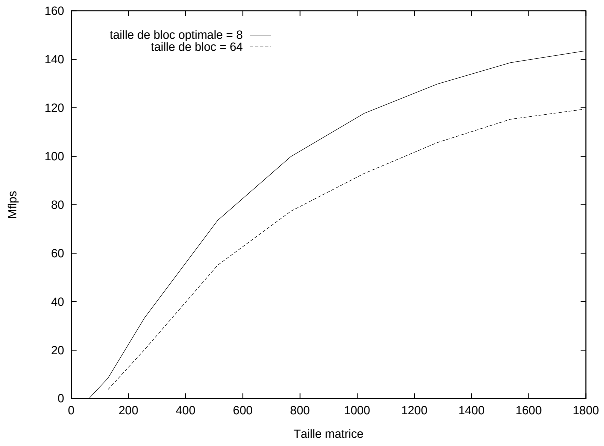

FIG. 3.1 - *Performance (en Mflops) de la factorisation LU sur 4 processeurs pour une taille de bloc de 8 et 64, sur une Intel Paragon.*

3.3 Description des problèmes

Le premier problème est de déterminer la taille de bloc optimale en fonction des paramètres d'exécution (taille et forme de grille de processeurs, taille de matrice). Dans [9], un modèle du temps d'exécution est proposé en fonction du temps moyen d'une instruction flottante, de la bande passante et de la latence du réseau de communication. La forme de la grille est prise en compte mais pas la taille de bloc, ce qui en fait un résultat inutilisable pour résoudre le problème. Dans la section 2, nous présentons un modèle qui intègre la taille de bloc et la forme de la grille.

Le deuxième problème est d'optimiser la routine *LU* grâce aux techniques de pipeline et de recouvrement calcul/communication utilisées dans [17]. L'analyse de la routine révèle quand les processeurs sont inactifs et donc où il est intéressant de pipeliner des calculs. L'avantage de cette optimisation est qu'elle ne brise pas la contrainte de portabilité puisque les communications peuvent être effectuées grâce à des BLACS. Dans le modèle du temps d'exécution, nous savons également quelles sont les communications les plus coûteuses, donc celles qui sont le plus intéressantes à recouvrir. Ce type d'optimisation sort du cadre de ScaLAPACK puisqu'elle utilise des communications asynchrones. Il faut donc qu'elle soit simple pour être facilement adaptable à n'importe quelle machine.

3.4 La factorisation *LU* par blocs en parallèle

La décomposition *LU* par blocs consiste en trois phases, répétées autant de fois qu'il y a de colonnes de blocs à factoriser dans la matrice. En effet, au lieu de travailler sur une seule colonne à la fois, on factorise une colonne de blocs puis on met a jour le reste de la matrice. Pour des raisons de place mémoire, on stocke L et U par dessus A.

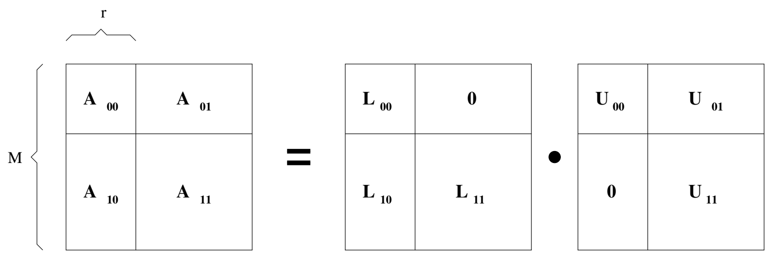

FIG. 3.2 - *Premier pas de la factorisation LU par blocs.*

Conformément à la figure 3.2, les trois phases sont les suivantes:

– Pour obtenir (L_{00}, L_{10}) et U_{00}, une élimination de Gauss est calculée sur $\binom{A_{00}}{A_{10}}$.

$$L_{00}.U_{00} = A_{00} \tag{3.1}$$

$$L_{10}.U_{00} = A_{10} \tag{3.2}$$

– U_{01} est obtenue par résolution d'un système triangulaire (cf. 3.3).

$$L_{00}.U_{01} = A_{01} \tag{3.3}$$

– $L_{11}.U_{11}$ est donnée par l'équation 3.4. Un produit matriciel est utilisé pour calculer $A_{11} - L_{10}.U_{01}$.

$$L_{10}.U_{01} + L_{11}.U_{11} = A_{11} \tag{3.4}$$

Les trois phases sont appliquées récursivement sur A_{11} pour obtenir L_{11} et U_{11}.

L'algorithme parallèle général est donné dans le tableau 3.1. La matrice A est de taille $M \times M$ distribuée sur une grille $P_r \times P_c$ de processeurs avec une taille de bloc de r. Les noms donnés en commentaire sont ceux des routines utilisées dans l'implémentation de ScaLAPACK. `_SCAL` est un BLAS de niveau 1 qui divise un vecteur par un scalaire. `_GER` est un BLAS de niveau 2 qui effectue $A \leftarrow \alpha x.y^T + A$. `_TRSM` est un BLAS de niveau 3 qui résout un système triangulaire ($B \leftarrow A^{-1}.B$). `_GEMM` est le produit matriciel général ($C \leftarrow \alpha A.B + \beta C$).

```
pcol = 0
prow = 0
For k = 0 To M by step r Do
    For i = 0 To r − 1 Do                                              ⎫
        If (my_col = pcol) Then                                        ⎪
            trouve le pivot et sa position                             ⎪
        Endif                                                          ⎪
        diffuse les deux valeurs à tous les processeurs                ⎪
        échange la ligne courante et la ligne de pivot                 ⎬ phase 1
        If (my_col = pcol) Then                                        ⎪
            divise les éléments sous-diagonaux de la colonne i par le pivot /*
    _SCAL */                                                           ⎪
            met à jour les colonnes i+1 à r-1 /* _GER */               ⎪
        Endif                                                          ⎪
    Endfor                                                             ⎭

    If (my_row = prow) Then                                            ⎫
        diffuse L00 à tous les processeurs de la ligne prow            ⎬ phase 2
        résout L00.U01 = A01 /* _TRSM */                               ⎪
    Endif                                                              ⎭

    diffuse L10 sur les lignes de processeurs                          ⎫
    diffuse U01 sur les colonnes de processeurs                        ⎬ phase 3
    met à jour A11 ← A11 − L10.U01 /* _GEMM */                         ⎭

    pcol = (pcol + 1)modPc
    prow = (prow + 1)modPr
Endfor
```

TAB. 3.1 - *Factorisation LU par blocs en parallèle, pour une distribution cyclique par blocs.*

3.5 Modèle théorique

Comme nous pouvons le constater dans le tableau 3.1, chaque pas de la factorisation LU est un enchaînement de calculs et de communications. Le temps de chaque pas est donc conditionné par le processeur qui a le plus de données à traiter et à envoyer. Comme nous utilisons une distribution cyclique par blocs, nous connaissons ce processeur et donc le volume de données qu'il traite et qu'il envoie. Le temps d'exécution de chaque pas est donc la somme des plus gros calculs et communications qui ont lieu. En multipliant ce temps par le nombre de pas, nous obtenons le temps total de la factorisation.

Pour modèle de communication, nous reprenons pour hypothèse $t(L) = \beta + L\tau$. On suppose également que les communications globales effectuées par les BLACS (`DGEBS2D`, `DGMAX2D`...) sont proportionnelles à $t(L)$. Le multiplicateur dépend du schéma de communication employé. Par exemple, une diffusion suivant un arbre est de la forme $t_{tree}(L) = \lceil log_2 P\rceil.(\beta + L\tau)$, alors que suivant un anneau orienté, cela donne $t_{ring}(L) = (P-1).(\beta + L\tau)$.

Pour modéliser le calcul, nous utilisons les résultats donnés dans la section 2. Comme dans [9], nous négligeons le temps des BLAS de niveau 1. Chaque calcul de LU est une routine BLAS et peut donc se modéliser en un polynôme à deux ou trois variables suivant le niveau du BLAS. Les variables correspondent aux dimensions des matrices locales puisque chaque processeur calcule indépendemment des autres. De plus, ces dimensions dépendent directement de la taille de bloc et de la forme de la grille puisqu'on utilise une distribution cyclique par bloc. On obtient donc un temps d'exécution avec les paramètres recherchés.

3.5.1 Temps d'exécution théorique

Posons maintenant $A_{M\times M}$ une matrice à factoriser, distribuée sur une grille $P_r \times P_c$ avec une taille de bloc S_b. Il y a donc $\lceil \frac{M}{S_b} \rceil$ pas de calculs. Le temps total pour chaque calcul ou communication est

– détermination du pivot :

$$T_{detpiv} = \lceil \frac{M}{S_b} \rceil . S_b . f[\tau + \beta]$$

– mise à jour de la colonne de bloc courante (`_GER`):

$$T_{GER} = \sum_{i=1}^{\lceil \frac{M}{S_b} \rceil} \sum_{j=1}^{S_b - 1} [(a_{ger}(S_b \times \lceil \frac{i}{P_r} \rceil) + b_{ger}).j + (c_{ger}(S_b \times \lceil \frac{i}{P_r} \rceil) + d_{ger})]$$

– diffusion du pivot:

$T_{difpiv} = \lceil \frac{M}{S_b} \rceil . S_b . f[\tau S_b + \beta]$ avec $f = \lceil log_2 P_r \rceil$

- échange des lignes :

  $T_{swap} = N_{mem}.[a_{dswap}(S_b \times \left\lceil \frac{\lceil \frac{M}{S_b} \rceil}{P_r} \right\rceil) + b_{swap}] + N_{com}.[\tau(S_b \times \left\lceil \frac{\lceil \frac{M}{S_b} \rceil}{P_r} \right\rceil) + \beta]$

  a_{swap} et b_{swap} sont les paramètres du temps d'exécution d'un échange de deux variables en mémoire. On considère qu'il y a un nombre négligeable de fois où le pivot est à la bonne place et que la répartition du choix des pivots est uniforme. On a donc $N_{mem} \approx \frac{M}{P_r}$ et $N_{com} \approx \frac{M.(P_r-1)}{P_r}$.

- diffusion de L_{00} et L_{10} pour résoudre l'équation 3.3 :

  $T_{difL00} = \sum_{i=2}^{\lceil \frac{M}{S_b} \rceil} [f(\tau(S_b^2 \times \lceil \frac{i}{P_r} \rceil) + \beta)]$ avec $f = \lceil log_2 P_c \rceil$

- résolution du système triangulaire (`_TRSM`) :

$$T_{TRSM} = \sum_{i=1}^{\lceil \frac{M}{S_b} \rceil - 1} [(a_{trsm}.(S_b \times \left\lceil \frac{i}{P_c} \right\rceil) + b_{trsm}).S_b^2 + (c_{trsm}.(S_b \times \left\lceil \frac{i}{P_c} \right\rceil) + d_{trsm}).S_b + (e_{trsm}.(S_b \times \left\lceil \frac{i}{P_c} \right\rceil) + f_{trsm})]$$

- diffusion de U_{01} pour résoudre l'équation 3.4 :

  $T_{difU01} = \sum_{i=1}^{\lceil \frac{M}{S_b} \rceil - 1} [f(\tau(S_b^2 \times \lceil \frac{i}{P_c} \rceil) + \beta)]$ avec $f = \lceil log_2 P_r \rceil$

- Mise à jour du reste de la matrice (`_GEMM`):

$$T_{GEMM} = \sum_{i=1}^{\lceil \frac{M}{S_b} \rceil - 1} [(a_{gemm}.S_b + b_{gemm})(S_b \times \left\lceil \frac{i}{P_r} \right\rceil).(S_b \times \left\lceil \frac{i}{P_c} \right\rceil) + (c_{gemm}.S_b + d_{gemm})\sqrt{(S_b \times \left\lceil \frac{i}{P_r} \right\rceil).(S_b \times \left\lceil \frac{i}{P_c} \right\rceil)} + (e_{gemm}.S_b + f_{gemm})]$$

3.5.2 Taille de bloc optimale

Pour trouver la taille de bloc optimale, nous avons deux possibilités. Soit nous sommons les équations en supprimant les parties entières. On obtient alors un polynôme dérivable par rapport à S_b. Soit nous gardons les parties entières et calculons le temps d'exécution pour une taille de bloc variant entre 1 et M. Le meilleur temps donne la valeur optimale.

La première méthode a l'avantage d'être rapide puisque l'on obtient une équation du degré trois à résoudre. Cependant, les approximations peuvent perturber le résultat et donner une taille de bloc loin de l'optimal (cf. résultats sur SP2). De plus, les paramètres (a_{gemm}, b_{gemm}, ...) des équations sont définis sur des intervalles. Il faut donc faire la dérivée sur toutes les intersections des intervalles (il y a plusieurs BLAS donc des intervalles différents). Ceci peut s'avérer coûteux si trop d'intervalles sont définis.

La deuxième méthode est plus lente mais la précision du résultat est meilleure, surtout pour de petites matrices où les effets de caches sont importants.

3.5.3 Résultats sur Intel Paragon

Pour ces résultats, nous avons employé la méthode de la dérivée afin de trouver la taille de bloc optimale. C'est pourquoi les résultats théorique sont des nombres réels.

taille grille	**taille matrice**							
	1000	2000	3000	4000	5000	6000	7000	8000
4×4	8.31	9.28	9.64	9.83	9.94	10.0	10.1	10.1
8×8	7.38	8.38	8.79	9.03	9.17	9.27	9.35	9.41
16×16	7.26	8.0	8.38	8.61	8.77	8.88	8.96	9.03

TAB. 3.2 - *Taille de bloc optimale théorique sur un système Paragon.*

type	**taille matrice**			
	1000	2000	3000	4000
expérimental	8	8	8	10
théorique	8.31	9.28	9.64	9.83

TAB. 3.3 - *Comparaison entre tailles optimales de bloc théoriques et expérimentales pour 16 processeurs*

Trois remarques sont à faire:

- Les tailles théoriques sont très proches des tailles expérimentales, ce qui indique une bonne modélisation du temps d'exécution.

- La taille de bloc dépend très peu du nombre de processeurs et de la taille de la matrice.
- Après expérience, il s'avère que la forme de la grille influe très peu sur la taille de bloc optimale même si elle est critique pour le temps d'exécution. Une grille avec peu de lignes de processeurs donne un meilleur temps d'exécution puisqu'il y a plus d'échanges de lignes en mémoire.

3.5.4 Résultats sur une IBM SP2

De nouveau nous utilisons la méthode de la dérivée pour trouver la taille de bloc optimale.

taille grille	**taille matrice**							
	256	512	768	1024	1280	1536	1792	2048
2×2	25.5	29.8	32.8	35.5	37.9	40.2	42.3	44.4
3×3	24.1	27.5	29.7	31.6	33.3	35.0	36.6	38.0
4×4	23.5	26.4	28.2	29.7	31.0	32.3	33.5	34.7

TAB. 3.4 - *Taille de bloc optimale théorique sur une IBM SP2.*

type	**taille matrice**							
	256	512	768	1024	1280	1536	1792	2048
3×3 exp.	27	33	30	35	36	35	38	38
3×3 theo.	24.1	27.5	29.7	31.6	33.3	35.0	36.6	38.0
1×9 exp.	27	25	24	25	26	27	26	27
1×9 theo.	20.3	21.7	22.7	23.7	24.6	25.4	26.3	27.1

TAB. 3.5 - *Comparaison entre tailles optimales de bloc théoriques et expérimentales sur une IBM SP2.*

Trois remarques sont à faire :

- Les tailles de bloc expérimentales sont difficiles à obtenir car le temps d'exécution varie beaucoup entre deux tests. Les résultats de la table 3.5 sont donc une moyenne sur plusieurs tests.
- Les valeurs théoriques sont parfois loin des valeurs expérimentales. Cependant, la valeur théorique donne une performance très proche de l'optimal. Par exemple, sur une grille 2×2 avec une matrice de taille 1024, la valeur théorique est 35 alors que la valeur expérimentale est 20. Les performances respectives sont 270 et 280 Mflops ce qui représente une différence de seulement 3.7 %.

- Contrairement à la Paragon, la taille de bloc est très dépendante de la taille de la grille et de la matrice. Cette machine a une performance de crête 5 fois plus grande que la Paragon mais une bande passante à peu près similaire. La taille de bloc augmente donc avec la taille de la matrice pour donner plus de poids aux calculs.

3.5.5 Commentaires

La première remarque concerne la justification de ces calculs. On s'aperçoit que les résultats sur Intel Paragon déservent l'objectif fixé puisque la taille de bloc optimale varie très peu. Cependant, des résultats plus anciens sur une Paragon avec un autre système d'exploitation donnaient une taille de bloc optimale de 16. On voit que les résultats dépendent directement de la machine cible. Les résultats sur la SP2 vont dans ce sens. Il est donc clair que le calcul est justifié par rapport à une recherche fastidieuse de la valeur expérimentale. De même, une valeur prise au "hasard" peut faire chuter les performances de 20% (cf. figure 3.1).

La deuxième remarque concerne la méthode utilisée pour trouver l'optimal. L'inconvénient est qu'elle donne des valeurs réelles. D'après expérience, il est difficile de trouver vers quelle valeur arrondir le résultat théorique. Un mauvais choix peut conduire à des pertes de performance de l'ordre de 10%. Ceci peut justifier l'utilisation de la méthode itérative expliquée plus haut.

3.6 Optimisations

D'après l'analyse de l'algorithme, nous voyons que beaucoup de processeurs sont souvent inactifs. Durant la phase 1, seule une colonne de processeur travaille alors que la phase 2 concerne uniquement une ligne de processeur. On remarque également qu'il y a de gros volumes de communications avant/après ces phases : l'échange des lignes, la diffusion de L_{10} (cf. figure 3.2. Notre but est de réduire cette inactivité en pipelinant des résultats et/ou en utilisant des recouvrements calculs/communication.

3.6.1 Recouvrement de la diffusion

Dans l'implémentation de LU, on remarque que L_{10} est envoyé en même temps que L_{00} bien que cette sous-matrice ne soit pas nécessaire à la résolution du système triangulaire. Elle ne sert que lors de la mise à jour. On peut modifier cela pour éviter de bloquer trop longtemps les processeurs qui participent à la résolution. On se contente d'envoyer L_{00} puis d'utiliser des envois asynchrones pour diffuser L_{10}. Autrement dit:

"Au lieu de diffuser L_{10} avant la résolution du système triangulaire, diffusons le en même temps".

On peut ainsi recouvrir le temps de diffusion par le calcul de `TRSM`.

Cette solution semble intéressante puisque le gain en temps de communication est égal à $T_{gain} = \sum_{i=1}^{\lceil \frac{M}{S_b} \rceil - 1} [f(\tau(S_b^2 \times (\lceil \frac{i}{P_r} \rceil - 1)) + \beta)]$ en supposant que le calcul recouvre entièrement les communications. En fait, ce gain représente seulement 1 à 2 pourcent du temps total d'exécution sur une Intel Paragon. De plus, la factorisation LU est plus rapide sur une grille avec peu de lignes de processeurs. On peut donc laisser de côté cette optimisation sauf pour des machines dont le réseau de communication est très lent et les calculs très rapides.

3.6.2 Recouvrement et pipelinage de l'échange de lignes

Contrairement à la diffusion, l'échange des lignes s'effectue toujours sur la même taille de données au cours de la factorisation. Il prend donc une part plus importante dans le temps d'exécution. On remarque également qu'il a lieu après la factorisation de la colonne de bloc courante, donc après le choix de tous les pivots. Ce qui amène l'idée suivante :

"Au lieu de diffuser les informations de pivotage après la factorisation, pipelinons-les et échangeons en même temps".

On peut donc utiliser les différentes opérations de la phase 1 pour pipeliner les informations de pivotage et échanger par des envois asynchrones les lignes des processeurs actifs. Pendant ce temps, les processeurs qui n'appartiennent pas à la colonne courante peuvent échanger les lignes dès qu'ils reçoivent une information de pivotage.

La figure 3.3 décrit le fonctionnement de cette optimisation pour une matrice $64.S_b \times 64.S_b$ distribuée sur une grille 4×4.

- Etape 1: c'est l'itération k de la phase 1. Le pivot est sur le processeur 11, qui le diffuse sur toute la colonne. Le processeur 15 envoie de manière asynchrone la ligne courante au processeur 11. En même temps, il effectue la division de la colonne (locale au processeur) courante par le pivot (`_SCAL`). De la même façon, le processeur 11 envoie la ligne de pivot au processeur 15 pendant qu'il effectue le `_SCAL`. Les autres processeurs de la colonne courante se contentent d'effectuer le `_SCAL`.

- Etape 2: Les processeurs 11 et 15 envoient de manière asynchrone l'information de pivotage sur leur ligne respective de processeurs. Pendant ce temps, tous les processeurs

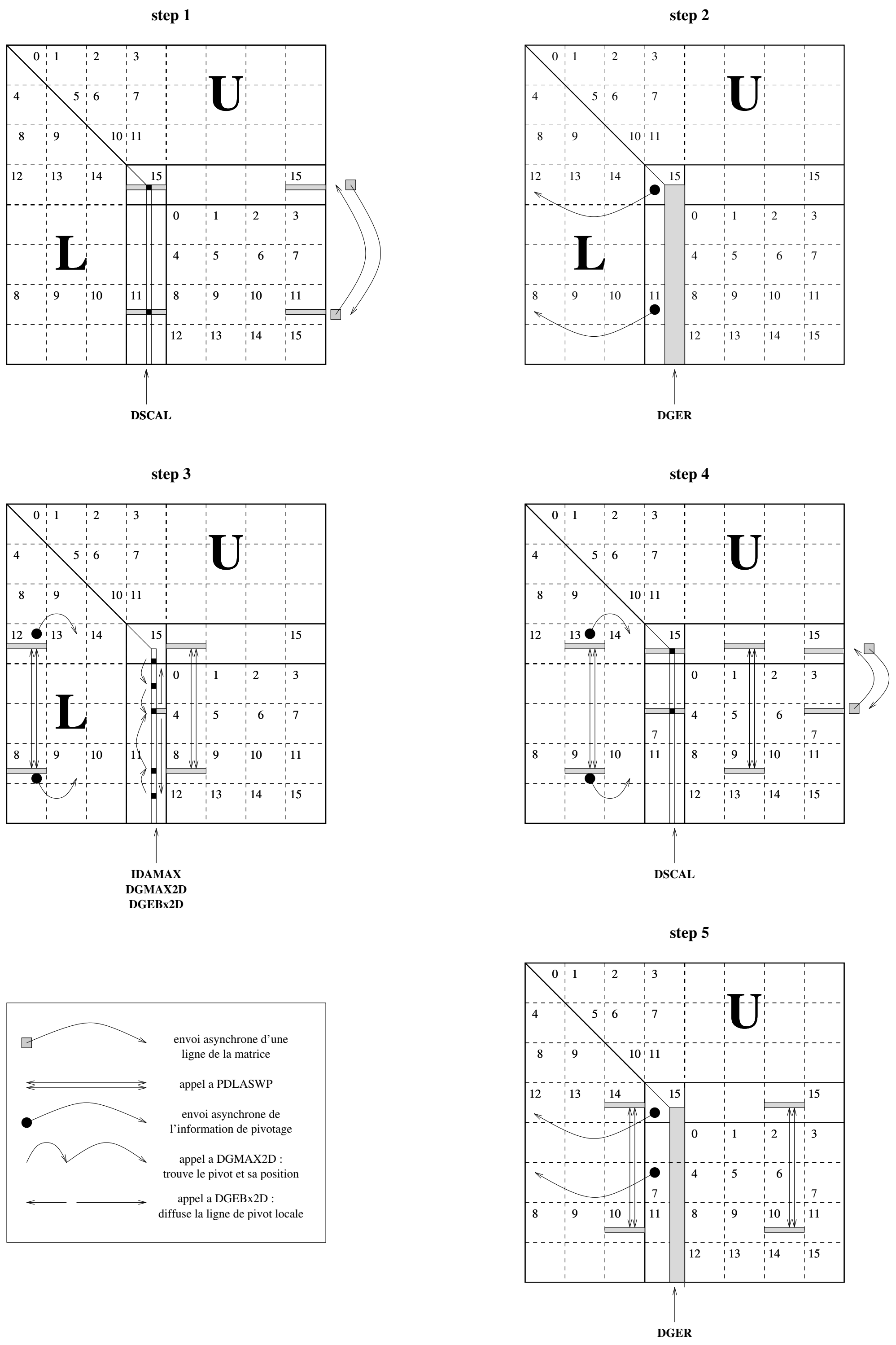

FIG. 3.3 - *Une itération de la phase 1 optimisée.*

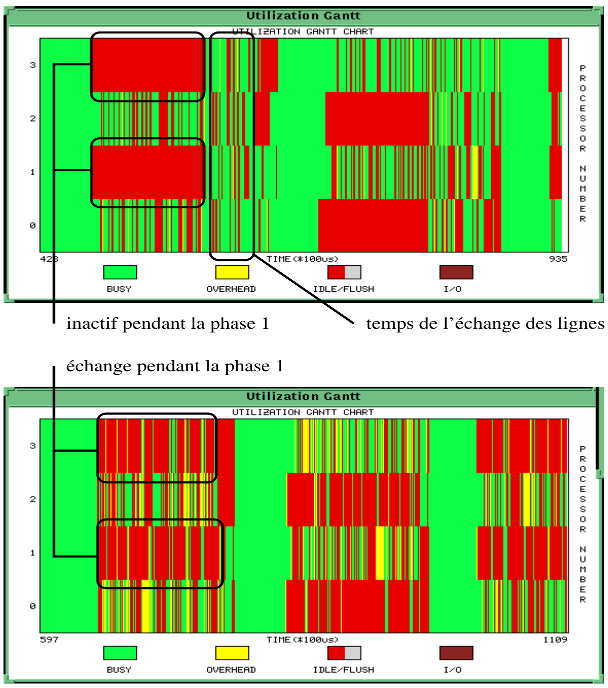

FIG. 3.4 - *Comparaison du diagramme de Gantt pour la version originale et la version optimisée.*

détenant la colonne courante effectuent la mise à jour des colonnes restantes (_GER). Les processeurs 12 et 8 attendent de recevoir l'information.

- Etape 3: Les processeurs 12 et 8 reçoivent l'information, l'envoient à leur voisins 13 et 9, puis échangent la ligne courante avec la ligne du pivot. Pendant ce temps, les processeurs exécutent l'itération $k+1$ de la phase 1.

- Etape 4: c'est l'itération $k+1$ de la phase 1 et se déroule comme l'étape 1. Pendant ce temps les processeurs 13 et 9 échangent leur lignes.

- Etape 5: comme en étape 2, mais avec des processeurs ne détenant pas la colonne courante en train d'échanger.

L'impact de cette optimisation se voit clairement sur la figure 3.4. En haut, pour la version originale, les processeurs 1 et 3 sont complètement inactifs lors de la phase 1. Il y a ensuite une phase de communication pour échanger les lignes. En bas, pour la version optimisée, les processeurs 1 et 3 ne sont pas inactifs : ils échangent les lignes.

3.6.3 Résultats expérimentaux

La version optimisée utilisant le pipelinage des échanges a été testée sur deux systèmes Paragon (ORNL,Tennessee USA, et CDCSP, Lyon, France) et une IBM SP2 (LaBRI, Bordeaux, France). Quatre tailles de grilles ont été employées sur Paragon : 4×4, 6×5 (Lyon), 8×8 et 16×16 (ONRL). Sur la SP2, des grilles 2×2, 3×3 et 4×4 ont été utilisées. Les grilles ont été choisie carrées pour bien montrer l'impact de l'optimisation. Des grilles rectangulaires impliquent un gain moins important puisque l'échange des lignes est plus souvent réalisé en mémoire. Suivant ce principe, une grille constituée d'un seule ligne de processeurs implique un gain nul. Le choix d'une grille carrée est également motivée par le fait qu'un code peut contenir plusieurs appels à ScaLAPACK. Si une multiplication matricielle est effectuée avant la factorisation, il est peut être plus coûteux de redistribuer que de lancer *LU* sur une grille carrée. Mais cela est discuté plus en détail dans le chapitre suivant.

Toutes les expériences ont été exécutées avec le driver de test de ScaLAPACK qui teste l'exactitude des calculs. On a ainsi vérifié la bonne implémentation de l'optimisation. Les résultats sur Paragon sont donnés dans les figures 3.5, 3.6, 3.7 et 3.8. Les résultats sur SP2 sont donnés en figure 3.9 et3.10.

La figure 3.5 montre une comparaison entre la version originale et la version optimisée pour une grille 16×16. La version optimisée augmente plus vite en performance avant de devenir pratiquement parallèle à la version originale. L'optimisation est donc plus efficace pour de petites matrices ($< 375 \times P_c$). Après cette limite, le temps d'échange des lignes devient plus grand que le temps d'exécution de la phase 1 donc le recouvrement n'est pas total et le gain diminue. Ceci est confirmé par la modèle théorique. Dans la figure 3.6, nous donnons le gain théorique et expérimental de la version optimisée par rapport à la version originale pour une grille 4×4. Le gain théorique est donné par le minimum entre le temps total des phases 1 et celui de l'échange des lignes. On constate que le gain expérimental suit bien le comportement théorique.

La figure 3.7 donne le gain pour des grilles 8×8 et 16×16. Il peut atteindre 15% pour de petites matrices et reste au dessus de 4% pour des matrices saturant la mémoire des processeurs. On remarque de nouveau que le gain décroît à partir de matrices de tailles supérieures à $375 \times P_c$.

La figure 3.8 est identique à la figure 3.7, excepté l'axe des abscisses qui est donné en fonction de la taille mémoire de la matrice locale d'un processeur. On remarque que le gain dépend uniquement de ce facteur. Cela est dû à la distribution cyclique par blocs qui assure un bon équilibrage de charge.

la figure 3.9 donne une comparaison entre la version originale et la version optimisée pour une grille 3×3 sur une SP2. On retrouve les même résultats que sur la Paragon excepté pour de grandes matrices. Le gain peut alors devenir négatif comme sur la figure 3.10. Le

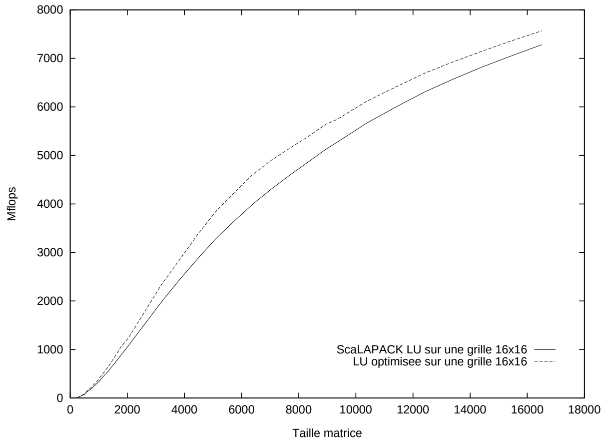

FIG. 3.5 - *Résultats expérimentaux pour une grille 16×16 sur une Paragon.*

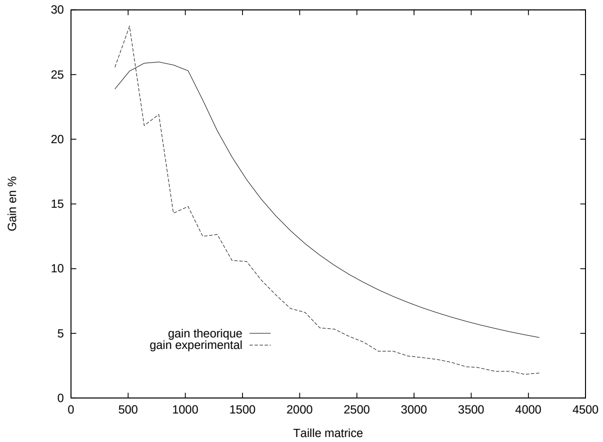

FIG. 3.6 - *Gain théorique et expérimental pour une grille 4×4 sur une Paragon.*

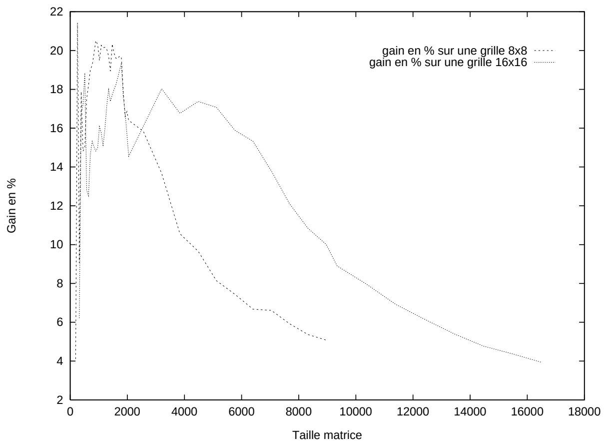

FIG. 3.7 - *Gain pour des grilles 8×8 et 16×16, sur une Paragon*

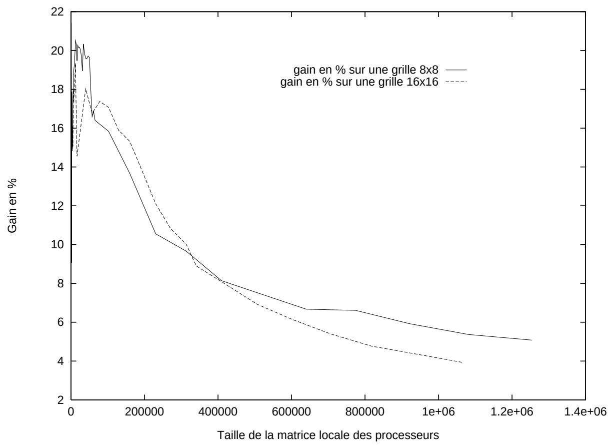

FIG. 3.8 - *Gain pour des grille 8×8 et 16×16 en fonction de la taille de la matrice locale en mémoire.*

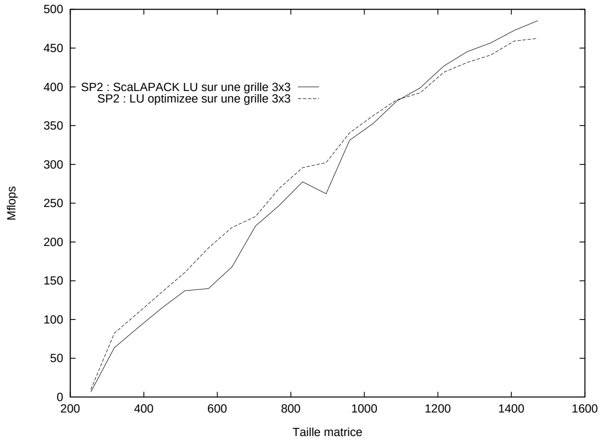

FIG. 3.9 - *Résultats expérimentaux pour une grille* 3×3*, sur une IBM SP2.*

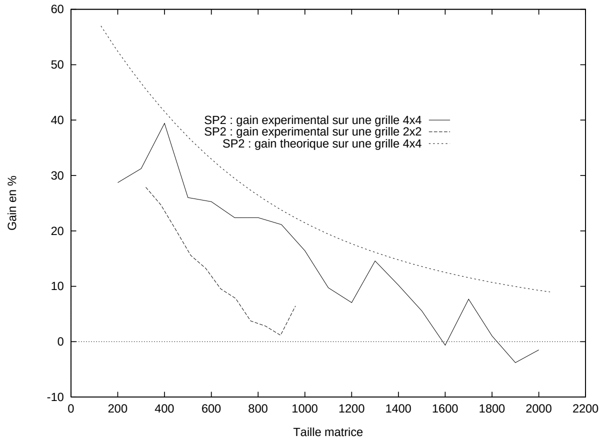

FIG. 3.10 - *Gain théorique et expérimental pour des grilles* 2×2 *et* 4×4*, sur une IBM SP2.*

recouvrement de l'échange de lignes prend alors beaucoup plus de temps que le calcul. Pour les processeurs qui travaillent sur la colonne courante, cela revient pratiquement à échanger après le calcul. De plus, les primitives de communication asynchrone sont plus lentes que les communications bloquantes, en particulier pour de gros messages. Le recouvrement peut donc devenir pénalisant.

3.7 Conclusion

Dans un premier temps, nous avons montré l'utilité et la possibilité de choisir par calcul la taille de bloc pour la factorisation *LU* de ScaLAPACK. Pour cela nous avons repris les travaux de modélisation de BLAS et construit un modèle théorique de l'exécution de la factorisation en fonction de la taille de bloc et de la forme de la grille de processeurs. Ce modèle est bien entendu dépendant de l'algorithme qui sert à implémenter la factorisation. Il est cependant basé sur un principe qui se retrouve dans toutes les routines PBLAS ou ScaLAPACK: l'enchaînement de phases de calcul BLAS et de communications BLACS. On peut donc appliquer ce type de construction du modèle à n'importe quelle routine suivant ce principe. C'est notamment ce que nous avons fait pour les résultats du chapitre 5 et ce qui est utilisé dans le chapitre suivant pour le calcul des redistributions au sein d'un programme HPF/ScaLAPACK.

Dans un deuxième temps, nous avons proposé des optimisations basées sur le pipeline et le recouvrement calcul/communication afin de réduire l'inactivité des processeurs. Le gain obtenu n'est pas négligeable même s'il rend le code non portable. Cependant, le passage de la Paragon à la SP2 s'est fait sans problème puisque seules les primitives de communications asynchrones ont été changées. Il en serait de même pour un portage sur une autre architecture avec MPI. On peut ajouter que le gain moyen (15 %) peut paraître faible mais il montre que les optimisations sont efficaces même pour une routine dont les calculs sont bien plus importants que les communications.

CHAPITRE 4

Calcul des redistributions

"- Distribuez, conclut-il. Qu'on en finisse.
Crane eut préféré que ses mains tremblent moins quand il donna les premières cartes. Ce serait trop bête de tout gâcher maintenant en se faisant accuser de maldonne."
T. Powers, Poker d'Ames.

Nous avons vu dans les chapitres précédents qu'il était possible de modéliser le coût des routines PBLAS, ou ScaLAPACK en fonction de la taille des matrices utilisées, de la forme de la grille, et de la taille de bloc utilisée pour distribuer la matrice sur les processeurs. Nous avons également vu que ces paramètres influaient grandement sur les performances des routines. Par exemple, la factorisation LU est efficace pour une grille avec peu de lignes de processeurs alors que la multiplication matricielle marche mieux avec une grille carrée. De même, le choix de la taille de bloc est critique pour le temps d'exécution.

Dans le cadre d'un programme appelant plusieurs routines PBLAS ou ScaLAPACK, il est évident que le choix d'une distribution unique pour tout le programme n'est pas forcément la meilleure solution. Mais si les routines utilisent chacune une distribution différente, il devient nécessaire de redistribuer les matrices entre chaque opération. La redistribution peut alors s'avérer coûteuse suivant la dimension des matrices, la taille des blocs, le nombre de processeurs... Le temps d'exécution de deux opérations PBLAS ou ScaLAPACK utilisant la même distribution est parfois plus court que si elles utilisaient chacune une distribution optimale avec une redistribution entre. Pour décider si l'utilisation de la redistribution entre deux routines est profitable, il nous faut également une modélisation du coût de la redistribution.

Partant de ces modèles, notre but est de déterminer automatiquement la distribution utilisée par chaque routine afin d'obtenir un temps d'exécution global (redistributions comprises) le plus petit possible.

Ce chapitre est organisé de la manière suivante. Dans une première partie, nous construisons un modèle du temps d'exécution de la redistribution. Dans la deuxième partie, nous présentons un algorithme capable de réaliser notre but. Enfin, nous donnons deux résultats expérimentaux préliminaires sur le comportement de l'algorithme.

Les travaux relatifs à la première partie peuvent également être trouvés dans l'article [15].

4.1 Un modèle pour la redistribution de matrices

Évaluer le temps d'exécution d'une redistribution est complexe. Nous nous plaçons uniquement dans le cadre d'une programmation à la HPF où les matrices sont distribuées par lignes, colonnes, blocs, cycliquement ou non. Nous nous intéressons particulièrement à la redistribution entre deux routines de ScaLAPACK. Cela implique que les matrices utilisées sont distribuées de manière cyclique par blocs. L'avantage de cette distribution est qu'elle inclut toutes les autres précédemment citées. Suivant la forme de la grille et la taille des blocs, il est en effet possible d'obtenir n'importe quelle distribution type HPF. Pour simplification, nous supposerons que les matrices sont toutes distribuées de manière cyclique par blocs et nous utilisons le terme "redistribution générale" comme étant la redistribution de telles matrices.

Soit une matrice A distribuée de manière cyclique par blocs de taille $r \times s$ sur une grille de processeur $P \times Q$. L'objectif de la redistribution est de placer cette matrice sur une grille $P' \times Q'$ en utilisant une taille de bloc $r' \times s'$. Chaque bloc de A va être réaffecté à un ou plusieurs processeurs. Pour chaque élément de A, il faut donc calculer le nouveau processeur auquel il appartient et lui envoyer l'élément en question. La façon d'agréger les éléments pour limiter le nombre de communication ainsi que l'ordre dans lequel sont effectuées les communications est discuté plus loin.

Nous allons voir qu'il est impossible d'obtenir un modèle purement mathématique utilisable pour la redistribution générale. Dans un premier temps, nous donnons quelques références sur le sujet et les problèmes soulevés pour la modélisation de la redistribution. Ensuite, nous présentons une méthode calculatoire donnant le volume de communication engendré par la redistribution. Nous terminons par des résultats sur l'optimalité de l'ordonnancement des messages ainsi qu'un algorithme générant des ordonnancements.

4.1.1 Des études et des problèmes liés à la redistribution

De nombreux modèles et algorithmes ont été proposés pour réaliser une redistribution de données. La plupart traitent uniquement de la redistribution d'un vecteur, certains dans des cas particuliers de distribution ([53], [55], [56]) ou bien dans le cas général ([8], [39]).

Pour la redistribution de matrices, des résultats théoriques sur le coût en communication existent ([42]) mais uniquement dans des cas particuliers (bloc vers bloc, colonne vers bloc ...). Pour le cas général (distribution cyclique par bloc vers distribution cyclique par bloc quelle que soit la taille de bloc), des études sur la redistribution d'une matrice proposent des implémentations ([58], [47]) mais pas de résultats théoriques de complexité.

De tout cela, il ressort deux problèmes lors de la redistribution: la génération des messages et l'ordonnancement de leur envoi. La génération des messages permet de déterminer le volume de communication de chaque processeur. L'ordonnancement influe sur le temps d'exécution global de la redistribution. La modélisation de la redistribution passe donc par une évaluation théorique de ces deux phases.

4.1.2 Calcul de la table des messages

Construire la table des messages revient à calculer le volume de communication que chaque processeur doit envoyer aux autres. Dans [47], un processeur calcule uniquement ce qu'il doit envoyer. Comme nous voulons également construire un ordonnancement des messages, au moins un processeur doit calculer entièrement la table (ou bien recevoir celle des autres processeurs).

Soit une matrice A de taille $M \times N$, distribuée par blocs cycliques de taille $r \times s$, sur une grille de processeurs $P \times Q$. Les processeurs sont numérotés de 0 à $P \times Q - 1$, lignes après lignes. On veut la redistribuer sur une grille $P' \times Q'$ avec une taille de bloc $r' \times s'$. Soient $0 \leq i < P \times Q$ et $0 \leq j < P' \times Q'$. Alors :

$$\begin{aligned} A^{i,j} = \{a_{k,l}, 0 \leq k < M, 0 \leq l < N \quad | \quad & \left(\left\lfloor \frac{k}{r} \right\rfloor \% P\right) \times Q + \left(\left\lfloor \frac{l}{s} \right\rfloor \% Q\right) = i, \\ & \left(\left\lfloor \frac{k}{r'} \right\rfloor \% P'\right) \times Q' + \left(\left\lfloor \frac{l}{s'} \right\rfloor \% Q'\right) = j\} \end{aligned}$$

est l'ensemble des éléments de A devant être envoyés du processeur i au processeur j. Il n'y a pas de méthode directe permettant de calculer le cardinal de cet ensemble. On doit donc parcourir l'ensemble des éléments de A pour construire la table des messages. Si on prend un algorithme glouton qui teste tous les éléments on obtient un temps de construction en $O(n^2)$. Voyons comment réduire ce temps.

La figure 4.1 donne un exemple de redistribution. La matrice a une taille de 120×120. Elle est distribuée sur une grille 2×3 avec des blocs de taille 20×20. On la redistribue sur une grille 2×2 avec des blocs de taille 30×30.

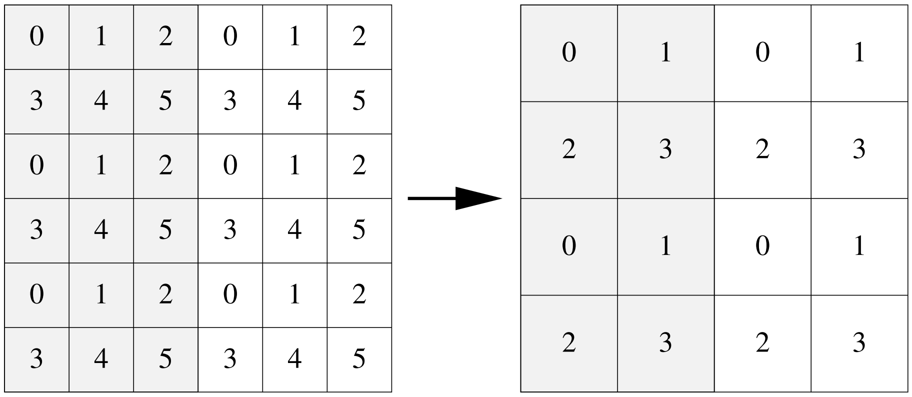

FIG. 4.1 - *Un exemple de redistribution de matrice.*

Pour les deux matrices, on remarque que la partie grisée est placée sur les mêmes processeurs que la partie non grisée. En fait, la partie grisée correspond à la sous-matrice de taille minimale sur laquelle on peut parcourir les éléments afin de construire la table des messages. Soit $L_M = \min(M, ppcm(Pr, P'r'))$ et $L_N = \min(N, ppcm(Qs, Q's'))$, alors la sous-matrice a une taille $L_M \times L_N$. Dans l'exemple, $L_M = 120$ et $L_N = 60$.

Si l'on considère uniquement cette sous matrice, il est évident qu'il n'est pas nécessaire de parcourir tous les éléments. On peut définir des intervalles où les processeurs source et destination sont fixes. Prenons par exemple la première ligne de la sous-matrice. En haut de la figure 4.2, on a découpé cette ligne en segments et marqué le numéro des processeurs qui les détenaient pour chacune des deux distributions. L'intersection des deux ensembles de segments est donnée au-dessous, avec chaque fois un couple de numéro de processeurs. Le chiffre de gauche désigne le processeur qui envoie et celui de droite, celui qui reçoit. Connaissant la taille de bloc pour les deux distributions, il est facile de construire cette intersection et de connaître la taille des segments résultants. La même opération est répétée avec la première colonne de la sous-matrice (à gauche de la figure 4.2).

En faisant le produit cartésien des intervalles de la première ligne et de la première colonne, on obtient un découpage de la sous-matrice en blocs de taille variable. Pour chaque bloc on est assuré que les processeurs source et destination sont fixes. Pour attribuer un couple à chaque bloc, il suffit d'additionner les couples des intervalles correspondant à la projection du bloc sur la première ligne et la première colonne. Le résultat de cette opération pour la sous-matrice d'exemple est donné dans la figure 4.2.

Pour construire la table des messages, il suffit donc de parcourir ces blocs en leur attribuant un couple puis de multiplier leur taille par le nombre d'occurence de la sous-matrice dans la matrice A. On détermine ainsi le volume de communication de chaque processeur.

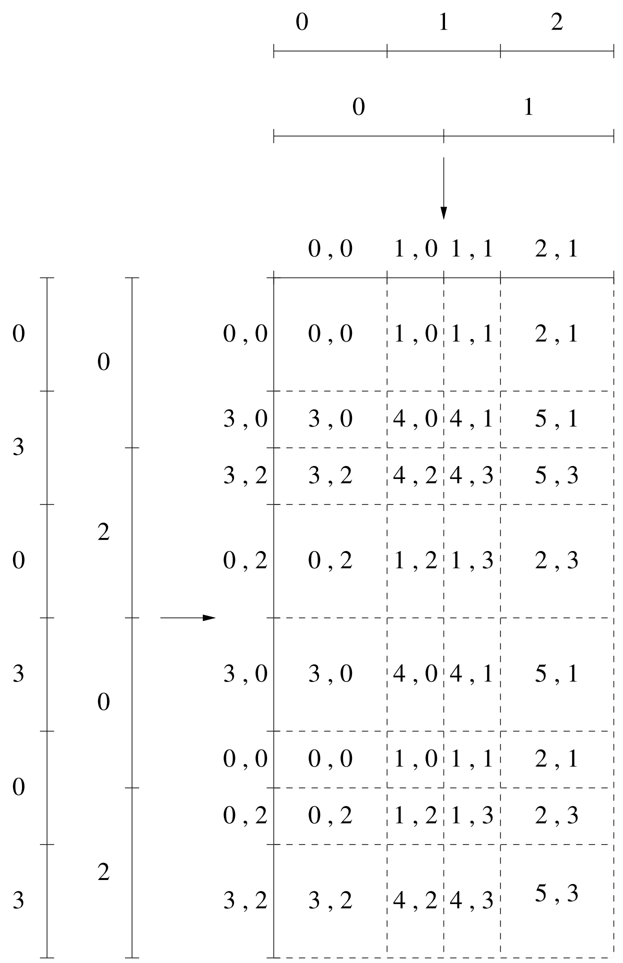

FIG. 4.2 - *Un exemple de redistribution de matrice.*

Il y a au plus $2 \times \frac{L_M}{\min(r,r')}$ intervalles sur la première colonne et $2 \times \frac{L_N}{\min(s,s')}$ sur la première ligne. Le calcul du couple nécessite deux additions. Le calcul du volume trois multiplications (tailles intervalles et nombre d'occurences). On peut donc construire la table des messages en au plus $20 \times \frac{L_M}{\min(r,r')} \times \frac{L_N}{\min(s,s')}$ opérations flottantes.

Le tableau 4.1 donne la table des messages générée pour notre exemple. Les processeurs source sont donnés dans la colonne de gauche alors que les processeurs destination sont sur la première ligne. Chaque case correspond au volume de communication à envoyer d'un processeur source à un processeur destination. On remarque que les processeurs source ont un volume de 2400 éléments à envoyer alors que les processeurs destination ont 3600 éléments à recevoir. Notre but est d'effectuer la redistribution en temps optimal, c'est-à-dire le temps d'envoyer (ou recevoir) 3600 éléments (moyennant les temps de latence). Le fait d'ordonnancer les messages peut-il résoudre ce problème ?

Env./Rec	0	1	2	3
0	1200	0	1200	0
1	600	600	600	600
2	0	1200	0	1200
3	1200	0	1200	0
4	600	600	600	600
5	0	1200	0	1200

TAB. 4.1 - *Table des messages pour* $M = N = 120$, $P \times Q = 2 \times 3$, $r = s = 20$, $P' \times Q' = 2 \times 2$, $r' = s' = 30$.

4.1.3 Ordonnancement des messages

Travaux précédents

Une fois calculée la table des messages, il reste à construire les messages et surtout les envoyer. Très peu de travaux relatifs à l'ordonnancement des messages ont été proposés. La plupart des auteurs proposent des stratégies simples pour réaliser les communications nécessaires à la redistribution. Un échange total comme proposé par Wang et al. [58] nécessite énormément de buffers mémoire pour recevoir les messages, provoque de nombreuses contentions et probablement un dead-lock si les messages sont trop grands. De plus, ce n'est pas une forme d'ordonnancement puisque les messages sont postés simultanément de manière asynchrone.

Dans [47], les auteurs réalisent l'échange total par une suite d'envoi entre paires de processeurs, différentes à chaque étape ("caterpillar"). Cependant, ils ne prennent pas en compte le fait que certaines paires n'ont pas à communiquer. La figure 4.3 montre l'ordonnancement des messages dû au "caterpillar" pour notre exemple. On considère comme une communication la copie des éléments qui ne changent pas de processeur. A gauche est présenté l'ordonnancement théorique, c'est-à-dire comme si l'on synchronisait les processeurs entre chaque étape. Chaque segment en gras correspond à une communication. Sa longueur dépend du volume de communication. Le couple associé indique le processeur source et le destinataire. Les messages sont ordonnés de haut en bas dans le temps pour chaque processeur destinataire. A droite est présenté l'ordonnancement réel. On remarque que les processeurs restent très souvent inoccupés.

Pour réduire ce temps d'inactivité, Walker et Otto proposent d'ordonnancer les messages de tel façon qu'un processeur n'envoie et ne reçoit qu'un seul message à chaque instant de la redistribution ([57]). Bien que cela synchronise les communications, cette stratégie est aussi efficace que celle proposée par Wang et al., et requiert beaucoup moins de tampon mémoire.

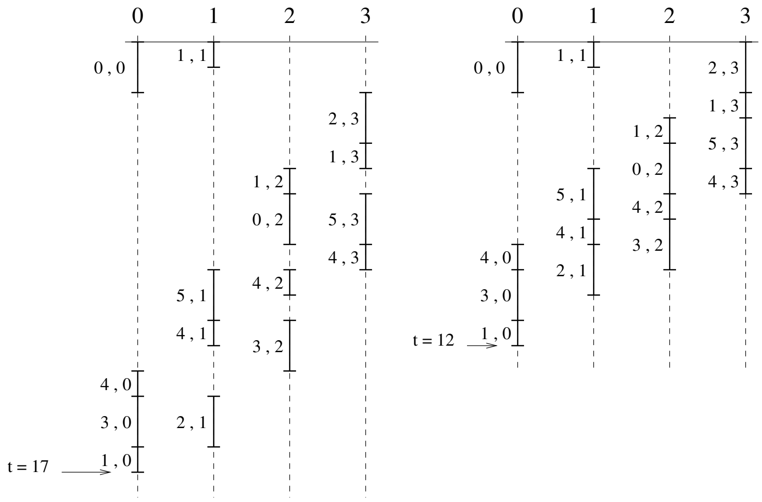

FIG. 4.3 - *Ordonnancement caterpillar pour notre exemple.*

Optimalité de l'ordonnancement

A partir de la table des messages, il est facile de voir quel est le processeur qui envoie ou reçoit le plus. Il est évident que le temps pour communiquer ce volume constitue la borne inférieure du temps de redistribution. On appellera ordonnancement optimal (en terme de temps de communication) un ordonnancement qui atteint cette borne.

Dans [20], les auteurs proposent une extension des travaux de Walker et Otto au problème de la redistribution générale d'un vecteur. Ils construisent un ordonnancement en plusieurs étapes de communications, les processeurs étant synchronisés après chaque étape. Un processeur ne peut envoyer et/ou recevoir qu'un seul message au cours d'une étape. Il est montré comment obtenir un ordonnancement en nombre minimal d'étape ou bien minimisant le coût total. Dans les deux cas, il y a peu de chance que l'on obtienne un ordonnancement optimal, au sens défini ci-dessus.

Dans [19], les travaux précédents sont étendus en supposant que les étapes peuvent être entrelacées. Il est montré notamment que l'on peut toujours obtenir un ordonnancement optimal si les messages sont découpés en communications de taille unitaire. Cette solution n'est pas vraiment satisfaisante au niveau pratique puisque cela implique trop de latence et que le temps de calcul de l'ordonnancement est prohibitif. La question est donc, peut-on obtenir un ordonnancement optimal sans découper les messages ?

Nous n'avons pas de réponse à ce problème pour la redistribution d'une matrice mais nous pouvons exhiber un contre-exemple dans le cas d'un vecteur.

Soit un vecteur de taille $M = 225$, distribué sur $P = 15$ processeurs avec une taille de processeurs recepteurs

processeurs emetteurs	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	3			3			3			3			3		
1	2	1		2	1		2	1		$\boxed{2}$	1		2	1	
2		3			3			3			3			3	
3		1	2		1	2		1	2		1	$\boxed{2}$		1	2
4			3			3			3			3			3
5	3			3			3			3			3		
6	2	1		2	1		2	1		2	1		$\boxed{2}$	1	
7		3			3			3			3			3	
8		1	2		1	2		1	2		1	2		1	$\boxed{2}$
9			3			3			3			3			3
10	3			3			3			3			3		
11	2	1		2	1		2	1		2	1		2	1	
12		3			3			3			3			3	
13		1	2		1	2		1	2		1	2		1	2
14			3			3			3			3			3

FIG. 4.4 - *Table des messages pour $M = 225$, $P = P' = 15$, $r = 3$, $r' = 5$.*

bloc $r = 3$. On veut le redistribuer sur $P' = 15$ processeurs avec une taille de bloc de $r' = 5$. La table des messages est donnée dans la figure 4.4.

On remarque que chaque processeur envoie et reçoit 15 éléments donc un ordonnancement optimal doit conduire à une redistribution effectuée en temps $T_{opt} \approx 15\tau$ ou τ est la bande passante du réseau. Or, il y a en tout $M = 225$ éléments à envoyer. Cela implique que chaque processeur doit à chaque instant de la redistribution envoyer ET recevoir un message.

Réduisons tout d'abord le problème au temps $t \in [0, 3\tau[$. 3 est l'intervalle minimum puisque certains processeurs n'envoient que des messages de taille 3. Essayons de trouver un ordonnancement d'une partie des messages qui tiennent exactement dans cet intervalle (soit un volume de 45 éléments). On commence par choisir 9 messages de taille 3 (cerclés dans la figure 4.4). Il reste donc 18 éléments à envoyer par 6 processeurs. On remarque que l'on peut uniquement envoyer 4 messages de taille 2 et 4 messages de taille 1 pour occuper les 6 processeurs restant jusqu'au temps $t = 2\tau$. Après, on ne peut envoyer 6 messages de taille 1 simultanément. Il est donc impossible de trouver un ordonnancement tenant exactement dans l'intervalle $[0, 3\tau[$.

En élargissant l'intervalle , on parvient à trouver un ordonnancement qui tienne exactement dans $t \in [0, 6\tau[$ (voir figure 4.5).

Cependant $15 - 2 \times 6 = 3$, donc on se retrouve de nouveau devant une impossibilité puisque

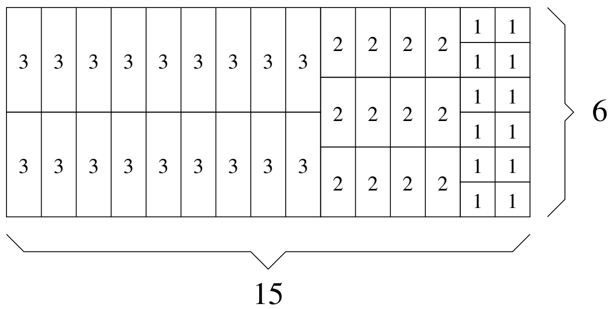

FIG. 4.5 - *Un ordonnancement pour* $t \in [0, 6\tau[$.

l'on vient de démontrer qu'il est impossible d'avoir un ordonnancement tenant strictement dans un intervalle de 3τ. De même, il n'y a pas de solutions pour l'intervalle $t \in [0, 9\tau[$ puisque $6 = ppcm(3, 2, 1)$.

En conclusion, il est impossible de trouver un ordonnancement optimal pour notre exemple. Les conditions pour lesquelles on peut trouver un ordonnancement optimal et comment le construire sont donnés dans [15]. Malheureusement, cela donne peu d'indications pour la redistribution de matrices. Pour revenir à notre objectif premier, nous avons une borne inférieure du temps d'exécution de la redistribution mais pas un temps "réel". Nous avons vu qu'il est difficile d'estimer le coût du caterpillar car certaines communications peuvent être effectuées avant leur commencement théorique. Nous proposons donc un algorithme d'ordonnancement pour essayer d'atteindre l'optimal et surtout donner le temps de la redistribution.

Un algorithme d'ordonnancement

Le principe de l'algorithme est simple. Il repose sur un choix successif de messages dans la table des messages suivant une contrainte stricte et une contrainte non stricte, respectivement :

- Un processeur ne peut envoyer (ou recevoir) au même instant plus d'un message.
- Un processeur ne doit jamais être inactif au cours de la redistribution.

La première contrainte repose sur le fait que les machines parallèles ont le plus souvent un seul port de communication bidirectionnel. Cette contrainte supprime donc les risques de dead-lock. Cependant, des contentions peuvent avoir lieu puisque rien n'est prévu pour éviter que deux messages ne passent par le même lien.

La deuxième contrainte n'est pas stricte pour deux raisons. La première est qu'il n'est pas forcément possible d'obtenir un ordonnancement optimal donc des processeurs peuvent être inactifs. Deuxièmement, lors de la redistribution d'une matrice de taille quelconque, un processeur peut avoir beaucoup plus d'éléments à envoyer (ou recevoir) que les autres. On peut donc avoir un ordonnancement optimal même si à la fin de la redistribution, seulement deux processeurs travaillent.

Ensuite, l'ordre du choix des messages dépend de la table des messages. Trois cas peuvent arriver:

- Le volume maximal envoyé par un processeur est supérieur à celui reçu. Dans ce cas, on cherche d'abord dans la table des messages le processeur (libre suivant la contrainte) à qui il reste le moins d'éléments à envoyer. Ensuite, parmi les destinataires possibles (donc libres) de ce processeur, on prend celui à qui il reste le plus d'éléments à recevoir.
- Le volume maximal envoyé par un processeur est inférieur à celui reçu. C'est le même choix que pour le cas précédent sauf que l'on choisit d'abord parmi les récepteurs et ensuite parmi les émetteurs.
- Le volume maximal envoyé par un processeur égale celui reçu. Dans ce cas, on choisit le couple qui a le moins à émettre et recevoir.

En fait, ces critères permettent de choisir un couple en laissant le plus de liberté possible pour le choix suivant. On peut ainsi facilement saturer tous les nœuds en émission et en réception à n'importe quel instant de la redistribution. La routine de choix de messages est donné dans les tableaux 4.2, 4.3. Le premier tableau correspond au premier cas exposé ci-dessus et le deuxième au troisième cas. Le deuxième cas n'est pas donné car il est pratiquement identique au premier (émetteur et récepteur sont intervertis). L'algorithme général est présenté dans le tableau 4.4. Les variables utilisées sont les suivantes:

- $P_{env} = P \times Q$: nombre de processeurs émetteurs.
- $P_{rec} = P' \times Q'$: nombre de processeurs récepteurs.
- $\mathrm{TM}[P_{env}][P_{rec}]$: table des messages.
- $\mathrm{Tot} = 1 + \sum_{i,j} \mathrm{TM[i][j]}$: total du volume communiqué plus un.
- N_{msg} : nombre de messages.
- $\mathrm{E}[P_{env}]$: table de booléens indiquant si un processeur émetteur est libre.
- $\mathrm{R}[P_{rec}]$: table de booléens indiquant si un processeur récepteur est libre.

- $\max_{env} = \max\limits_{1 \leq i \leq P_{env}} \sum\limits_{j=1}^{P_{rec}} \mathrm{TM}[i][j]$.
- $\max_{rec} = \max\limits_{1 \leq i \leq P_{rec}} \sum\limits_{j=1}^{P_{env}} \mathrm{TM}[j][i]$.
- D_{crit} : vaut 2 si $\max_{env} = \max_{rec}$, vaut 0 si $\max_{env} > \max_{rec}$ et 1 sinon.
- $\mathrm{O}[\min(P_{env}, P_{rec})][\infty]$: tableau servant à stocker l'ordonnancement. Par exemple si $\mathrm{O}[5][4] = \{2, 5\}$, cela signifie qu'au temps t=4, le processeur 2 envoie un message au processeur 5. L'unité de temps est l'envoi d'un message de taille unitaire.

On remarque que le cœur de l'algorithme donné dans le tableau 4.4 se contente d'appeler la routine de choix et de mettre à jour différentes tables. On stocke notamment le message choisi dans un table (O[][]) simulant une sorte de journal des communications dans le temps. Si aucun message n'est trouvé, on laisse un temps d'inactivité, on vérifie quels processeurs sont libres, puis on reprend la recherche.

En fait, cet algorithme est une simplification de celui que nous avons implémenté. Il cherche notamment autant de couples possibles pour *time* fixé avant de mettre à jour la table des messages. De plus, cette variable *time* n'est pas incrémentée de 1 mais d'une valeur suffisante pour atteindre la prochaine fin d'un message.

L'ordonnancement donné par l'algorithme n'est pas toujours optimal. Nous avons donc ajouté la possibilité de couper et d'échanger après le calcul les messages qui "dépassent" le temps optimal. Cela permet de combler des temps d'inactivité que l'algorithme aurait placé dans l'ordonnancement. Comme la recherche des échanges possibles est combinatoire, cela peu prendre énormément de temps avant d'atteindre l'ordonnancement optimal mais on est assuré de le trouver.

Performance de l'algorithme

Pour tous les résultats suivants, l'unité de temps est basée sur le volume de communication. Par exemple, un ordonnancement en temps 15 signifie que l'ensemble des messages inclus dans l'ordonnancement, peut être communiqué dans un temps équivalent à l'envoi de 15 éléments.

Les premiers résultats (tableau 4.5) sont donnés à titre de comparaison avec ceux du rapport [20]. Ils concernent donc la redistribution d'un vecteur de longueur $L = ppcm(Qr, Q'r')$. T_{opt} est le temps de l'ordonnancement optimal. T_{algo} est le temps donné par notre algorithme seul (= sans échange après coup). $T_{rapport}$ est le temps donné dans le rapport [20].

On remarque que pour les deux premières lignes, la méthode du rapport donne un ordonnancement optimal alors que notre algorithme en est seulement proche. En fait, ce sont

```
M = Tot
Choix = {−1, −1}
If (D_crit = 0) Then
   For i=0 To P_env Do
      If (E[i] ≠ 1) Then
         S = Σ_k T[i][k]
         If (0 < S < M) Then
            M = S
            Em = i
         Endif
      Endif
   Endfor
   If (M = Tot) Return Choix
   M = 0
   For i=0 To P_rec Do
      If (R[i] ≠ 1) Then
         S = Σ_k T[k][i]
         If (S > M) Then
            M = S
            Re = i
         Endif
      Endif
   Endfor
   If (M = Tot) Return Choix
   Choix = {Em, Re}
   Return Choix
Endif
```

TAB. 4.2 - *Routine de choix d'un message : $D_{crit} = 0$.*

```
M = Tot
Choix = {−1, −1}
If (D_crit = 2) Then
   For i=0 To P_env Do
      For j=0 To P_rec Do
         If (E[i] ≠ 1) and (R[j] ≠ 1) Then
            S = Σ_k T[i][k] + Σ_k T[k][j]
            If (0 < S < M) Then
               M = S
               Choix = {i, j}
            Endif
         Endif
      Endfor
   Endfor
   Return Choix
Endif
```

TAB. 4.3 - *Routine de choix d'un message : $D_{crit} = 2$.*

```
i=N_msg
time = 0
initialise E et R à 0.
initialise O à -1.
While (i > 0) do
   {Em, Re} ← appel de la routine de choix.
   If ({Em, Re} ≠ {−1, −1}) Then
      If (P_env < P_rec) Then
         O[Em][time,...,time+TM[Em][Re]] = {Em, Re}
      Else
         O[Re][time,...,time+TM[Em][Re]] = {Em, Re}
      Endif
      TM[Em][Re] = 0.
      i = i+1
   Else
      time = time + 1
   Endif
   For j=1 To min(P_env, P_rec) Do
      {Em, Re} = O[j][time]
      If ( {Em, Re} ≠ {−1, −1}) Then
         E[Em] = 1.
         R[Re] = 1.
      Endif
   Endfor
Enddo
```

TAB. 4.4 - *Algorithme principal d'ordonnancement.*

Paramètres					T_{opt}	T_{algo}	$T_{rapport}$
$P \times Q$	r	$P' \times Q'$	r'	L			
1×16	3	1×16	5	240	15	18	15
1×16	7	1×16	11	1232	77	88	77
1×15	3	1×15	5	225	15	18	26
1×12	4	1×8	3	48	6	6	8
1×15	2	1×6	3	90	15	15	18

TAB. 4.5 - *Comparaison entre notre algorithme et la méthode du rapport [20].*

des cas caractéristiques où les messages d'une étape de communication ont tous la même taille. Cela implique qu'il n'y a aucun entrelacement entre les étapes. Nous avons testé l'algorithme en lui faisant choisir les messages dans des sous-ensembles de la table des messages. Les sous-ensembles sont bien entendu une table des messages de même taille. Nous avons alors obtenu un ordonnancement optimal. Comme cela ne concerne que la redistribution d'un vecteur, nous avons préféré laisser de côté ce cas particulier.

A l'inverse, dans les cas où les messages doivent être entrelacés, notre algorithme fonctionne bien mieux que la méthode du rapport. Pour le contre-exemple donné plus haut, on obtient un ordonnancement en temps 18, ce qui est l'optimum sans découper les messages.

Voyons maintenant quelque cas pathologiques (tableau 4.6) où notre algorithme ne donne pas du tout l'ordonnancement optimal. Pour ces cas, nous avons utilisé l'échange et la coupe de messages afin d'améliorer l'ordonnancement. T_{algo}^{ech} donne le temps après la fin des échanges. Le nombre maximum de messages échangés et coupés a été fixé pour limiter le temps de calcul. On constate malgré tout une bonne convergence vers l'optimal.

Paramètres					T_{opt}	T_{algo}	T_{algo}^{ech}
P	r	Q	s	L			
15	4	16	3	240	16	27	17
15	2	14	3	210	15	22	16
15	2	16	3	240	16	21	17
16	9	18	5	720	45	56	46
15	16	18	32	2800	192	208	192

TAB. 4.6 - *Cas problématiques: optimisation de l'ordonnancement de l'algorithme.*

Enfin, nous donnons plusieurs résultats pour le cas de la redistribution de matrices. Le premier résultat concerne l'exemple de la section précédente. L'ordonnancement que produit l'algorithme est donné en figure 4.6. On remarque qu'il est optimal et prend un temps théorique de moitié inférieur à celui du "caterpillar".

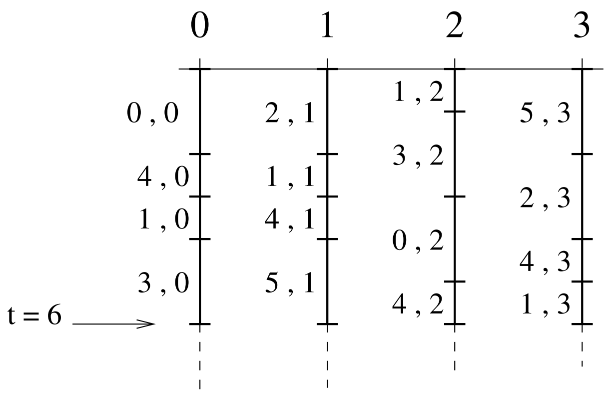

FIG. 4.6 - *Ordonnancement donné par l'algorithme pour $M = N = 120$, $P \times Q = 2 \times 3$, $r = s = 20$, $P' \times Q' = 2 \times 2$, $r' = s' = 30$.*

Le tableau 4.7 donne deux exemples pour lesquels on fait varier la forme des grilles mais pas leur taille. Ceci influe évidement sur la table des messages générée. Le premier exemple ($P \times Q = 12$, $r = 4$, $P' \times Q' = 8$, $r' = 3$) est celui du tableau 4.5 mais appliqué à une matrice. On s'aperçoit que si l'on a des grilles en forme de ligne de processeurs, la table des messages a la même structure que pour le cas vectoriel. La taille des messages est simplement multipliée par la deuxième dimension de la matrice, dans le cas présent 48. Quand la forme de la grille devient rectangulaire, la table des messages change également et l'algorithme ne fournit plus un ordonnancement optimal.

Ce n'est cependant pas une cause de non optimalité comme on le voit dans le premier cas du deuxième exemple ($P \times Q = 16$, $r = 8$, $P' \times Q' = 16$, $r' = 30$). Les grilles choisies sont rectangulaires (2×8, 4×4) et on obtient pourtant un ordonnancement optimal. De même, le fait d'avoir des lignes de processeurs ne garantit pas d'avoir un ordonnancement optimal, comme on le voit pour le deuxième cas.

Paramètres					T_{opt}	T_{algo}
$P \times Q$	r	$P' \times Q'$	r'	M		
1×12	4	1×8	3	48	288	288
3×4	4	2×4	3	48	288	300
2×6	4	4×2	3	48	288	300
2×8	8	4×4	30	1920	230400	230400
1×16	8	1×16	30	1920	230400	264960

TAB. 4.7 - *Résultat de notre algorithme pour la redistribution de matrices de taille $M \times M$.*

Conclusion sur l'ordonnancement

Du point de vue pratique, le fait d'ordonnancer les messages apporte un gain non négligeable sur le temps de la redistribution quand on compare à des méthodes telles que le "caterpillar". Par rapport à l'envoi asynchrone massif, il évite les risques de dead-lock et limite les contentions. De plus, il est facile de mettre en œuvre le schéma de communication à partir de l'ordonnancement puisque chaque processeur peut construire une suite ordonnée d'envois et de réceptions.

Dans ce même esprit pratique, il est inutile d'optimiser l'ordonnancement en coupant et échangeant des messages. Il est en effet impensable de passer plusieurs minutes à optimiser sachant qu'un ordonnancement prend quelques centièmes de seconde pour être calculé, et que le temps de redistribution théorique est de l'ordre de la seconde voire du centième de seconde (voir [58]).

4.1.4 Conclusion

Le fait de calculer la table des messages est une étape nécessaire avant la génération des tampons de communication ou l'ordonnancement. D'un point de vue pratique, notre routine est idéale pour servir de base dans une routine plus générale de redistribution. A part dans des cas limites (taille de bloc unitaire) que l'on peut résoudre autrement, le calcul s'effectue en quelques centièmes de secondes. Certes, il est plus important que celui obtenu dans [47] puisque l'ensemble de la table est calculée mais il reste négligeable par rapport à la génération des tampons ou aux communications. De plus, il permet de lancer notre algorithme d'ordonnancement dont les résultats compensent largement la perte due au temps de calcul supplémentaire.

Du point de vue théorique, nous avons vu qu'il est difficile d'avoir un modèle mathématique du temps d'exécution de la redistribution générale. Nous avons cependant une méthode calculatoire rapide de ce temps. Même si une redistribution avec ordonnancement optimal n'a pas été implémentée, cela permet d'obtenir une valeur théorique valable. On peut également prendre comme temps théorique celui du caterpillar ou bien la borne inférieure du temps de redistribution. Le choix dépend avant tout de la précision ou de la validité souhaitée. En tous les cas, nous avons atteint notre but, à savoir obtenir un temps d'exécution théorique de la redistribution générale.

4.2 Détermination des redistributions dans un programme F77 / ScaLAPACK

La compilation de code d'algèbre linéaire écrits en HPF donne des résultats très inférieurs aux routines des bibliothèques développées spécialement. Il est donc intéressant d'interfacer ces routines avec HPF. Des interfaces ont été réalisées, notamment avec la bibliothèque ScaLAPACK ([6, 43, 5]). Malheureusement, cela demande beaucoup d'efforts et surtout le contrôle sur l'exécution est perdu, notamment en ce qui concerne le choix de la distribution, la taille et la forme de la grille, la taille de bloc... On peut bien entendu spécifier grâce aux directives HPF la distribution ou les redistributions que l'on souhaite mais dans le paradigme HPF, c'est le compilateur qui décide si la redistribution d'une matrice est nécessaire ou non. Or, aucune version de compilateur HPF n'inclut actuellement la redistribution.

Laissons donc de côté HPF et revenons au Fortran 77. L'hypothèse de départ est celle d'un programme F77 appelant les librairies BLAS ou LAPACK. Par programme, nous entendons un ou plusieurs blocs d'instructions inclus dans un unique fichier F77. La première tâche consiste à transformer ce programme en un programme parallèle appelant les PBLAS ou ScaLAPACK. Ensuite, nous voulons générer les distributions et redistributions afin que le temps d'exécution du programme soit minimal. Enfin, on génère un code F77 avec des appels PBLAS ou ScaLAPACK et les redistributions données par la phase précédente.

La première et la troisième parties sont réalisées à l'aide d'un parseur qui repère et extrait les paramètres des appels BLAS ou LAPACK. Vient ensuite la phase de transformation en appel PBLAS et ScaLAPACK et la génération d'une macro-structure contenant tous les paramètres du programme à traiter (nom et paramètres des routines, nom et taille des tableaux...). Le parseur, `NESTOR`, a été développé par G. Silber ([51]) et la phase de transformation développée par C. Randriamaro ([16]).

Nous présentons ici uniquement les travaux liés à la deuxième partie, c'est-à-dire générer des redistributions à partir de la "description" d'un code appelant des routines PBLAS et ScaLAPACK. En premier lieu, nous donnons un aperçu du problème et de ses motivations. Ensuite, nous donnons un algorithme qui génère un ensemble de redistributions qui minimise le temps d'exécution du programme. Enfin, nous présentons les résultats de l'algorithme pour deux squelettes de programmes.

4.2.1 Motivation et problématique

Comme on a pu le voir dans le chapitre 3, la taille de bloc, la forme et la taille de la grille optimales ne sont pas standards pour toutes les routines PBLAS ou ScaLAPACK. Dans [47, 46], les auteurs présentent plusieurs courbes où trois opérations ScaLAPACK sont enchaînées. Il s'agit d'abord d'une multiplication matricielle, puis d'une factorisation LU et enfin la résolution du système. Sur Intel Paragon, les temps d'exécution vont du simple au double quand il n'y a pas de redistribution entre ces opérations. Sur Cray T3D, cela va du simple au triple. L'utilité de la redistribution n'est donc pas à prouver.

Cependant, cet exemple reste simple. Il n'y a que trois opérations enchaînées, sans boucle... Il se trouve que la meilleure solution est de redistribuer entre chaque opération mais est-on assuré que ce soit le cas pour n'importe quelle suite d'opérations?

Dans [42], les auteurs proposent une méthode pour calculer un ensemble de redistributions qui minimise le temps de calcul d'une application d'imagerie. Plusieurs opérations sur une image sont enchaînées, parfois incluses dans une boucle, et les auteurs cherchent quelle redistribution est la meilleure entre chaque opération. Pour cela, ils considèrent un ensemble limité de 6 redistributions dont on a un coût mathématique. Ils construisent un graphe avec toutes les solutions, les arrêtes étant pondérées par le coût de la redistribution. Le problème des boucles est traité en fragmentant la boucle en plusieurs arrêtes dont les poids sont multipliés par un coefficient qui dépend de la taille de la boucle. Ils cherchent ensuite le chemin de poids minimum dans ce graphe et en déduisent le meilleur ensemble de redistributions.

Les résultats de cette approche sont identiques à ceux que nous voulons atteindre mais la technique utilisée ne convient pas. Les hypothèses de départ sont trop restreintes : seulement 6 redistributions possibles et une seule matrice traitée. Si l'on revient à l'expérience menée dans [47], le code contient au moins quatre matrices différentes (3 pour la multiplication plus 1 pour la résolution du système). De plus, ces matrices sont distribuées de manière cyclique par blocs. Cela implique une redistribution générale entre chaque opération. On a vu dans la section précédente qu'il n'y avait pas d'expression mathématique modélisant ce type de redistribution. On ne peut donc utiliser leur méthode pour résoudre notre problème.

4.2.2 Un algorithme de génération des redistributions

Nous avons vu dans le chapitre 3 qu'il est possible de déterminer le temps d'exécution ainsi que taille de bloc optimale d'une routine PBLAS ou ScaLAPACK. De même, nous avons une méthode calculatoire pour connaître le temps d'exécution d'une redistribution. On peut donc déterminer le temps d'exécution total du programme. C'est un critère qui permet de savoir si un ensemble de redistributions est meilleur qu'un autre. Bien entendu, il est impossible de tester tous les ensembles de redistributions possibles. C'est pourquoi, nous partons d'une solution initiale supposée bonne que nous essayons d'optimiser. C'est l'idée principale de l'algorithme, qui comporte 3 phases :

1. détermination des redistributions possibles.

2. construction de la solution initiale.

$A \times B + C \rightarrow C$ (PDGEMM)
LOOP 2
$C \times D + E \rightarrow E$ (PDGEMM)
LOOP 4
$D^T \times E + sub(A) \rightarrow sub(A)$[a] (PDGEMM)
ENDLOOP
$A \times E^T + C \rightarrow C$ (PDGEMM)
ENDLOOP
$B \times C + B \rightarrow B$ (PDGEMM)

TAB. 4.8 - *Exemple 1 de programme.*

[a] $sub(A)$ est une sous-matrice de A.

3. optimisation de la solution principale.

Voyons plus en détail chacune de ces phases en prenant comme exemple le petit programme du tableau 4.8. PDGEMM est la routine PBLAS qui effectue le produit matriciel parallèle.

Détermination des redistributions

Avant toute chose, il convient de construire un tableau avec toutes les redistributions possibles au cours de l'exécution du programme. En effet, la solution optimale est peut être celle où les matrices sont redistribuées entre chaque routine. Nous disposons pour cela de la macro-structure citée précédemment. Nous savons donc le nombre de routines appelées, les matrices que chacune utilise, lesquelles sont dans des boucles et quelle est la taille de ces boucles. Dans l'exemple de le tableau 4.8, il y a 5 multiplications matricielles et deux boucles imbriquées de tailles respectives 2 et 4. Les routines de ScaLAPACK emploient généralement des matrices distribuées identiquement. Cela veut dire par exemple que pour la première multiplication du programme ($A \times B + C \rightarrow C$), A, B et C doivent avoir la même distribution.

Soit D_i la distribution des matrices utilisées par l'instruction numéro i. $R^X(D_i, D_j)$ désigne la redistribution d'une matrice X entre les instructions i et j, mais aussi son temps d'exécution. $R^A(D_1, D_3)$, $R^C(D_1, D2)$, $R^E(D_3, D_4)$ sont des exemples de redistributions possibles. Le schéma complet des redistributions est donné en figure 4.7.

Une flèche indique la redistribution d'une matrice entre deux instructions. On remarque que certaines flèches, en pointillés, sont orientées vers le haut. Elles correspondent aux redistributions engendrées par les boucles. Plus généralement le problème des boucles est le suivant :

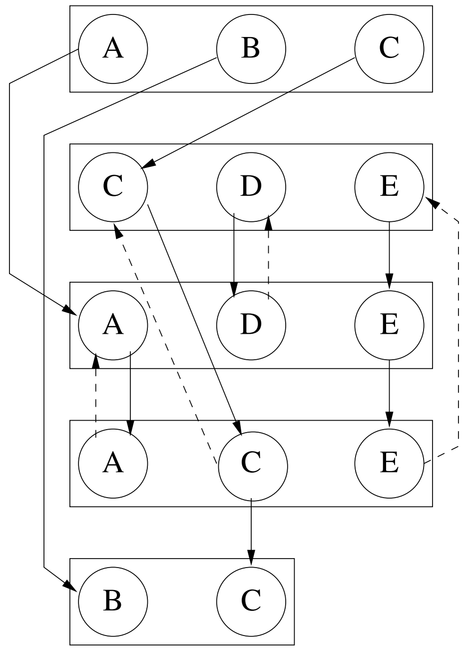

FIG. 4.7 - *Toutes les redistributions possibles pour l'exemple 1.*

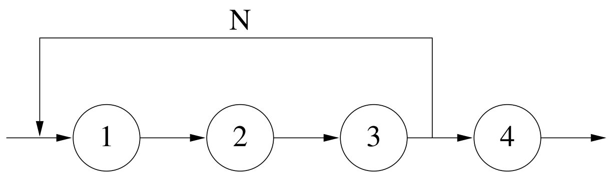

FIG. 4.8 - *Cas d'une boucle.*

Soit quatre instructions dont trois se trouvent dans une boucle de taille N (cf. figure 4.8). Sans la boucle, les redistribution possibles sont $R^X(D_1, D_2)$, $R^X(D_2, D_3)$, $R^X(D_3, D_4)$. Mais il faut tenir compte du fait que les redistributions $R^X(D_1, D_2)$, $R^X(D_2, D_3)$ sont internes à la boucle. Elles sont donc exécutées N fois.

De plus, après l'exécution de l'instruction 3, la matrice X n'est pas forcément distribuée correctement pour l'instruction 1. Il faut donc ajouter la redistribution $R^X(D_3, D_1)$ pour que l'instruction 1 travaille toujours avec la bonne distribution. Cette redistribution est effectuée $N-1$ fois puisqu'après la dernière itération, on passe à l'instruction 4.

En terme de temps de redistribution, on a donc :

$$T_{redist} = N \times (R^X(D_1, D_2) + R^X(D_2, D_3)) + (N-1) \times R^X(D_3, D_1) + R^X(D_3, D_4)$$

Pour résumer, il suffit donc de connaître la taille de la boucle et d'insérer une redistribution supplémentaire avant la fin de la boucle.

Pour résoudre le problème initial, nous parcourons la macro-structure dans l'ordre des instructions et générons une nouvelle redistribution pour chaque matrice utilisée par une routine PBLAS ou ScaLAPACK. Comme nous considérons une redistribution comme un couple de deux distributions ($R^X(D_i, D_j)$), nous gardons en mémoire, pour chaque matrice, le numéro de la dernière routine l'ayant utilisée, c'est-à-dire i. Lorsque l'on rencontre une boucle, on garde en mémoire le numéro de la première routine de la boucle. Dès que l'on atteint la fin de la boucle, on génère la redistribution supplémentaire indiquée précédemment, puis on continue le parcours de la macro-structure.

A la fin du parcours, nous avons déterminé toutes les redistributions possibles ainsi que le nombre de fois où elles sont exécutées. Cependant, nous n'avons pas déterminé les distributions initiales ($P \times Q$, $r \times s$) et finales ($P' \times Q'$, $r' \times s'$) de chaque redistribution. Pour revenir au formalisme mathématique employé précédemment, pour chaque $R^X(D_i, D_j)$ possible, nous avons déterminé X, i, et j mais ni D_i, ni D_j, ni forcement le coût de la redistribution. Ceci est le rôle du calcul de la solution initiale.

Calcul de la solution initiale

De par l'exemple dans [47], nous savons qu'il est souvent profitable de redistribuer entre des opérations qui s'effectuent de manière optimale. Dans le chapitre 3, nous avons montré une méthode pour calculer la taille de bloc optimale des routines PBLAS ou ScaLAPACK, en fonction de la taille de la matrice, de la taille et forme de la grille de processeurs. Si on fixe le nombre de processeurs disponibles, on est capable de déterminer toutes les formes de grille possibles et donc de trouver la distribution optimale d'une routine.

Opération	$P \times Q$	r
$A \times B + C \to C$	3×3	24
$C \times D + E \to E$	3×3	166
$D^T \times E + sub(A) \to sub(A)$	3×3	83
$A \times E^T + C \to C$	3×3	24
$B \times C + B \to B$	3×3	24

TAB. 4.9 - *Solution initiale de l'exemple 1 pour une pile de Pentium.*

C'est ainsi que nous calculons la solution initiale. Nous parcourons de nouveau la macro-structure et pour chaque routine PBLAS ou ScaLAPACK, nous déterminons la distribution optimale ainsi que le temps d'exécution théorique de la routine. Nous parcourons ensuite le tableau des redistributions possibles et initialisons chaque distribution avec celle de la routine correspondante. Nous calculons également le temps d'exécution de chaque redistribution grâce à la méthode exposée dans la section précédente.

Nous avons calculé la solution initiale de l'exemple 4.8 pour une pile de Pentium. Nous avons pris 9 processeurs avec les matrices A, D et E de taille 500×250, B de taille 250×500 et enfin C de taille 500×500. Les blocs sont carrés. Dans le tableau 4.9, nous donnons les distributions trouvées pour chaque routine.

D'après la figure 4.7 et les résultats donnés dans le tableau 4.9, certaines redistributions ont la même distribution initiale que finale. Par exemple la matrice C est redistribuée entre la routine 4 et 5, mais la même distribution est utilisée pour les deux routines. On considère dans ce cas que la redistribution prend un temps nul.

Le temps théorique total d'exécution, redistributions comprises est de 5,64 secondes. Voyons maintenant comment réduire ce temps.

Optimisation de l'ensemble des redistributions

L'optimisation est basée sur la réduction du temps passé dans les redistributions. On sélectionne donc la redistribution la plus coûteuse et on essaie de la modifier pour réduire le temps total d'exécution du programme. On itère sur les redistributions jusqu'à ce qu'il n'y ait plus de modifications possibles. L'algorithme général est donné dans le tableau 4.10. Les variables utilisées sont les suivantes :

- R[N_{redist}] : tableau des redistributions possibles. R[i] est le temps d'exécution de la redistribution i.

- Flag[N_{redist}] : tableau indiquant si une redistribution peut être modifiée.

```
Initialise Flag à 1.
Soit i_max| R[i_max] = max_i R[i] et Flag[i_max] = 1
While (∃j| Flag[j] = 1) Do
   Essaie de modifier R[i_max]
   i_last = i_max
   Soit i_max| R[i_max] = max_i R[i] et Flag[i_max] = 1
   If (i_last = i_max) Then
      Flag[i_max] = 0
      If (∃j| Flag[j] = 1) Then
         Soit i_max| R[i_max] = max_i R[i] et Flag[i_max] = 1
      Endif
   Else
      Initialise Flag à 1.
   Endif
Enddo
```

TAB. 4.10 - *Algorithme principal d'optimisation des redistributions.*

Redistribution initiale	**Redistribution modifiée**
$P \times Q, r \rightarrow P' \times Q', r'$	$P \times Q, r \rightarrow P \times Q, r$
$P \times Q, r \rightarrow P' \times Q', r'$	$P' \times Q', r' \rightarrow P' \times Q', r'$
$P \times Q, r \rightarrow P' \times Q', r'$	$P'' \times Q'', r'' \rightarrow P'' \times Q'', r''$

TAB. 4.11 - *Modification de redistribution.*

La principale remarque sur cet algorithme est qu'il n'y a aucune raison pour qu'il se termine puisque l'ordonnancement des redistributions par temps d'exécution décroissant varie suivant les modifications des redistributions. On peut donc aboutir dans une situation où deux ordonnancements se ramènent toujours l'un à l'autre et donc tomber dans une boucle infinie. Heureusement, le bouclage infini est impossible si l'on considère que les modifications des redistributions ne sont prises en compte que si elles diminuent le temps total d'exécution. A une itération de l'optimisation, on peut parfaitement retrouver un ordonnancement antérieur mais les paramètres de certaines redistributions ne seront plus les mêmes. On ne peut donc jamais avoir deux fois le même ensemble de redistributions au cours de l'optimisation, sauf quand plus aucune modification n'est possible.

Comme il est impossible de tester toutes les modifications possibles, nous avons limité le nombre de modifications à tester à 3. Elles sont données dans le tableau 4.11.

Les trois modifications consistent à “supprimer” la redistribution, ce qui est un choix logique pour réduire le temps total d’exécution. Dans le premier cas, on initialise la distribution finale d’une redistribution avec les paramètres de la distribution initiale. Dans le deuxième cas, on initialise la distribution initiale d’une redistribution avec les paramètres de la distribution finale. Dans le troisième cas, on cherche une distribution unique optimale. Elle est trouvée à partir de “l’addition” du modèle mathématique des instructions qui ont généré la redistribution.

Une quatrième modification a également été testée. Elle consiste à prendre des tailles de blocs multiples entre elles en restant proche de la taille optimale. Cela permet de réduire le temps de la redistribution tout en gardant un temps d’exécution des routines presque optimal. Malheureusement, les calculs sont souvent plus important que les redistributions. Le gain sur les communications est donc annihilé par la perte sur le temps de calcul. Ce type de modification a donc été délaissé.

Bien entendu, la moindre modification d’une distribution a des influences sur d’autres distributions et redistributions. On a déjà dit que les routine PBLAS ou ScaLAPACK utilisaient des matrices distribuées identiquement. Quand on modifie la distribution de l’un d’elles, il faut modifier également celle des autres. Les redistributions liées à ces matrices sont donc également modifiées. C’est pourquoi “supprimer” une redistribution ne diminue pas forcément le temps total d’exécution. En résumé, pour chaque type de modification, nous calculons le nouveau temps d’exécution et prenons la meilleure solution à condition qu’elle diminue ce temps.

4.2.3 Résultats

Nous avons testé notre algorithme sur deux exemples. Le premier est donné dans le tableau 4.8 et le deuxième, dans le tableau 4.12. `PDTRSM` est la routine PBLAS qui effectue la résolution d’un système triangulaire en parallèle.

Les résultats concernant le premier exemple sont donnés dans le tableau 4.13 et dans le tableau 4.14 pour le deuxième exemple. Pour chaque routine, nous donnons la distribution avant optimisation, après optimisation et la meilleure distribution uniforme pour toutes les routines.

Dans l’exemple 1, on remarque que l’optimisation conduit à une distribution unique pour toutes les routines. Il y a donc de réels changements par rapport à la solution initiale même si le gain après optimisation est de 5%. Il faut également savoir que la meilleure distribution uniforme a été calculée à partir du modèle mathématique de `PDGEMM`. Ce n’est donc pas une valeur prise au hasard. Les tailles de blocs conseillées pour l’exécution de cette routine sont généralement 32 ou 64. Les temps théoriques d’exécution pour ces valeurs (distribution

$A \times B + C \to C$ (`PDGEMM`)
LOOP 2
LOOP 4
$A \times B + C \to C$ (`PDGEMM`)
$C^{-1} \times D \to D$ (`PDTRSM`)
ENDLOOP
$D \times sub(B) + A \to A$[a] (`PDGEMM`)
ENDLOOP
$A^{-1} \times D \to D$ (`PDTRSM`)

TAB. 4.12 - *Exemple 2 de programme.*

[a] $sub(B)$ est une sous matrice de B.

Opération	**Avant Optimisation**		**Après Optimisation**		**Distribution Unique**	
	$P \times Q$	r	$P \times Q$	r	$P \times Q$	r
$A \times B + C \to C$	3×3	24	3×3	21	3×3	21
$C \times D + E \to E$	3×3	166	3×3	21	3×3	21
$D^T \times E + sub(A) \to sub(A)$	3×3	83	3×3	21	3×3	21
$A \times E^T + C \to C$	3×3	24	3×3	21	3×3	21
$B \times C + B \to B$	3×3	24	3×3	21	3×3	21
Temps total	5.64		5.34		5.34	

TAB. 4.13 - *Résultats de l'optimisation sur l'exemple 1 pour une pile de Pentium.*

uniforme) sont respectivement 6.67 et 7.52 secondes.

Pour l'exemple 2, pratiquement aucune modification n'est faite. L'optimisation apporte donc un gain inférieur à 2%. Par contre, la solution de l'algorithme est supérieure de 10% à une distribution uniforme. Pour référence, les temps d'exécution pour une distribution uniforme utilisant des blocs de taille 32 et 64 sont respectivement 13.9 et 14.9 secondes.

D'après les deux exemples, on s'aperçoit que l'optimisation ne semble pas vraiment nécessaire puisque les gains sont minimes. En fait, notre optique est seulement d'aider un utilisateur dans la parallélisation d'un code. Il peut donc avoir mis des contraintes de distribution sur certaines matrices dans son programme. Par exemple, il peut spécifier que les produits matriciels doivent s'effectuer avec une taille de bloc de 64. Dans ce cas, notre algorithme tient compte de ces contraintes lors du calcul de la solution initiale. Si les contraintes s'avèrent désastreuses, l'optimisation sera beaucoup plus performante.

Opération	Avant Optimisation		Après Optimisation		Distribution Unique	
	$P \times Q$	r	$P \times Q$	r	$P \times Q$	r
$A \times B + C \rightarrow C$	3×3	24	3×3	24	3×3	24
$A \times B + C \rightarrow C$	3×3	24	3×3	24	3×3	24
$C^{-1} \times D \rightarrow D$	9×1	53	3×3	53	3×3	24
$D \times sub(B) + A \rightarrow A$	3×3	21	3×3	24	3×3	24
$A^{-1} \times D \rightarrow D$	9×1	53	3×3	53	3×3	24
Temps total	10.6		10.4		11.5	

TAB. 4.14 - *Résultats de l'optimisation sur l'exemple 2 pour une pile de Pentium.*

4.2.4 Conclusion

En général, un code parallèle appelant des routines PBLAS ou ScaLAPACK utilise une distribution uniforme des matrices. Ceci est également vrai lorsqu'il s'agit de paralléliser un code F77. Il existe peu de routines qui gèrent la redistribution et celles qui sont inclues dans ScaLAPACK nécessitent une certaines expertise dans l'utilisation de ScaLAPACK. Notre algorithme apporte plusieurs améliorations sur un code utilisant une distribution uniforme. Premièrement, il évite toute une série de tests pour déterminer la bonne taille de bloc et génère automatiquement les redistributions. Deuxièmement, les performances sont équivalentes et souvent meilleures.

Cet algorithme est destiné à être inclus dans un outil d'aide à la parallélisation. Son but est de fournir à l'application principale un moyen d'inclure automatiquement des redistributions dans un code F77 appelant des routines ScaLAPACK. Quelqu'un qui n'a peu ou jamais parallélisé de code et donc certainement jamais entendu parler de redistribution aura tendance à utiliser l'algorithme sur l'ensemble du programme si on lui assure que son programme s'exécute le plus vite possible. Un "expert" en parallélisation saura certainement placer des redistributions aux points critiques mais il pourra utiliser l'algorithme pour ses capacités à calculer de bonnes distributions sur n'importe quelle machine. Dans le cadre de l'aide à la parallélisation, notre algorithme devient donc essentiellement un outil de conseil même s'il est à l'origine conçu pour générer du code optimisé.

CHAPITRE 5 Le calcul de valeur propres

"Toute la philosophie humaine pourrait se réduire à des valeurs intellectuelles, religieuses, morales et esthétiques. Ummon et les Stables, eux, n'en reconnaissent qu'une seule : la valeur existentielle."
D. Simmons, La chute d'Hypérion.

5.1 Introduction

Le problème du calcul de valeurs propres intervient dans de nombreux calculs scientifiques, notamment les simulations en physique, chimie,... Différentes méthodes de calcul sont utilisées, suivant les propriétés et la structure de la matrice à traiter, ou bien suivant les calculs à effectuer. Une synthèse exhaustive de ces méthodes est donné dans la thèse de F. Tisseur [54].

Nous nous intéressons ici uniquement au cas où la matrice est symétrique et dense, définie positive. Nous donnons le résultat de recherches sur deux méthodes particulières: Yau et Lu, et Jacobi. Dans les deux cas, le but est d'exploiter le parallélisme potentiel de ces méthodes. Pour Yau et Lu, une implémentation parallèle ainsi que des résultats expérimentaux sont donnés. Pour Jacobi, un algorithme parallèle ainsi qu'un modèle théorique du temps d'exécution sont présentés.

5.2 Méthode de Yau et Lu

L'approche de Yau et Lu repose sur le principe de l'accélération polynomiale pour réduire le problème des valeurs propres en une série de produits matriciels. L'accélération polynomiale consiste à construire un polynome d'une matrice A tel qu'il atteigne son maximum pour une des valeurs propres de A. Un descriptif complet de la méthode est donné dans les thèses de F. Tisseur [54] et M. Pourzandi [44]. Bien que nécessitant plus de calculs que des méthodes basées sur la tridiagonalisation de matrice, Yau et LU permet plus de parallélisme : parallélisme de données à gros grains pour les calculs matriciels de la phase d'accélération polynomiale, et parallélisme fonctionnel pour la construction des éléments propres. Nous nous attachons ici uniquement au noyau principal de calcul de la méthode, c'est à dire à l'accélération polynomiale. Les travaux précédents de F. Tisseur et M. Pourzandi proposent des résultats sur la version séquentielle de l'algorithme ainsi qu'une parallélisation possible du noyau principal de calcul. Ils proposent également une modélisation théorique de leur algorithme parallèle. Notre but est d'implémenter une version parallèle optimisée de ce noyau avec l'aide des librairies BLAS, PBLAS et BLACS.

L'organisation de cette section est la suivante :

Dans un premier temps, nous décrivons les étapes de calculs que nous avons parallélisées. Ensuite, nous présentons une nouvelle routine PBLAS, spécialement implémentée pour optimiser ces étapes, ainsi que des tests de performance de cette routine. Enfin, nous donnons une comparaison du noyau de calcul de Yau et Lu, avec une routine de calcul de valeurs propres de ScaLAPACK.

L'ensemble de ces travaux peut également être trouvé dans les articles [23, 21].

5.2.1 Implémentation parallèle de l'accélération polynomiale

Partant d'une matrice dense symétrique A, l'accélération polynomiale comporte quatre étapes :

1. Construction de la matrice unitaire e^{iA}.
2. Construction des coefficients $\beta_j, j = 0 \ldots, N-1$ du polynôme P_N.
3. Construction des vecteurs $\beta_j.v_j, j = 0, \ldots, N-1$.
4. Calcul des $u_l = P_N(e^{\frac{il\pi}{N}}.e^{iA})v_0$ à l'aide de la transformée de Fourier rapide.

Les première et troisième étapes nous intéressent particulièrement puisqu'elles comportent de nombreux produits matriciels. Les tableaux 5.1 et 5.2 donnent respectivement les algorithmes des étapes 1 et 3. Ces algorithmes sont facilement parallélisables en utilisant les routines parallèles PBLAS à la place de chaque calcul matriciel séquentiel. Le nom des routines PBLAS utilisées dans la première version parallèle a été ajouté dans les tableaux 5.1 et 5.2 après les formules mathématiques. `PDGEMV` effectue un produit matrice-vecteur général $(A \leftarrow \alpha Ax + \beta y)$. `PDGEMM` effectue le produit matriciel général $(C \leftarrow \alpha A.B + \beta C)$. `PDMATADD` effectue l'addition de deux matrices $(C \leftarrow \alpha A + \beta C)$.

```
/*Initialisation */
A → A_0, c_24 I → C
For i = 1 to 4 Do
      A_{2i-2} * A_{2i-2} → A_{2i} : PDGEMM (→ PDSCMM)
Endfor
For i = 0 to 2 Do
      c_{24+2i} A_{2i} + C → A_{2i} : PDMATADD
Endfor

/* Boucle principale */
For i = 2 downto 0 Do
      C_{8i} I → B
      For j = 1 to 3 Do
            c_{8i+2j} A_{2j} + B → B : PDMATADD
            ⇒ A_8 * C + B → C : PDGEMM (→ PDSCMM)
      Endfor
Endfor
```

TAB. 5.1 - *Algorithme de construction de* $C = \cos(A)$.

```
/* Initialisation */
v_0 → W
C v_0 + i S v_0 → U : PDGEMV
2C * U − W → V : PDGEMM

/* Boucle principale */
For i = 1 to log_2(N/2) do
      2C * C − I → C : PDGEMM (→ PDSCMM)
/*    Mise à jour de W, send/recv entre les colonnes de processeurs:*/
      [perm(V̄), W] → W
/*    Mise à jour en local de U:*/
      [U, V] → U
      2C * U − W → V : PDGEMM
Endfor
```

TAB. 5.2 - *Algorithme pour la construction des* v_j, $j = 0, \ldots, N-1$.

On remarque cependant qu'il existe de nombreux produits matriciels entre des puissances de A ou $C = cos(A)$. Comme A ou C sont symétriques et commutent avec leurs puissances, on obtient toujours des matrices symétriques en les multipliant par elles-mêmes. Cette propriété n'est pas exploitée dans la première version car `PDGEMM` calcule entièrement la matrice résultat, même si cette dernière est symétrique. Nous proposons donc une deuxième version parallèle, optimisée, qui utilise une routine de multiplication de matrices symétriques qui commutent. Nous avons appelé cette routine `PDSCMM` comme indiqué sur les tableaux 5.1 et 5.2 chaque fois qu'il est possible de remplacer `PDGEMM`. Voyons à présent les spécificités de cette nouvelle routine.

5.2.2 La multiplication en parallèle de matrices symétriques commutantes

Notre but est d'implémenter `PDSCMM` comme une routine PBLAS. Elle doit effectuer l'opération $C \leftarrow \alpha AB + \beta C$ avec A, B et C des matrices symétriques de taille $M \times M$. A et B commutent. Nous voulons également qu'elle n'effectue que les calculs nécessaires. Puisque $A.B$ est une matrice symétrique, nous ne devons calculer que la partie triangulaire supérieure ou inférieure.

L'implémentation dans le style des PBLAS implique que les matrices A, B et C de départ ainsi que la matrice résultat (c'est-à-dire C) soient entièrement stockées, bien que symétriques car d'autres PBLAS doivent pouvoir les utiliser. Puisque l'on ne calcule que la partie triangulaire supérieure (par économie de calcul) de la matrice résultat, l'autre partie doit être obtenue par "transposition". Cela implique également une distribution cyclique par blocs des matrices, sur une grille virtuelle de processeurs. Les blocs sont des sous-matrices de taille $S_b \times S_b$ réparties sur P_r lignes et P_c colonnes de processeurs. La figure 5.1 montre un exemple de distributioncyclique par blocs d'une matrice $8.S_b \times 8.S_b$ sur une grille 2×3. Chaque bloc est stocké dans la mémoire d'un processeur particulier. Par exemple, tous les blocs grisés sont sur le processeur 4.

On note $A^{x,y}$ la sous-matrice de A que possède le processeur de coordonnées (x, y) dans la grille (de même pour B et C). Sur la figure 5.1, le processeur 4 est aux coordonnées $(1, 1)$ dans la grille. La sous-matrice de A qu'il possède est composée de tous les blocs grisés et est notée $A^{1,1}$. On note $A^{x,y}_{i \to j, k \to l}$ la sous-matrice de $A^{x,y}$, commençant au bloc (i, k) et finissant au bloc (j, l). Dans l'exemple, la zone grisée de $A^{1,1}$ commence au bloc $(1, 1)$ et va jusqu'au bloc $(3, 2)$. On désigne donc cette zone par $A^{1,1}_{1 \to 3, 1 \to 2}$. Les coordonnées (j, l) étant celle du coin inférieur droit de la matrice locale, on simplifie l'écriture en $A^{1,1}_{1 \to, 1 \to}$. Un bloc unique est noté $A^{x,y}_{i,k}$.

Partant de ces considérations, on peut construire l'algorithme en trois phases présenté dans le tableau 5.3, que chaque processeur exécute.

(mr, mc) sont mes coordonnées dans la grille.
$Work$ et $Work'$ sont des tampons temporaires

/* calcul des blocs diagonaux */
For i $= 0$ *to* $\left\lfloor \frac{M}{S_b} \right\rfloor$ *do*
 $cr =$ i mod P_r ; $cc =$ i mod P_c
 If ($mc = cc$) *then*
  calcule $A^T{}^{mr,mc}_{0\to,\lfloor \frac{i}{P_c} \rfloor} \times B^{mr,mc}_{0\to,\lfloor \frac{i}{P_c} \rfloor} \to Work$ /* MUL */
  Somme globale de $Work$ sur la col. cc /* DGSUM2d */
   le résultat est sur le processeur (cr, cc).
  If ($mr = cr$) *then*
   ajoute $Work$ à $\beta.C^{mr,mc}_{\lfloor \frac{i}{P_r} \rfloor,\lfloor \frac{i}{P_c} \rfloor}$ /* DMATADD */
  Endif
 Endif
Endfor

} phase 1

/* calcul des blocs non diagonaux */
For i $= 0$ *to* $\left\lfloor \frac{M}{S_b} \right\rfloor - 1$ *do*
 $cr =$ i mod P_r ; $cc =$ i mod P_c
 If ($mc = cc$) *then*
  diffuse $A^{mr,mc}_{0\to,\lfloor \frac{i}{P_c} \rfloor}$ sur la ligne mr de processeurs /* DGEBS2D */
 Else
  reçoit du processeur (mr, cc), $A^{mr,cc}_{0\to,\lfloor \frac{i}{P_c} \rfloor} \to Work$ /*DGEBR2D */
 Endif
 /* on ne doit pas calculer les blocs diagonaux donc : */
 If $(mc = cc)$ *then* $\delta = 1$ *Else* $\delta = 0$
 Endif
 calcule $Work^T \times B^{mr,mc}_{0\to,\lfloor \frac{i}{P_c} \rfloor+\delta\to} \to Work'$ /* DGEMM */
 somme globale de $Work'$ sur la col. mc /* DGSUM2d */
  le résultat est sur le processeur (cr, mc).
 If ($mr = cr$) *then*
  ajoute $Work'$ à $\beta.C^{mr,mc}_{\lfloor \frac{i}{P_r} \rfloor,\lfloor \frac{i}{P_c} \rfloor+\delta\to}$ /* DMATADD */
 Endif
Endfor

} phase 2

“transpose” chaque diagonale de la triangulaire supérieure de C } phase 3

TAB. 5.3 - *Algorithme de multiplication de deux matrices symétriques commutantes en parallèle.*

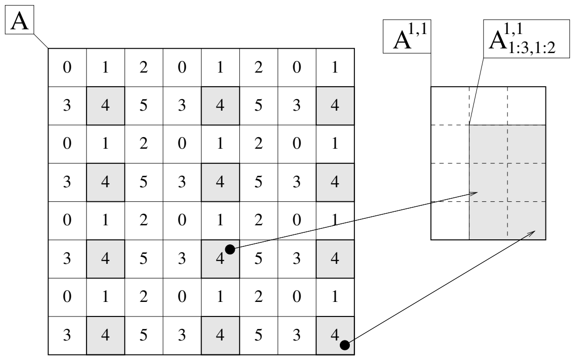

FIG. 5.1 - *Distribution bloc cyclique d'une matrice $8.S_b \times 8.S_b$ sur une grille 2×3.*

Phase 1 : On commence par calculer les blocs diagonaux les uns après les autres. En exploitant la symétrie, le i^{eme} bloc diagonal est le produit de la i^{eme} colonne de blocs de A avec la i^{eme} colonne de blocs de B. Ces colonnes sont sur les mêmes processeurs donc il n'y a aucune communication nécessaire avant le calcul. On peut donc calculer P_c colonnes[1] à la fois en faisant travailler tous les processeurs. Chaque processeur calcule une partie d'un bloc diagonal. Une somme globale sur la colonne de processeur détenant le bloc diagonal permet de reconstituer le résultat final. Cependant, les blocs diagonaux doivent impérativement être "à moitié" calculés puis copiés et transposés pour conserver la symétrie. Il faut donc une routine spéciale pour faire ce travail puisqu'aucun BLAS 3 ne le peut. Elle est notée `MUL` dans le tableau 5.3.

Phase 2 : La partie triangulaire supérieure est calculée lignes de blocs après lignes de blocs. La i^{eme} ligne de blocs est le produit de la i^{eme} colonne de blocs de A (en exploitant la symétrie) et les $\left\lceil \frac{M}{S_b} \right\rceil - i - 1$ dernières colonnes de blocs de B. Le -1 assure que le i^{eme} bloc diagonal n'est pas recalculé (d'où le δ dans le tableau 5.3). Comme A et B sont distribuées de manière cycliques par blocs, la i^{eme} colonne de blocs de A n'est pas sur tous les processeurs détenant B. Il faut donc diffuser cette colonne avant d'effectuer la multiplication (appel au BLACS `DGEBS2D`). Ensuite chaque processeur multiplie (appel au BLAS `DGEMM`) la partie qu'il a reçue (`DGEBR2D`) avec la sous-matrice locale de B qu'il détient. Une somme globale est faite (`DGSUM2D`) pour reconstituer le résultat final sur les bons processeurs.

La figure 5.2 décrit le premier pas de la phase 2 pour une matrice $4.S_b \times 4.S_b$ (S_b est la taille de bloc) distribuée sur une grille 2×2. La figure supérieure donne les blocs concernés par le calcul. Les zones grisées définissent les matrices à multiplier et la sous-matrice de C où est stocké le résultat. Les chiffres sont les numéros des processeurs. La figure du milieu décrit

1. Par colonne, on entend ici colonne de blocs.

le travail effectué par le processeur 1, et la figure du bas celui effectué par le processeur 3.

Considérant uniquement le travail de ces deux processeurs, on remarque que les blocs $C_{0,1}$ et $C_{0,3}$ sont le produit de la colonne 0 de A transposée (détenues par les processeurs 0 et 2) et des colonnes[1] 1 et 3 de B. Il faut donc diffuser la colonne 0 pour que les processeurs 1 et 3 aient les données nécessaires au calcul. Chacun des deux processeurs possède alors une partie de cette colonne de blocs qu'il peut multiplier par les colonnes 1 et 3 de B. Dans la mémoire locale des processeurs, ces colonnes sont consécutives. On peut donc effectuer le calcul en une seule fois et augmenter ainsi la granularité du calcul. Cependant, le processeur 1 n'a calculé qu'une partie des blocs $C_{0,1}$ et $C_{0,3}$, l'autre partie étant calculée par le processeur 3. Il faut donc effectuer une somme globale de ces parties puis additionner le résultat à $C_{0,1}$ et $C_{0,3}$ pour obtenir le résultat final.

Les processeurs 0 et 2 effectuent le même travail sauf qu'ils ne multiplient pas la colonne 0 de A par leur colonne 0 de B (cela reviendrait à calculer le bloc diagonal $C_{0,0}$).

Phase 3: on transpose la triangulaire supérieure de C sur place pour obtenir une matrice parfaitement symétrique. Il s'agit de communications point à point entre des processeurs extra-diagonaux (au sens de la matrice) et non d'une transposition classique.

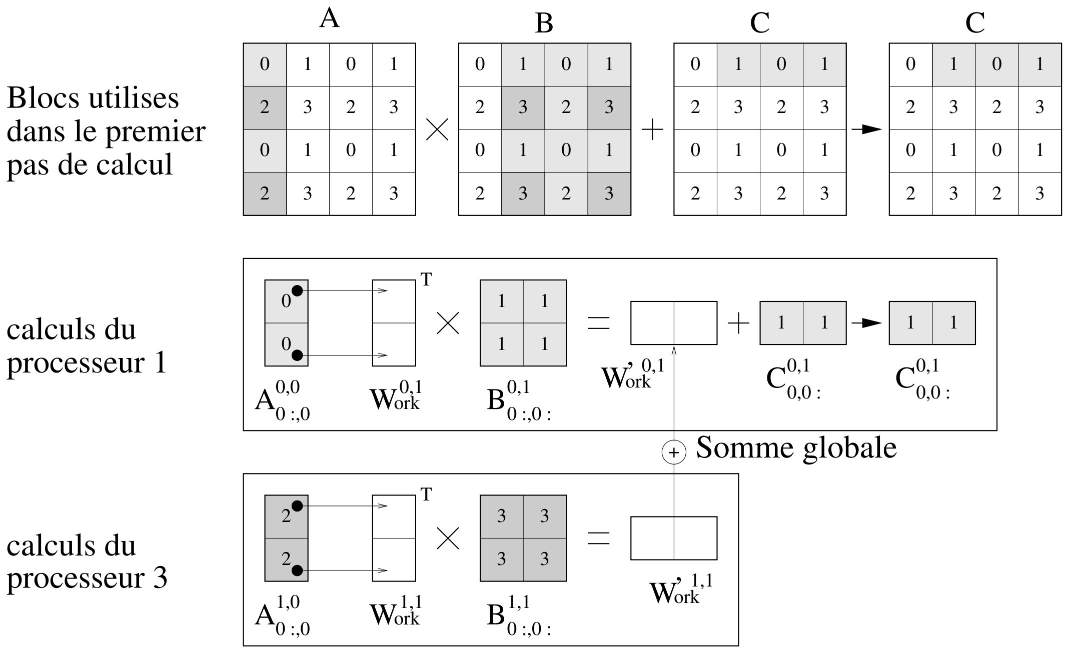

FIG. 5.2 - *Première étape du calcul de blocs non diagonaux sur une grille* 2×2.

Résultats expérimentaux

Avant d'implémenter directement l'algorithme du tableau 5.3, nous avons construit un modèle théorique du temps d'exécution[2] de PDGEMM et de notre routine PDSCMM. Le mode de construction est similaire à la technique employée dans le chapitre 3. PDGEMM est la routine PBLAS effectuant le produit matriciel général. Elle n'exploite donc pas la symétrie. Le modèle théorique a été validé sur une Intel Paragon et a confirmé l'efficacité de notre algorithme puisque le gain théorique approche 2. Après implémentation, les résultats expérimentaux se sont avérés très proches des estimations (moins de 10% d'écart).

Les mesures présentées dans les figures 5.3 et 5.5 ont été obtenues sur le système Intel Paragon de L'IRISA de Rennes. Des tests similaires sur l'IBM SP2 du LaBRI de Bordeaux sont présentées en figures 5.4 et 5.6. Les figures 5.3 et 5.4 donnent le temps d'exécution de PDGEMM et de notre routine PDSCMM en fonction de la taille des matrices. Les calculs ont toujours été faits en double précision avec la taille de bloc donnant le meilleur temps.

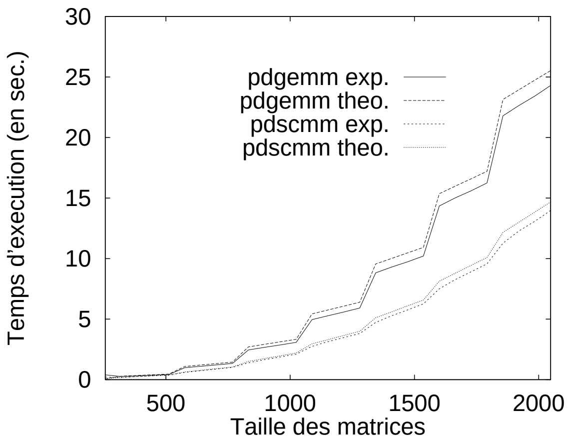

FIG. 5.3 - *Temps d'exécution théorique et expérimental sur une grille 4×4 d'une Intel Paragon.*

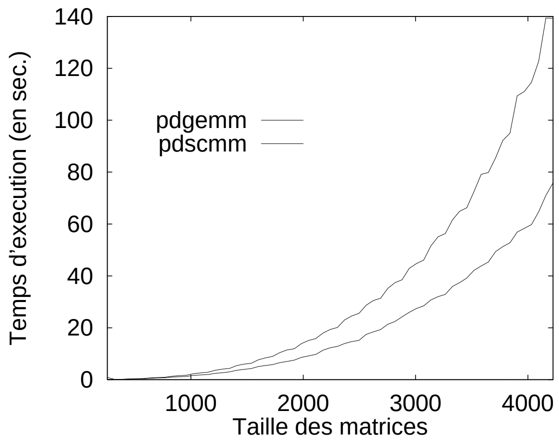

FIG. 5.4 - *Temps d'exécution expérimental sur une grille 3×3 d'une IBM SP2.*

Les paliers à intervalles réguliers sont caractéristiques de l'algorithme. Pour des matrices de taille $i.(P_c \times S_b) + S_b$, le nombre de blocs que calcule le processeur 0 augmente beaucoup, ce qui produit un déséquilibre de charge. Cette augmentation est plus importante pour PDGEMM (plus de calculs), d'où le gain "oscillatoire" présenté en figure 5.5. Le gain est le résultat de la division du temps d'exécution des deux routines. Puisqu'on ne calcule que la triangulaire supérieure dans le cas de PDSCMM, il devrait approcher 2. De fait, pour de grosses matrices, il atteint 2. Pour des matrices de plus petite taille, les communications sont primordiales : PDSCMM fait moitié moins de calcul que PDGEMM mais presque autant de communications. Le

2. les équations sont données en annexe

gain est alors moins important. C'est particulièrement visible pour la SP2 sur la figure 5.6 où le gain augmente plus lentement que sur la Paragon. La vitesse dépend en fait du ratio τ entre le temps moyen d'une instruction flottante et celui de la bande passante du réseau de communication. Ce ratio est grand sur la Paragon. Les calculs prédominent rapidement sur les communications donc le gain augmente rapidement. Sur la SP2, le ratio est petit. Il faut donc calculer sur de très grosses matrices pour atteindre un gain égal à 2.

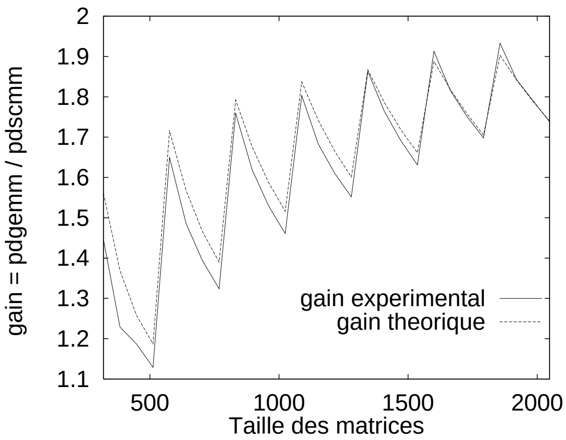

FIG. 5.5 - *Gain de la version symétrique par rapport à* `PDGEMM` *sur Intel Paragon.*

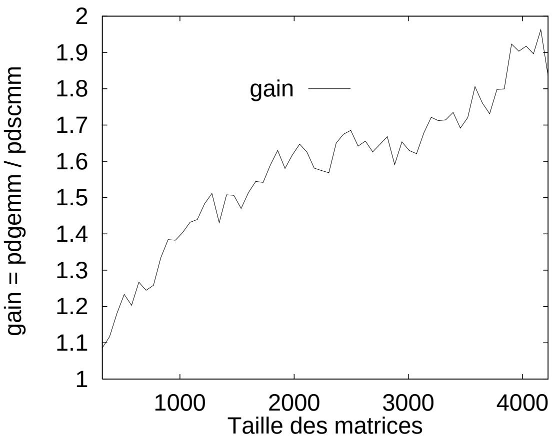

FIG. 5.6 - *Gain de la version symétrique par rapport à* `PDGEMM` *sur IBM SP2.*

Conclusion et remarques

En plus de la réalisation de nos buts et le gain significatif obtenu expérimentalement, l'implémentation de `PDSCMM` a confirmé l'utilité d'avoir à sa disposition un modèle théorique constructif, permettant d'évaluer finement le comportement d'un algorithme utilisant des routines BLAS et BLACS. Cela nous a évité d'implémenter une routine dont l'amélioration par rapport à `PDGEMM` aurait été minime.

Un autre avantage du modèle est qu'il constitue une base pour optimiser la routine qu'il modélise. On peut en voir la démonstration dans le chapitre 3. Dans le cas présent, nous avons construit un autre algorithme et ajouté l'utilisation de recouvrements calculs/communications. Le modèle théorique indique alors un gain supérieur à 2. Mais l'utilisation de communications asynchrones sort du cadre des BLACS et donc des contraintes que nous nous étions fixées au départ de cette étude.

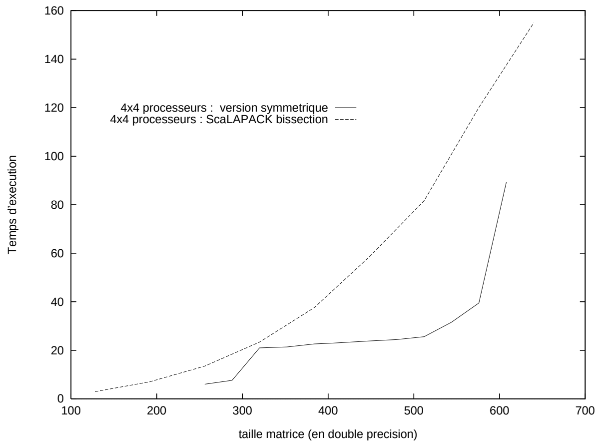

FIG. 5.7 - *Comparaison entre le noyau de calcul parallèle et* PDSYEVX.

5.2.3 Validation de notre approche par comparaison du noyau de calcul de Yau et Lu avec ScaLAPACK

PDSYEVX est une routine de calcul de valeurs propres contenue dans ScaLAPACK. Elle est basée sur une version parallèle de la bissection et de l'itération inverse ([13]). Avant de comparer l'implémentation parallèle du noyau de calcul avec la routine PDSYEVX, nous avons voulu tester l'influence de l'utilisation de PDSCMM sur le temps d'exécution du noyau. Les tests ont eu lieu sur Intel Paragon et IBM SP2. Suivant la machine et la taille de la matrice, nous avons constaté des gains entre 5 et 15% sur le temps d'exécution du noyau de calcul. C'est un gain non négligeable puisque le noyau représente plus de 3/4 du temps d'exécution global (cf. [54]).

La figure 5.7 donne le temps d'exécution de PDSYEVX et du noyau de calcul en fonction de la taille de la matrice dont les valeurs propres sont calculées. Lors du calcul des valeurs propres dans PDSYEVX, une faible précision a été demandée pour réduire le temps d'exécution. On s'aperçoit que notre implémentation parallèle concurrence largement celle de ScaLAPACK, avec des rapports de temps d'exécution allant de 1 à 2,5. Il faudrait cependant ajouter au temps donné par la courbe celui de la construction des vecteurs propres. Cette partie n'a pas encore été implémentée mais son coût théorique ne devrait pas réduire l'efficacité de notre implémentation.

Les marches d'escalier sont dues à la taille de la matrice servant à calculer les vecteurs de Fourier. Pour conserver une bonne précision des valeurs propres, il est parfois nécessaire de doubler cette taille, d'où la brusque augmentation du temps de calcul.

5.3 Méthode de Jacobi

Dans le cadre d'une recherche des valeurs propres en parallèle, nous nous sommes également intéressés à la méthode de Jacobi. Comme pour Yau et Lu, la méthode de Jacobi travaille sur une matrice symétrique. Malgré sa convergence assez lente, elle est plus stable que la méthode QR (sous certaines conditions [36]) et surtout fortement parallélisable.

Après une aperçu de la méthode séquentielle, nous faisons un état de l'art sur quelques parallélisations proposées dans la littérature existante. Nous donnons ensuite un algorithme parallèle par bloc qui applique la méthode de Jacobi. Enfin, nous exhibons un coût théorique de cet algorithme et le comparons à celui donné dans [35].

5.3.1 L'algorithme séquentiel

L'idée principale de la méthode de Jacobi est de construire une suite de matrices $A_i, i = 1, 2, \ldots$ telles que $A_{i+1} = J_i A_i J_i^T$, avec $A_1 = A$. Sous certaines conditions, la suite converge vers une matrice diagonale $D = J_n J_{n-1} \ldots J_2 Q_1 A J_1^T J_2^T \ldots J_{n-1}^T J_n^T$ qui contient les valeurs propres de A. Chaque matrice J_i, dite de Jacobi ou de rotation, est une matrice carrée de la forme $J(p, q, \theta)$ telle que :

$$j_{pq} = sin(\theta),\ j_{qp} = -sin(\theta),\ j_{pp} = j_{qq} = cos(\theta),\ j_{rs} = \delta_{rs}, r \neq p, q \text{ et } s \neq p, q.$$

L'angle θ est choisi afin que le résultat du produit $J(p, q, \theta) A J^T(p, q, \theta)$ annule les éléments a_{pq} et a_{qp}. On détermine θ par :

$$tan(2\theta) = \frac{2a_{pq}}{a_{qq} - a_{pp}}.$$

On défini un critère de convergence ε tel que la solution est atteinte lorsque :

$$off(A) = \sqrt{\sum_{j=1, j\neq i}^{n} \sum_{i=1}^{n} a_{ij}} < \varepsilon.$$

La méthode complète consiste à annuler plusieurs fois tous les éléments sous-diagonaux de A en choisissant les couples (p, q) dans un ordre tel qu'il assure la convergence de l'algorithme. L'annulation des $\frac{n(n-1)}{2}$ éléments sous-diagonaux constitue un "balayage" (ou sweep). En général la convergence est assurée au bout de $log_2 n$ balayages.

5.3.2 Travaux relatifs à Jacobi en parallèle

On remarque premièrement que le choix des couples (p,q) et du calcul de θ peut s'effectuer en parallèle. Secondement, chaque rotation (p, q) entraîne uniquement la modification des lignes et des colonnes p et q de la matrice A. Pour une matrice A de taille n répartie sur $P = \frac{n}{2}$ processeurs, on peut donc calculer $\frac{n}{2}$ rotations et effectuer la mise à jour de A en parallèle moyennant les communications pour diffuser les rotations à tous les processeurs.

Dans [44] et [45], les auteurs utilisent ce principe mais les performances sont peu satisfaisantes à cause du grain de calcul. En effet, les mises à jour sont effectuées par des BLAS de niveau 1 à cause de la distribution par colonnes cycliques de A. De plus, il faut une phase d'échange total pour que tous les processeurs disposent des rotations avant la mise à jour. Une optimisation utilisant des recouvrements calcul/communication est proposée mais ne parvient pas à réduire de manière significative le coût des communications. De plus, la symétrie n'est pas exploitée.

Dans [1], la matrice est stockée de manière cyclique par colonnes en groupant les colonnes par deux. De plus, un découpage des colonnes permet d'exploiter la symétrie tout en conservant une charge de calcul équivalente pour tous les processeurs. Cependant, la phase d'échange total est toujours nécessaire et les mises à jour sont des opérations vecteurs-vecteurs.

Dans [35], les auteurs résolvent le problème des calculs à grain fin en proposant une version par blocs de la méthode de Jacobi. La matrice est divisée en blocs de taille $\frac{n}{2\sqrt{p}}$ où n est la taille de la matrice, et p le nombre de processeurs. Chaque bloc du quadrant sud-ouest de la matrice est attribué à un processeur différent. Les quadrants nord-ouest et sud-est de la matrice sont "repliés" sur le quadrant "sud-est" afin que les processeurs aient la même charge de calcul et pour exploiter la symétrie. Chaque bloc est ensuite subdivisé en un multiple de $2s$, où s est la taille de bloc sur laquelle sont calculées les rotations. Les auteurs donnent une modélisation du temps théorique d'exécution ainsi que des résultats expérimentaux sur une Touchstone DELTA. Les résultats sont plus intéressants que dans les deux cas précédents puisque les calculs de mise à jour sont faits par des BLAS de niveau 3. De plus, les auteurs obtiennent une très bonne efficacité pour des matrices assez grandes.

5.3.3 Un algorithme parallèle par blocs

Comme pour la référence précédente, notre algorithme repose sur un découpage d'une matrice A en blocs de taille $S_b \times S_b$. L'objectif principal est de distribuer ces blocs sur les processeurs afin que les calculs de mise à jour (l'opération la plus coûteuse) utilisent efficacement les BLAS de niveau 3, tout en utilisant la symétrie. Il faut donc accumuler les couples de rotation dans des matrices et diffuser ces matrices pour effectuer la mise à jour. Pour diminuer le coût total, nous voulons également que cette diffusion ne soit pas un échange total et que la phase de transfert des blocs après la mise à jour n'occasionne aucune contention dans le réseau d'interconnexion.

La distribution initiale des blocs

Pour obtenir les conditions posées, nous proposons la distribution initiale présentée en figure 5.8. On ne stocke que la partie triangulaire inférieure de la matrice A. Le numéro dans chaque bloc est celui du processeur dans lequel il est stocké. Pour cet exemple, nous utilisons une grille de processeurs de taille de 3×3.

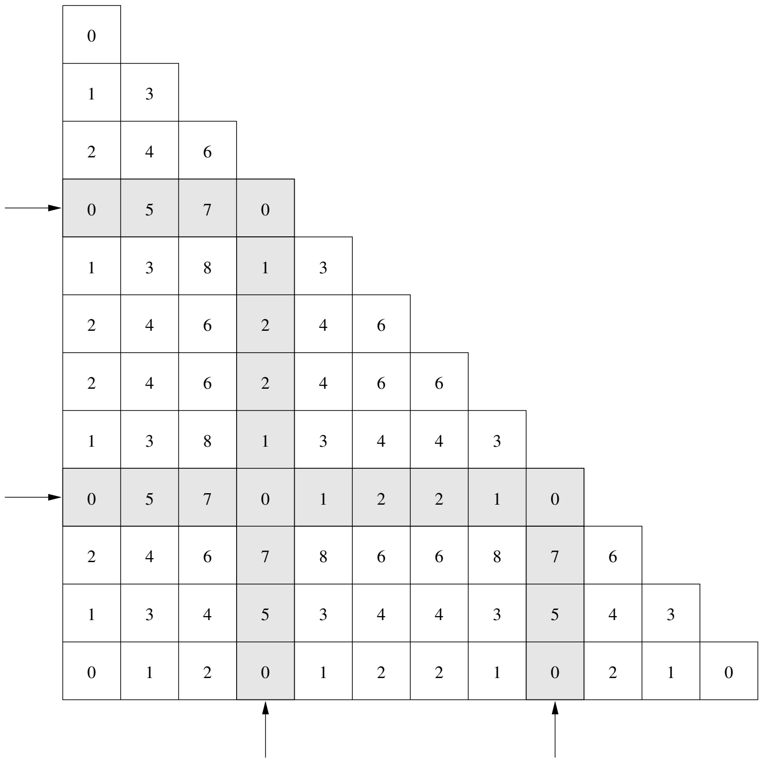

FIG. 5.8 - *Distribution initiale pour l'algorithme de Jacobi.*

On remarque que les lignes et colonnes de blocs forment des palindromes "imbriqués" les uns dans les autres. Prenons par exemple les lignes et colonnes de blocs 4 et 9 (en grisé) sur la figure. Si on transpose les colonnes, on s'aperçoit que les lignes de blocs 4 et 9 sont stockées sur les même processeurs. La même remarque s'applique aux couples de lignes $(1, 12)$, $(2, 11)$, ..., $(6, 7)$. Même si cette distribution semble compliquée, on peut l'obtenir facilement à partir d'une distribution cyclique par blocs, en utilisant uniquement des communications entre processeurs voisins.

Voyons maintenant pourquoi nous choisissons une telle distribution.

Le calcul des rotations

Comme dans [35], le calcul des rotations est effectué en séquentiel en accumulant les rotations dans une matrice de rotation. Les données nécessaires aux calculs des rotations doivent être des couples de blocs stockés sur un même processeur. Par exemple, on peut calculer une matrice de rotation à partir des blocs $A_{4,4}$ et $A_{9,9}$ car ils sont stockés sur le même processeur 0. On pourrait calculer la matrice de rotation à partir des blocs $A_{1,1}$ et $A_{9,9}$ mais on s'aperçoit que les lignes et colonnes 1 et 9 ne sont pas stockées dans les même processeurs. Ceci pose un problème pour la diffusion de la matrice ainsi que la mise à jour. La solution pour éviter ce problème est de prendre comme couples de blocs $(A_{1,1}, A_{12,12})$, $(A_{2,2}, A_{11,11})$, ..., $(A_{6,6}, A_{7,7})$, c'est à dire les blocs aux extrémités des palindromes.

La diffusion des matrices de rotation

Dans cette optique, la diffusion des matrices de rotation ne nécessite pas un échange total. Pour les lignes de blocs 4 et 9, seuls les processeurs 0, 1, 2, 5 et 7 sont concernés. Puisque c'est le processeur 0 qui détient la matrice de rotation, il la diffuse sur sa ligne de processeurs (0,1,2) puis sur sa diagonale (0,5,7). De même pour les lignes de blocs 3 et 10, le processeur 6 diffuse sur sa ligne de processeurs (6,7,8) puis sur sa diagonale (6,4,2). Puisque ces lignes et diagonales sont disjointes, les processeurs 0 et 6 peuvent envoyer en parallèle leurs matrices de rotation, sans créer de contentions sur une grille de processeurs. De manière générale, le total des matrices peut être envoyé grâce à une suite de diffusions sur les lignes puis les colonnes et enfin les diagonales de processeurs. Si le réseau d'interconnexion est un tore 2D, cela ne crée aucune contention.

La mise à jour

Après la diffusion des matrices de rotation, chaque processeur peut mettre à jour les parties de ligne et de colonne de blocs qu'il possède. On utilise pour cela la routine de multiplication matricielle des BLAS de niveau 3. Cependant, chaque processeur peut recevoir plusieurs matrices de rotation, chacune concernant un ensemble de blocs précis. Ces ensembles sont bien entendu non disjoints. Un placement astucieux en mémoire (moyennant la transposition de certains blocs) permet d'agréger ces blocs afin que les pré et post-multiplication avec les matrices de rotation ne nécessitent aucune duplication en mémoire. De plus, l'agrégation donne une meilleur efficacité des BLAS 3.

On remarque également que la symétrie est parfaitement exploitée puisqu'on met à jour les lignes ET les colonnes en même temps.

L'intérêt de la distribution par palindromes imbriqués est donc de faciliter la diffusion des matrices de rotation, d'utiliser la symétrie pour les calculs et d'augmenter la granularité et l'efficacité de ces calculs.

Toutes ces propriétés suivent notre objectif mais ne concernent que la première itération. Voyons comment étendre cela à un balayage complet.

La translation des blocs

Au cours des itérations suivantes, nous voulons conserver cette distribution en palindromes imbriqués mais en décalant les imbrications vers le sud-est après la mise à jour. Ceci est nécessaire afin de calculer les matrices de rotation à partir de blocs différents pour chaque itération. La figure 5.9 montre les blocs choisis au cours des huit itérations nécessaires au balayage d'une matrice de taille $8.S_b \times 8.S_b$. Quand deux blocs servent à calculer une matrice de rotation, ils sont désignés par des triangles reliés par des segments. Un rond désigne les blocs dont les éléments sont annulés au cours de l'itération. Quand un rond n'est pas relié à deux triangles, cela signifie que l'annulation des éléments est produite par le calcul des rotations sur ce même bloc. Cela n'arrive que pour les blocs diagonaux lors des étapes paires.

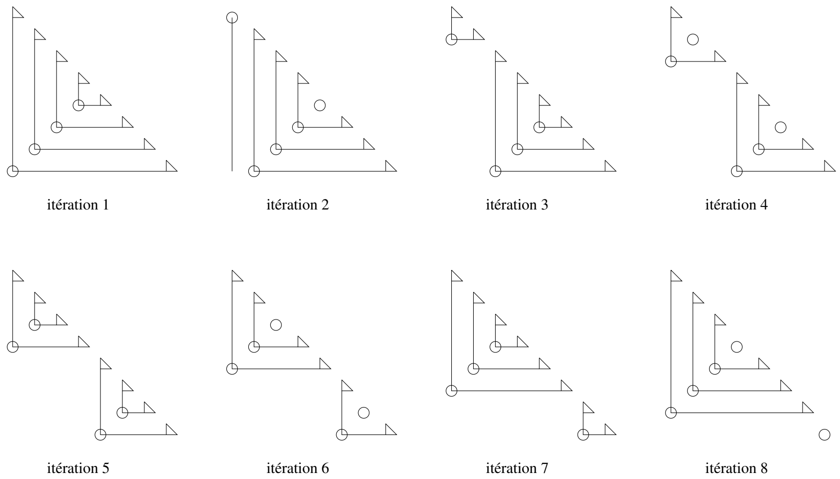

FIG. 5.9 - *Structure des palindromes au cours des itérations d'un balayage.*

A la fin des huit itérations, on remarque que tous les blocs de la matrice ont été annulés une fois et une seule. On a donc bien défini un ordonnancement effectuant un balayage ("sweep").

Pour chaque itération i d'un balayage, nous définissons le terme de quadrant (nord-ouest, sud-ouest, sud-est) comme indiqué sur la figure 5.10.

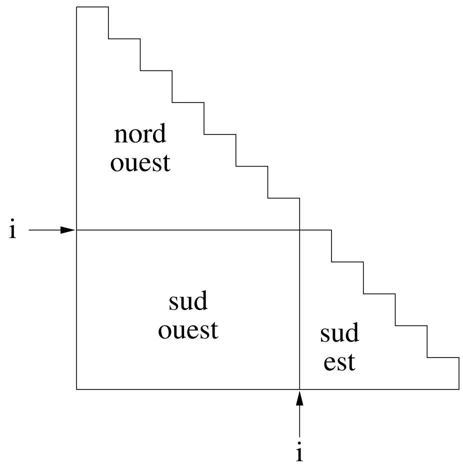

FIG. 5.10 - *Quadrants de la matrice à l'itération i d'un balayage.*

Le fait de décaler les palindromes vers le sud-est implique un schéma de communication très particulier pour translater les blocs. Ce décalage correspond à un changement de propriétaire des blocs (en terme de processeur), chaque processeur envoyant certains blocs qu'il détient à ses différents voisins. La figure 5.11 montre la distribution des blocs que l'on souhaite avoir lors de la deuxième itération. On remarque que les imbrications correspondent bien à celle de la deuxième itération de la figure 5.9. Mais un problème évident se pose, à quels processeurs appartiennent les blocs de la première colonne (en grisé)? Plus généralement, à quels processeurs appartiennent les blocs du quadrant nord-ouest?

?											
?	0										
?	1	3									
?	2	4	6								
?	0	5	7	0							
?	1	3	8	1	3						
?	2	4	6	2	4	6					
?	1	3	8	1	3	4	3				
?	0	5	7	0	1	2	1	0			
?	2	4	6	7	8	6	8	7	6		
?	1	3	4	5	3	4	3	5	4	3	
?	0	1	2	0	1	2	1	0	2	1	0

FIG. 5.11 - *Distribution à l'itération 2 pour une matrice* $12.S_b \times 12.S_b$.

Nous résolvons le problème en construisant le palindrome de la première colonne avec un

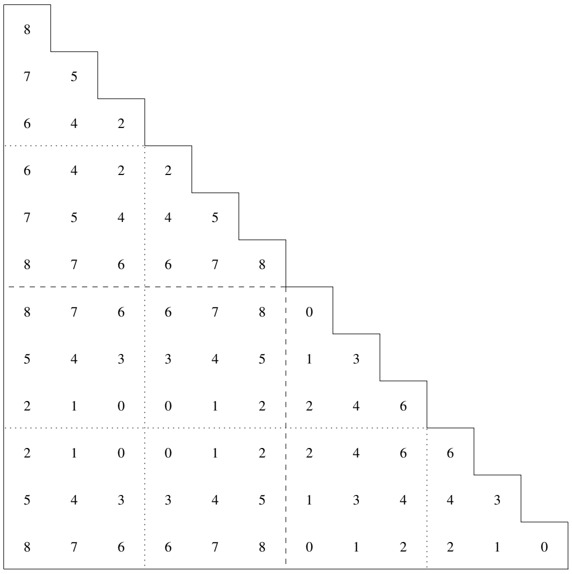

FIG. 5.12 - *Distribution à l'itération 6 pour une matrice* $12.S_b \times 12.S_b$.

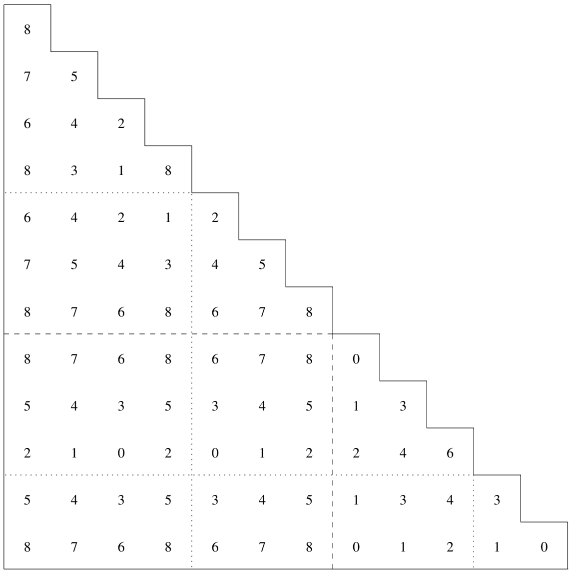

FIG. 5.13 - *Distribution à l'itération 7 pour une matrice* $12.S_b \times 12.S_b$.

ordre inversé des processeurs (8,7,...,2,1,0). Cet ordre est également utilisé pour le quadrant nord-ouest de la matrice lors des itérations suivantes. Par exemple, la figure 5.12 montre la distribution souhaitée pour l'itération 6 sur une matrice $12.S_b \times 12.S_b$ distribuée sur 9 processeurs. Le palindrome "externe" du quadrant nord-ouest est de la forme $8, 7, 6, 8, ..., 8, 6, 7, 8$ et non $0, 1, 2, ..., 2, 1, 0$ comme dans le quadrant sud-est. Quant au quadrant sud-ouest, il forme des palindromes aussi bien sur les lignes que sur les colonnes de blocs. Ce choix permet un bon équilibrage de charge et assure que les translations ne causent pas de contentions. Etudions maintenant les translations nécessaires entre les itérations 6 et 7.

D'après les figures 5.12 et 5.13, pour décaler les palindromes vers le sud-est de l'itération 6 à l'itération 7, il faut "supprimer" la ligne et la colonne 6, et "ajouter" une nouvelle ligne et colonne 3. On remarque également que certaines zones de la matrice ne bougent pas. En comparant les deux figures, on peut entièrement déterminer les communications qui ont lieu pour changer l'appartenance des blocs. Le schéma des communications réalisant cette opération est donné en figures 5.14 et 5.15. Chaque flèche correspond à l'envoi d'un bloc d'un processeur à un autre. Il y a également des recopies en mémoire de certains blocs. Les zones hachurées en traits pleins correspondent aux blocs qui ne sont pas communiqués. La zone hachurée en traits pointillés correspond à la ligne de blocs à "supprimer".

La première remarque est que toutes les communications se font entre processeurs voisins. De plus, pour chaque zone de la matrice, les communications ont lieu uniquement sur les lignes, colonnes ou diagonales de processeurs. Pour chaque zone, les communications n'engendrent donc pas de contentions sur une grille de processeurs.

La deuxième remarque découle de la précédente. Si on ordonnance les communications zones après zones, la translation des blocs n'occasionne aucune contention. En fait, les communications de plusieurs zones peuvent être effectuées en même temps. La figure 5.16 donne l'ordonnancement minimum des zones pour ne pas créer de contentions.

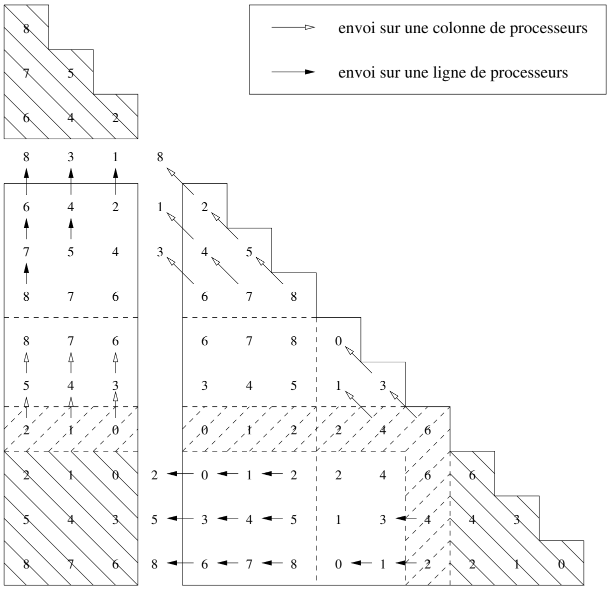

FIG. 5.14 - *Translation des blocs: communications sur lignes et colonnes de processeurs.*

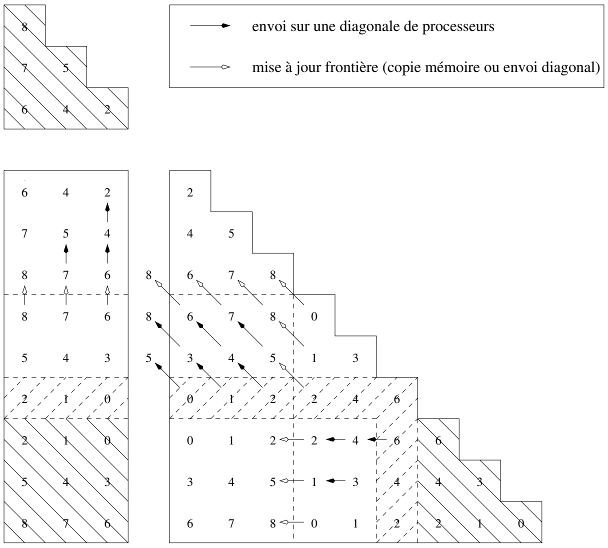

FIG. 5.15 - *Translation des blocs: communications sur les diagonales de processeurs.*

Les figures 5.14 et 5.15 concernent les communications entre les itérations 6 et 7. Cependant, quelque soit le numéro d'itération, on s'aperçoit que le schéma de communication à l'intérieur des zones (par ligne, colonnes ou diagonale de processeur) est invariant. Seul change le volume des zones donc le volume de communication pour chaque processeur. En suivant l'ordonnancement de la figure 5.16 et en calculant pour chaque zone les blocs que chaque processeur envoie, on est donc capable de générer automatiquement les communications nécessaires pour translater les blocs entre deux itérations.

L'algorithme pour un balayage

L'algorithme pour un balayage complet est donné dans le tableau 5.4. La matrice A est de taille $N \times N$, découpée en blocs de taille S_b, distribuée sur une grille de processeurs $P_r \times P_c$. Il y a donc $M = \frac{N}{S_b}$ itérations. Lors du calcul des rotations, les deux boucles d'indice j correspondent aux calculs des couples de blocs du quadrant nord-ouest ($j = 1 \ldots \frac{i}{2}$) et sud-est ($j = i \ldots \frac{M+i}{2}$) de la matrice. Lors de la mise à jour, on ne concatène pas vraiment les blocs nécessaires au calcul. Cela demanderait trop de recopies en mémoire. On préfère leur assigner une place fixe en mémoire après leur réception lors de la translation. Ce placement

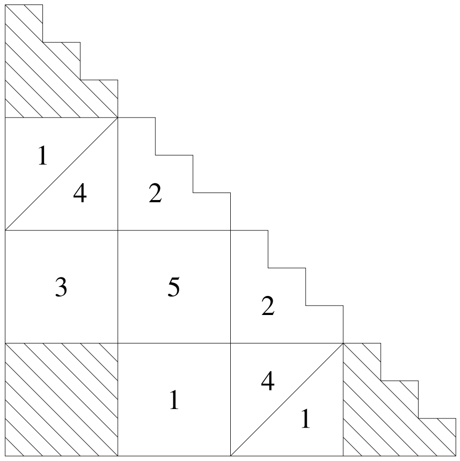

FIG. 5.16 - *Ordonnancement de phase de communication lors de la translation des blocs.*

agrège au mieux les blocs en mémoire afin que chaque mise à jour locale se fasse en une seule multiplication matricielle.

5.3.4 Temps d'exécution théorique de l'algorithme parallèle

Nous voulons calculer le temps théorique d'exécution en fonction des paramètres suivants:

- N, la taille de la matrice.
- S_b, la taille de bloc.
- $M = \frac{N}{S_b}$, le nombre de blocs diagonaux.
- $P_r \times P_c$, la taille de la grille de processeurs. On prend pour hypothèse que $P_r = P_c = r$.
- τ, la bande passante de la grille.
- k_1, le temps moyen d'une instruction flottante réalisée par un BLAS de niveau 1.
- k_3, le temps moyen d'une instruction flottante réalisée par un BLAS de niveau 3.

Comme les calculs s'effectuent en parallèle et indépendemment sur chaque processeur, nous choisissons les processeurs qui ont la plus grande charge de calcul pour trouver le temps d'exécution théorique. Suivant notre distribution, ce sont les processeurs $(0, 0)$ ou $(P_r - 1, P_c - 1)$. Pour chaque phase, nous allons calculer le temps de calcul ou de communication de ces processeurs.

(m_r, m_c) sont mes coordonnées dans la grille.
$Me = m_r.P_c + m_c$.
$M = \frac{N}{S_b}$
/* calcul des matrices de rotation et diffusion */
For i = 1 *To* M *Do*
  k=0
  For j = 1 *To* $\frac{i}{2}$ *Do*
    If $(A_{j,j} \in Me)$ *Then*
      $p = j$, $q = i - j$
      calculer la matrice de rotation $J^k_{p,q}$ à partir des blocs $A_{p,p}$ et $A_{q,q}$
      diffuse $J^k_{p,q}$.
      k++
    Else if $(\exists k | A_{k,j}$ or $A_{j,k} \in Me)$ *Then*
      $p = j$, $q = i - j$
      reçoit $J^k_{p,q}$
      k++
    Endif
  EndFor
  For j = i *To* $\frac{M+i}{2}$ *Do*
    If $(A_{j,j} \in Me)$ *Then*
      $p = j$, $q = M - j$
      calcule matrice de rotation $J^k_{p,q}$ à partir des blocs $A_{p,p}$ et $A_{q,q}$
      diffuse $J^k_{p,q}$.
      k++
    Else if $(\exists k | A_{k,j}$ or $A_{j,k} \in Me)$ *Then*
      $p = j$, $q = M - j$
      reçoit $J^k_{p,q}$
      k++
    Endif
  EndFor

  /*calcul des mises à jour */
  For j = 1 *To* k *Do*
    $A^{p,q} = \left\{ concat \left(\binom{A_{p,l}}{A_{q,l}}, \binom{A_{l,p}}{A_{l,q}}^T \right) | \forall l, A_{p,l}, A_{q,l}, A_{l,p}, A_{l,q} \in Me \right\}$
    Calcule $J^k_{p,q}.A^{p,q}.{J^k_{p,q}}^T$
  EndFor

  /*translation des blocs */
EnfFor

TAB. 5.4 - *Algorithme parallèle par bloc pour un balayage.*

Calcul des rotations

Pour la simplicité du calcul, on suppose que les matrices de rotation ont toujours une taille de $2.S_b \times 2.S_b$. On obtient donc une borne supérieure du temps d'exécution puisque les blocs diagonaux sont mis à jour à partir de matrices de rotation de taille $S_b \times S_b$. Cette remarque vaut également pour les autres phases (diffusion, mise à jour, translation).

Le temps de calcul d'une rotation est de $6.n$ instructions flottantes. Le calcul d'une matrice de rotation consiste à calculer $S_b \times S_b$ rotations, soit $6.{S_b}^3$ opérations flottantes (flops), plus $2.{S_b}^3$ flops pour accumuler (par multiplication) les rotations dans la matrice. Cela fait un total de $14.{S_b}^3$ flops.

A l'itération i, le processeur $(0,0)$ calcule $\left\lceil \frac{\lceil \frac{M-i}{2} \rceil}{r} \right\rceil$ matrices de rotation, alors que le processeur $(P_r - 1, P_c - 1)$ en calcule $\left\lceil \frac{\lceil \frac{i}{2} \rceil}{r} \right\rceil$. Au minimum, ces processeurs calculent $\left\lceil \frac{\lceil \frac{M}{4} \rceil}{r} \right\rceil$ matrices. On peut donc approximer le temps de calcul des rotations au cours d'un balayage par :

$$T_{rot} = 2.\sum_{\frac{M}{2}}^{M} \left\lceil \frac{\lceil \frac{i}{2} \rceil}{r} \right\rceil \times 14.{S_b}^3.k_1$$

$$T_{rot} \approx \left(\frac{21}{4}.\frac{N^2.S_b}{r} + \frac{21}{2}.\frac{N.{S_b}^2}{r} \right).k_1 \tag{5.1}$$

Diffusion des matrices de rotation

A l'itération i, on considère que l'on diffuse d'abord les matrices calculées à partir des blocs diagonaux $A_{1,1} \ldots A_{i-1,i-1}$ (quadrant nord-ouest) puis celles calculées à partir des blocs diagonaux $A_{i,i} \ldots A_{M,M}$ (quadrant sud-est). Effectuer ces deux phases de diffusion en même temps provoquerait des contentions car les blocs diagonaux n'appartiennent pas aux mêmes processeurs dans les deux quadrants. Cela impliquerait deux diffusions simultanées sur une ligne, colonne ou diagonale de processeurs, donc des contentions.

Nous considérons que la diffusion est effectuée en temps logarithmique. Dans notre cas, la diffusion d'une matrice de rotation sur une ligne (colonne ou diagonale) de processeur prend un temps $\tau.\lceil log_2 r \rceil.4{S_b}^2$.

- pour $A_{1,1} \ldots A_{i-1,i-1}$: si l'on se réfère aux figures 5.12 ou 5.13, on remarque que les $\left\lceil \frac{\lceil \frac{i}{2} \rceil}{r} \right\rceil$ matrices de rotation doivent être diffusées sur les lignes, colonnes et diagonales de processeurs. A l'itération i, cela prend donc un temps :

$$T^1_{diff} = \left\lceil \frac{\lceil \frac{i}{2} \rceil}{r} \right\rceil \times 3\tau.\lceil log_2 r \rceil.4{S_b}^2$$

- pour $A_{i,i} \ldots A_{M,M}$: pour diffuser les matrices dans le quadrant sud-est, il faut envoyer sur les lignes et les diagonales de processeurs. Ensuite, pour diffuser dans le quadrant sud-ouest, il faut d'abord envoyer les matrices à la dernière colonne de processeurs. Cette colonne s'occupe finalement de la diffusion. A l'itération i, cela prend un temps :

$$T^2_{diff} = \left\lceil \frac{\lceil \frac{M-i}{2} \rceil}{r} \right\rceil \times 3\tau.(\lceil log_2 r \rceil + \tfrac{1}{3}).4{S_b}^2$$

Le temps total des diffusion au cours d'un balayage est donc :

$$T_{diff} = \sum_{i=1}^{M} T^1_{diff} + T^2_{diff}$$

$$T_{diff} \approx \tau(6.\lceil log_2 r \rceil + 1)\frac{N^2}{r} \tag{5.2}$$

Mise à jour

Les $\left\lceil \frac{\lceil \frac{i}{2} \rceil}{r} \right\rceil$ matrices de rotation calculées dans le quadrant nord-ouest vont être pré et post-multipliées par une matrice de longueur $2.S_b \left(\left\lceil \frac{\lceil \frac{i}{2} \rceil}{r} \right\rceil + \left\lceil \frac{\lceil \frac{M-i}{2} \rceil}{r} \right\rceil \right)$, ce qui peut s'approximer par $\frac{M.S_b}{r}$. Ce qui donne en temps de calcul :

$$T^1_{maj} = 2.k_3 \left(\left\lceil \frac{\lceil \frac{i}{2} \rceil}{r} \right\rceil \times \frac{M}{r}.4.{S_b}^3 \right)$$

Pour les $\left\lceil \frac{\lceil \frac{M-i}{2} \rceil}{r} \right\rceil$ matrices de rotations calculées dans le quadrant sud-est, le temps de calcul est :

$$T^2_{maj} = 2.k_3. \left(\left\lceil \frac{\lceil \frac{M-i}{2} \rceil}{r} \right\rceil \times 2. \left\lceil \frac{\lceil \frac{i}{2} \rceil}{r} \right\rceil + \left(\left\lceil \frac{\lceil \frac{M-i}{2} \rceil}{r} \right\rceil^2 + \left\lceil \frac{\lceil \frac{M-i}{2} \rceil}{r} \right\rceil \right) \right) .4.{S_b}^3$$

Le temps total des mises à jour au cours d'un balayage est donc :

$$T_{maj} = \sum_{i=1}^{M} T^1_{maj} + T^2_{maj}$$

$$T_{maj} \approx \left(\frac{10}{3}.\frac{N^3}{r^2} + 2.\frac{N^2 S_b}{r}\right).k_3 \tag{5.3}$$

Translation des blocs

Pour calculer le temps total des translations au cours d'un balayage, il est nécessaire de connaître le nombre de blocs envoyés pour chaque zone par processeur. Comme indiqué précédemment, le temps total est fonction du processeur qui a le plus de blocs à envoyer. Or, dans les premières itérations, c'est le processeur $(0,0)$ et après, c'est le processeur (P_r-1, P_c-1). Pour simplifier le calcul, nous avons choisi d'additionner le temps de communication du processeur $(0,0)$ dans le quadrant sud-est au temps de communication du processeur (P_r-1, P_c-1) dans les quadrant nord-ouest et sud-ouest. Nous obtenons donc une borne supérieure du temps de translation puisque deux processeurs sont pris en compte. Le nombre de zones étant important, nous donnons juste le temps total des translations pour un balayage :

$$T_{trans} \approx \left(\frac{3}{8}.\frac{N^3}{r^2.S_b} + \frac{1}{4}.\frac{N^2}{r}\right).\tau \tag{5.4}$$

Temps d'exécution total

Pour les calculs, nous avons au total :

$$T_{calc} \approx \left(\frac{21}{4}.\frac{N^2.S_b}{r} + \frac{21}{2}.\frac{N.{S_b}^2}{r}\right).k_1 + \left(\frac{10}{3}.\frac{N^3}{r^2} + 2.\frac{N^2 S_b}{r}\right).k_3 \tag{5.5}$$

Et pour les communications :

$$T_{comm} \approx \tau(6.\lceil log_2 r\rceil + 1)\frac{N^2}{r} + \left(\frac{3}{8}.\frac{N^3}{r^2.S_b} + \frac{1}{4}.\frac{N^2}{r}\right).\tau \tag{5.6}$$

5.3.5 Discussion sur l'algorithme parallèle

Le meilleur critère de comparaison que nous ayons trouvé est l'article [35]. Les auteurs donnent un temps théorique d'exécution basé sur les même paramètres que notre modèle.

En ce qui concerne les phases de communication, nous sommes très au dessus de leurs résultats puisqu'ils obtiennent :

$$T_{comm} \approx \left(2 + \tfrac{3}{r}\right) N^2 \tau$$

Ceci s'explique par le fait que nous translations les trois-quart de la partie triangulaire inférieure entre chaque itération d'un balayage. On obtient donc un temps en $O(N^3)$.

Pour la partie calcul, nos résultats sont meilleurs puisqu'ils obtiennent :

$$T_{calc} \approx 4\tfrac{N^3}{r^2}.k_3$$

Malheureusement, sur la plupart des machines parallèles, on a $\tau >> k_3$. Cela implique que le gain au niveau du calcul ne parvient jamais à compenser la perte due aux communications.

Il existe malgré tout une solution pour palier à ce problème : le recouvrement calcul / communication. D'après la figure 5.16, il y a 5 phases de communications au cours d'une translation. Suivant la taille de la matrice et des blocs, un processeur doit effectuer un certain nombre de pré et post-multiplications pour la mise à jour. Si ce nombre est supérieur à 5, on peut entièrement recouvrir les communications dues aux translations en envoyant les blocs pendant la mise à jour. Cela implique plus de place en mémoire pour stocker les blocs reçus, mais assure une meilleure efficacité de l'algorithme.

5.4 Comparaison de Yau et Lu, et Jacobi

Du point de vue de la complexité en terme d'opération flottantes, le noyau principal de la méthode de Yau et Lu est en $6.\frac{n^2 N}{p^2}$, n étant la taille de la matrice dont il faut trouver les valeurs propres, N un paramètre qui fixe la précision du résultat et p^2 le nombre de processeurs. Sachant que N est une puissance de 2, il est pratiquement toujours supérieur à n pour que la précision des calculs soit valable. Pour référence, $N = 4096$ pour une matrice de taille 600 dans la figure 5.7. La complexité de la méthode de Jacobi est en $log_2(n)4.\frac{n^3}{p^2}$. $log_2(n)$ est une estimation du nombre de balayages nécessaires à la convergence.

A taille de matrices égales, Yau et Lu demande donc en théorie un peu plus de calculs que Jacobi. Cependant, il suffit que la convergence soit obtenue pour quelques balayages en plus par rapport à la théorie pour que Yau et Lu devienne plus performante que Jacobi. Le principal avantage de Yau et Lu par rapport à Jacobi, c'est donc le nombre borné d'opérations flottantes, dépendant de la précision des résultats que l'on souhaite.

CHAPITRE 6

Conclusion et Perspectives

"Je ne vois aucune raison de continuer à perdre notre temps avec cette affaire répugnante. Mais, naturellement, c'est le Conservateur qui a le dernier mot dans les affaires de ce genre."
J. Vance, Les chroniques de Cadwal: Araminta 2.

Nous avons présenté dans cette thèse plusieurs utilisations et optimisations autour de la bibliothèque ScaLAPACK et ses composants. Dans le premier chapitre, nous avons montré qu'il est possible d'utiliser efficacement les BLAS 3 pour des codes implémentés autrement que par passage de messages. Nous avons appliqué des techniques d'optimisation (pipeline et recouvrement calcul/communication) sur un graphe de tâche contenant des appels aux BLAS 3. Nous avons introduit la notion de modèle théorique du temps d'exécution d'un BLAS de niveau 3 et donné une méthode calculatoire pour déterminer la meilleure taille de découpage des tâches pour un pipeline. Dans le deuxième chapitre, nous avons montré qu'il est également possible de construire un modèle théorique du temps d'exécution d'une routine parallèle PBLAS ou ScaLAPACK, en se basant sur le modèle des routines séquentielles. Ce modèle permet de calculer la distribution optimale des données d'une routine mais aussi de trouver les phases de communications intéressantes à optimiser. Nous avons montré de nouveau l'utilité du pipeline et du recouvrement calcul/communication pour optimiser une routine parallèle, en l'occurence la factorisation *LU* de ScaLAPACK. Dans le troisième chapitre, nous avons donné un algorithme efficace d'ordonnancement des messages lors d'une redistribution et modélisé ainsi le temps de la redistribution de matrices distribuées par bloc cyclique. Nous avons utilisé ce modèle pour déterminer l'ensemble des meilleures redistributions entre plusieurs appels aux routines PBLAS ou ScaLAPACK. Dans le dernier chapitre, nous avons montré comment paralléliser efficacement les méthodes de Yau et Lu et Jacobi en utilisant les routines BLAS et PBLAS. Nous avons également montré une lacune de ScaLAPACK et la nécessité d'implémenter de nouvelles routines.

Voyons à présent les travaux qu'il reste à accomplir sur les sujets abordés au cours de cette thèse.

Les BLAS 3 pipelinés

Le but final est de produire une bibliothèque de création et d'exécution d'un pipeline de BLAS 3. Pour cela, nous devons encore implémenter la routine qui s'occupe d'exécuter le pipeline. Nous voulons qu'elle utilise les routines BLACS pour les communications. Cela suppose que l'utilisateur connaisse cette bibliothèque et sache appeler les routines d'initialisation correctement en fonction du graphe de tâche. La transparence d'utilisation n'est donc pas complète. Pour éviter ce problème, nous devons prévoir l'initialisation automatique des BLACS avant que le pipeline ne soit exécuté.

Au niveau des performances, l'ensemble de la méthode de calcul est satisfaisante. Cependant, nous avons vu que les tailles de découpage théoriques sont parfois très différentes des tailles expérimentales. Cela provient en partie de la précision du modèle théorique des BLAS 3 mais également de la méthode de calcul. Nous employons effectivement une dérivée pour calculer L_{opt}, ce qui privilégie les paramètres b_i, d_i, f_i de chaque BLAS 3. Or, il y a plus l'imprécision sur ces paramètres après interpolation. Pour mieux approcher les tailles expérimentales, nous pouvons tester chaque taille de découpage possible et prendre celle qui donne un temps d'exécution minimal. Cela demande beaucoup plus de calculs mais peut être profitable quand l'écart entre la théorie et l'expérimental devient trop important.

D'autre part, nous avons traité uniquement le cas de graphes linéaires, avec une seule arrête entre les tâches. Il est peut être possible d'avoir des résultats similaires pour certains type de graphes non linéaires.

Calcul des redistributions

Le travail essentiel reste sur le calcul de l'ordonnancement. Nous n'avons effectivement aucun résultat sur l'optimalité d'un ordonnancement dans le cas de la redistribution d'une matrice.

D'autre part, il reste un long travail d'expérimentation pour obtenir les paramètres pour chaque BLAS et ainsi construire un modèle du temps d'exécution des routines PBLAS et ScaLAPACK.

Calcul de valeurs propres

Dans le cas de Yau et Lu, le noyau de calcul (accélération polynomiale) a été implémenté dans le style ScaLAPACK. Il est possible, notamment pour la multiplication de matrice symétrique, de laisser de coté la portabilité et d'utiliser le recouvrement calcul / communication pour optimiser certaines partie du code. De plus, il reste la construction des éléments propres à paralléliser.

Pour la méthode de Jacobi, il serait intéressant d'implémenter notre algorithme afin de valider les résultats théoriques et comparer des résultats expérimentaux avec ceux qui sont donnés dans [35]. Nous n'avons aucune certitude sur la convergence de l'algorithme puisque nous utilisons un ordonnancement "baroque" des calculs. Les résultats expérimentaux peuvent fixer cette incertitude. Nous avons également vu qu'il est théoriquement possible d'utiliser le recouvrement calcul / communication pour supprimer le surcoût des communications. Il serait intéressant de tester expérimentalement les limites de cette hypothèse.

Perspectives

L'utilisation des bibliothèques de calcul parallèle est souvent fastidieuse. De nombreux paramètres et initialisations sont nécessaires à leur utilisation. Si les performances des bibliothèques restent inégalées, ces mêmes paramètres peuvent parfois donner des résultats décevants si l'utilisateur ne donne pas les bonnes valeurs. Un néophyte n'aura certainement aucune idée sur la taille bloc utilisée pour distribuer ses matrices, même s'il a une vague idée de ce qu'est une distribution cyclique par bloc. C'est pourquoi le calcul automatique de ces paramètres est un pas important vers la facilité d'utilisation de ScaLAPACK.

Un autre avantage du calcul automatique est qu'il assure l'efficacité d'un code produit par un compilateur paralléliseur, interfacé avec ScaLAPACK. Un compilateur HPF peut avantageusement inclure nos routines pour déterminer les meilleures distributions et redistributions. Cela implique bien entendu qu'il gère la redistribution générale, ce qui n'est malheureusement le cas d'aucun compilateur actuellement. C'est pourquoi, nous voulons inclure toutes ces routines de calcul automatique... dans une bibliothèque.

Les travaux présentés dans cette thèse peuvent également s'appliquer à d'autres domaines du calcul parallèle. On peut citer les problèmes non réguliers où l'emploi de techniques similaires à celles exposées dans le chapitre 2 peuvent être employées pour simplifier la programmation est obtenir de meilleures performances. On peut encore citer le metacomputing avec des outils comme SciLAB parallèle. Dans ce cas, la gestion de la distribution des données doit être totalement transparente à l'utilisateur. On peut donc avantageusement utiliser les résultats du chapitre 4 pour gérer les redistributions entre les calculs. De même, l'utilisation de bibliothèques parallèles dans de vraies applications commence à voir le jour (ASCI, Europort).

En conclusion, l'avenir des bibliothèques de calcul parallèle portables et efficaces semble assuré, même si de gros efforts doivent être faits pour simplifier leur utilisation, par exemple leur interfaçage avec des compilateurs paralléliseurs.

Annexe

Modèle théorique de `PDSCMM`.

Soit A, B, et C trois matrices de taille $M \times M$ distribuées sur une grille de processeurs $P_r \times P_c$ de manière cyclique par blocs de taille $S_b \times S_b$. D'après l'algorithme présenté dans le chapitre 5, les différentes phases de calcul sont :

- Calcul des blocs diagonaux :

$$\begin{aligned} T_{diag} = \lceil \frac{M}{P_c.S_b} \rceil \quad . \quad & [(a_{gemm}.S_b \times \left\lceil \frac{M}{P_r.S_b} \right\rceil + b_{gemm}).{S_b}^2 + \\ & (c_{gemm}.S_b \times \left\lceil \frac{M}{P_r.S_b} \right\rceil + d_{gemm}).S_b + \\ & (g_{gemm}.S_b \times \left\lceil \frac{M}{P_r.S_b} \right\rceil + h_{gemm})] \end{aligned}$$

  auquel il faut ajouter le temps de la somme globale pour additionner les résultats partiels :

$$T_{somme1} = f(\tau(S_b^2 \times \lceil \frac{M}{P_c.S_b} \rceil) + \beta)]$$

  avec $f = \lceil log_2 P_r \rceil$.

- Diffusion avant chaque calcul :

$$T_{diff} = (\lceil \frac{M}{P_c.S_b} \rceil - 1).[f(\tau(S_b^2 \times \lceil \frac{M}{P_r.S_b} \rceil) + \beta)]$$

  avec $f = \lceil log_2 P_c \rceil$.

– Calcul des lignes de blocs :

$$
\begin{aligned}
T_{gemm} \quad = \quad & \sum_{i=1}^{\lceil \frac{M}{S_b} \rceil -1} (a_{gemm}.S_b + b_{gemm}).(S_b \times \left\lceil \frac{M}{P_r.S_b} \right\rceil).(S_b \times \left\lceil \frac{i}{P_c} \right\rceil) + \\
& (c_{gemm}.S_b + d_{gemm}).S_b.\sqrt{\left\lceil \frac{M}{P_r.S_b} \right\rceil . \left\lceil \frac{i}{P_c} \right\rceil} + \\
& (g_{gemm}.S_b + h_{gemm})
\end{aligned}
$$

– Somme globale pour additionner les résultats partiels :

$$T_{somme2} = \sum_{i=1}^{\lceil \frac{M}{S_b} \rceil -1} .[f(\tau(S_b^2 \times \lceil \frac{i}{P_c} \rceil) + \beta)]$$

avec $f = \lceil log_2 P_r \rceil$.

– "Transposition" des blocs de la partie triangulaire supérieure :

$$T_{trans} = \sum_{i=1}^{\lceil \frac{M}{S_b} \rceil -1} .(\tau(S_b^2 \times \lceil \frac{i}{P} \rceil) + \beta)$$

avec $P = P_c = P_r$ si la grille est carrée et $P = P_r \times P_c$ si la grille est rectangulaire.

Modèle théorique de PDGEMM

En fait peu de choses changent par rapport au modèle précédent si ce n'est que la transposition n'existe pas et que les lignes de blocs calculées ont toujours la même longueur, quelle que soit le numéro d'itération. Il suffit donc de remplacer $\lceil \frac{i}{P_c} \rceil$ par $\lceil \frac{M}{P_c.S_b} \rceil$ dans les équations de T_{dgemm} et T_{somme2}.

Table des figures

1.1 Organisation de ScaLAPACK. 8

1.2 Distribution cyclique par bloc d'une matrice. 9

2.1 Exemple typique : pipeline de 5 produits matriciels. 14

2.2 Trois façons de découper un produit matriciel. 16

2.3 Exemple typique : diagramme de Gantt pour les quatre solutions de découpage. 18

2.4 Comparaison entre les solutions 3 et 4. 20

2.5 Exemple 2: Trois BLAS de niveau 3 enchaînés. 22

2.6 Algorithme de construction des meilleures solutions. 25

2.7 Résultat de l'algorithme pour l'exemple typique. 25

2.8 Temps d'exécution d'un `DGEMM` découpé en L blocs. 29

2.9 Temps d'exécution de `DGEMM` en fonction de la dimension M de A. 29

2.10 Chemin critique. 30

2.11 Temps d'exécution du premier exemple sur IBM SP2. 34

2.12 Gain pour le premier exemple sur Intel Paragon et IBM SP2. 34

2.13 Temps d'exécution du deuxième exemple sur Intel Paragon. 35

2.14 Gain pour le deuxième exemple sur Intel Paragon. 35

2.15 Temps d'exécution du troisième exemple sur Intel Paragon. 36

2.16 Gain pour le troisième exemple sur Intel Paragon. 36

2.17 Temps d'exécution de `PDGEMM` et de notre multiplication matricielle pipelinée sur 16 processeurs d'Intel Paragon. 38

2.18 Gain en % de la version pipelinée par rapport à `PDGEMM`. 38

2.19 Trace d'exécution de PDGEMM. 38

2.20 Trace d'exécution de notre multiplication matricielle pipelinée. 39

3.1 Performance (en Mflops) de la factorisation LU sur 4 processeurs pour une taille de bloc de 8 et 64, sur une Intel Paragon. 43

3.2 Premier pas de la factorisation LU par blocs. 44

3.3 Une itération de la phase 1 optimisée. 52

3.4 Comparaison du diagramme de Gantt pour la version originale et la version optimisée. 53

3.5 Résultats expérimentaux pour une grille 16×16 sur une Paragon. 55

3.6 Gain théorique et expérimental pour une grille 4×4 sur une Paragon. 55

3.7 Gain pour des grilles 8×8 et 16×16, sur une Paragon 55

3.8 Gain pour des grille 8×8 et 16×16 en fonction de la taille de la matrice locale en mémoire. 55

3.9 Résultats expérimentaux pour une grille 3×3, sur une IBM SP2. 56

3.10 Gain théorique et expérimental pour des grilles 2×2 et 4×4, sur une IBM SP2. 56

4.1 Un exemple de redistribution de matrice. 62

4.2 Un exemple de redistribution de matrice. 63

4.3 Ordonnancement caterpillar pour notre exemple. 65

4.4 Table des messages pour $M = 225$, $P = P' = 15$, $r = 3$, $r' = 5$. 66

4.5 Un ordonnancement pour $t \in [0, 6\tau[$. 67

4.6 Ordonnancement donné par l'algorithme pour $M = N = 120$, $P \times Q = 2 \times 3$, $r = s = 20$, $P' \times Q' = 2 \times 2$, $r' = s' = 30$. 73

4.7 Toutes les redistributions possibles pour l'exemple 1. 78

4.8 Cas d'une boucle. 78

5.1 Distribution bloc cyclique d'une matrice $8.S_b \times 8.S_b$ sur une grille 2×3. . . . 90

5.2 Première étape du calcul de blocs non diagonaux sur une grille 2×2. 91

5.3 Temps d'exécution théorique et expérimental sur une grille 4×4 d'une Intel Paragon. 92

5.4 Temps d'exécution expérimental sur une grille 3×3 d'une IBM SP2. 92

5.5 Gain de la version symétrique par rapport à `PDGEMM` sur Intel Paragon. . . . 93

5.6 Gain de la version symétrique par rapport à `PDGEMM` sur IBM SP2. 93

5.7 Comparaison entre le noyau de calcul parallèle et `PDSYEVX`. 94

5.8 Distribution initiale pour l'algorithme de Jacobi. 97

5.9 Structure des palindromes au cours des itérations d'un balayage. 99

5.10 Quadrants de la matrice à l'itération i d'un balayage. 100

5.11 Distribution à l'itération 2 pour une matrice $12.S_b \times 12.S_b$. 100

5.12 Distribution à l'itération 6 pour une matrice $12.S_b \times 12.S_b$. 101

5.13 Distribution à l'itération 7 pour une matrice $12.S_b \times 12.S_b$. 101

5.14 Translation des blocs: communications sur lignes et colonnes de processeurs. 102

5.15 Translation des blocs: communications sur les diagonales de processeurs. . . 102

5.16 Ordonnancement de phase de communication lors de la translation des blocs. 103

Liste des tableaux

2.1 Algorithme de multiplication matricielle($C = C + AB$). 15

2.2 Algorithmes de la multiplication matricielle pour trois différents découpages des calculs. 16

2.3 Quatre solutions pour découper les tâches de l'exemple typique. 17

2.4 Solution 4: codes pour les tâches 1 et 2. 19

2.5 Solution 4: codes pour les tâches 3, 4 et 5. 20

2.6 Solutions possibles pour découper les calculs des BLAS 3. 23

2.7 Code C contenant les valeurs des paramètres de `DGEMM`. 29

2.8 Taille de découpage optimale pour le premier exemple sur Intel Paragon. . . 34

2.9 Taille de découpage optimale pour le deuxième exemple sur Intel Paragon. . 35

2.10 Taille de découpage optimale pour le troisième exemple sur Intel Paragon. . 37

2.11 Code C pour l'initialisation et les calculs de l'exemple 1. 39

3.1 Factorisation LU par blocs en parallèle, pour une distribution cyclique par blocs. 45

3.2 Taille de bloc optimale théorique sur un système Paragon. 48

3.3 Comparaison entre tailles optimales de bloc théoriques et expérimentales pour 16 processeurs 48

3.4 Taille de bloc optimale théorique sur une IBM SP2. 49

3.5 Comparaison entre tailles optimales de bloc théoriques et expérimentales sur une IBM SP2. 49

4.1 Table des messages pour $M = N = 120$, $P \times Q = 2 \times 3$, $r = s = 20$, $P' \times Q' = 2 \times 2$, $r' = s' = 30$. 64

4.2 Routine de choix d'un message : $D_{crit} = 0$. 70

4.3 Routine de choix d'un message : $D_{crit} = 2$. 70

4.4 Algorithme principal d'ordonnancement. 71

4.5 Comparaison entre notre algorithme et la méthode du rapport [20]. 72

4.6 Cas problématiques: optimisation de l'ordonnancement de l'algorithme. . . . 72

4.7 Résultat de notre algorithme pour la redistribution de matrices de taille $M \times M$. 73

4.8 Exemple 1 de programme. 77

4.9 Solution initiale de l'exemple 1 pour une pile de Pentium. 80

4.10 Algorithme principal d'optimisation des redistributions. 81

4.11 Modification de redistribution. 81

4.12 Exemple 2 de programme. 83

4.13 Résultats de l'optimisation sur l'exemple 1 pour une pile de Pentium. 83

4.14 Résultats de l'optimisation sur l'exemple 2 pour une pile de Pentium. 84

5.1 Algorithme de construction de $C = \cos(A)$. 87

5.2 Algorithme pour la construction des v_j, $j = 0, \ldots, N-1$. 87

5.3 Algorithme de multiplication de deux matrices symétriques commutantes en parallèle. 89

5.4 Algorithme parallèle par bloc pour un balayage. 104

Liste des publications personnelles

Conférences internationales avec publication des actes

[1] F.Desprez , S. Domas , and B. Tourancheau. Optimization of the ScaLAPACK *LU* factorization routine using communication/computation overlap. In *Europar'96 Parallel Processing*. Volume 1124 of *Lecture Notes in Computer Science*, pages 3-10. Springer-Verlag, August 1996.

[2] F. Desprez and S. Domas and J. Dongarra and A. Petitet and C. Randriamaro and Y. Robert. More on Scheduling Block-Cyclic Array Redistribution. In proceedings *of 4th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98)*. Volume 1511 of *Lecture Notes in Computer Science*, pages 275-287. Springer-Verlag, 1998.

[3] S.Domas. A parallel implementation of a symmetric matrix product. In *ParCo'97*, 1997.

[4] S. Domas and F. Tisseur. Parallel implementation of a symmetric egensolveur based on Yau and Lu method. In *VecPar'96. Lecture Notes in Computer Science*, Springer-Verlag, 1996.

Conférences nationales avec publication des actes

[5] S. Domas. Une Version Parallèle de la Multiplication de deux Matrices. In *RenPAR'9*, 1997.

[6] F. Desprez and S. Domas and B. Tourancheau. Optimisation de la factorisation LU grâce aux recouvrement calcul/communication. In *RenPar'8*, 1996.

Rapports de recherche

[7] F. Desprez and S. Domas. Efficient pipelining of Level 3 BLAS Routines. Rapport de recherche INRIA, à paraître, 1998.

[8] F.Desprez , S. Domas , and B. Tourancheau. Optimisation de la factorisation LU grâce aux recouvrement calcul/communication. Technical Report RR96-17, LIP.

[9] F. Desprez and S. Domas and J-A. Gerber and C. Randriamaro. Transtool working note : Data Redistribution between ScaLAPACK routine calls. Technical Report LIP.

Bibliographie

[1] Abdelhak Lakhouaja. *Contribution à l'Algorithmique Numéeique Parallèle.* PhD thesis, Université Mohamed Premier, Faculté des Sciences, Oujda, 1996.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide, Second Edition.* SIAM, Philadelphia, PA, 1995.

[3] Z. Bai and J. Demmel. Design of a Parallel Nonsymmetric Eigenroutine Toolbox - Part I. In SIAM, editor, *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, 1993. [Long version available as UC Berkeley Computer Science report all.ps.Z via anonymous ftp from tr-ftp.cs.berkeley.edu, directory pub/tech-reports/csd/csd-92-718.].

[4] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. In *Proceedings of the International Conference on Supercomputing*, Vienna, Austria, July 1997. ACM SIGARC. Also LAPACK Working Note 111, UTK, `http://www.netlib.org/lapack/lawns`.

[5] L.S. 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.C. Whaley. *ScaLAPACK Users' Guide.* SIAM, 1997.

[6] T. Brandes and D. Greco. Realization of an HPF interface to ScaLAPACK with Redistributions. In H. Liddel, A. Colbrook, B. Hertzberger, and P. Sloot, editors, *High-Performance Computing and Networking (HPCN)*, volume 1067 of *Lectures Notes in Computer Science*, pages 834–839. Springer Verlag, 1996.

[7] S. Chakrabarti, J. Demmel, and K. Yelick. Models and Scheduling Algorithms for Mixed Data and Task Parallel Programs. *Journal of Parallel and Distributed Computing*, 47:168–184, 1997.

[8] S. Chatterjee, J. R. Gilbert, F. J. E. Long, R. Schreiber, and S.-H. Teng. Generating local addresses and communication sets for data-parallel programs. *Journal of Parallel and Distributed Computing*, 26(1):72–84, 1995.

[9] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. LAPACK Working Note: ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design and Performance. Technical Report 95, Department of Computer Science - University of Tennessee, 1995.

[10] E. Chu and A. George. Gaussian Elimination with Partial Pivoting and Load Balancing on a Multiprocessor. *Parallel Computing*, 5:65–74, 1987.

[11] M. Cosnard and D. Trystram. *Algorithmes et Architectures Parallèles*. Interéditions, 1993.

[12] M.J. Dayde, I.S. Duff, and A. Petitet. A Parallel Block Implementation of Level 3 BLAS for MIMD Vector Processors. Technical Report RT/APO/93/1, ENSEEIHT - Département Informatique N7 -I.R.I.T. Gpe Algorithmes Parallèles, 1992.

[13] J. W. Demmel and K. Stanley. The performance of finding eigenvalues and eigenvectors of dense symetric matrices on distributed memory computers. Technical Report 86, Department of Computer Science - University of Tennessee, 1994.

[14] J.W. Demmel, J.R. Gilbert, and X.S. Li. An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination, February 1997. [Available via ftp in `ftp://parcftp.xerox.com/pub/gilbert/parlu.ps.Z`].

[15] F. Desprez, S. Domas, J. Dongarra, A. Petitet, C. Randriamaro, and Y. Robert. More on scheduling block-cyclic array redistribution. In *Proc. of 4th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98)*, volume 1511 of *Lecture Notes in Computer Science*, pages 275–287. Springer-Verlag, Pittsburgh, PA, 1998.

[16] F. Desprez, S. Domas, J-A. Gerber, and C. Randriamaro. Transtool working note : Data Redistribution between ScaLAPACK routine calls. Technical Report 7, Laboratoire LIP, 1998.

[17] F. Desprez, J.J. Dongarra, and B. Tourancheau. Performance Complexity of LU Factorization with Efficient Pipelining and Overlap on a Multiprocessor. *Parallel Processing Letters*, 5-II, 1995.

[18] F. Desprez, P. Ramet, and J. Roman. Optimal Grain Size Computation for Pipelined Algorithms. In *Europar'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Science*, pages 165–172. Springer Verlag, August 1996.

[19] Frédéric Desprez, Stéphane Domas, Jack Dongarra, Antoine Petitet, Cyril Randriamaro, and Yves Robert. More on Scheduling Block-Cyclic Array Redistribution. Technical Report 98-17, LIP ENS Lyon, March 1998. To appear in the proceedings of LCR'98, LNCS, Springer Verlag.

[20] Frédéric Desprez, Jack Dongarra, Antoine Petitet, Cyril Randriamaro, and Yves Robert. Scheduling block-cyclic array redistribution. *IEEE Trans. Parallel Distributed Systems*, 1997. To appear. Available as Technical Report CS-97-349, The University of Tennessee - Knoxville. **http://www.netlib.org/lapack/lawns/** (LAWN #120).

[21] S. Domas. A parallel implementation of a symmetric matrix product. In *ParCo'97*, 1997.

[22] S. Domas, F. Desprez, and B. Tourancheau. Optimization of the ScaLAPACK LU Factorization Routine Using Communication/Computation Overlap. In *Europar'96 Parallel Processing*, volume 1124 of *Lecture Notes in Computer Science*, pages 3–10. Springer Verlag, August 1996.

[23] S. Domas and F. Tisseur. Parallel implementation of a symmetric eigensolveur based on yau and lu method. In *VecPar'96, Lecture Notes in Computer Science. Springer-Verlad*, 1996.

[24] J.J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transaction on Mathematical Software*, 16(1):1–17, 1990.

[25] J.J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An Extended Set of Fortran Basic Linear Algebra Subroutines. *ACM Transaction on Mathematical Software*, 14(1):1–17, March 1988.

[26] J.J. Dongarra, R.A. Van De Geijn, and R.C. Whaley. A User's Guide to the BLACS, March 1993.

[27] High Performance Fortran Forum. High Performance Fortran Language Specification. Technical Report 2.0, Rice University, January 1997.

[28] I. Foster. *Designing and building parallel programs.* Addison Wesley, 1995.

[29] P. Fraigniaud. Communications dans un Réseau de Processeurs. In M. Cosnard, M. Nivat, and Y. Robert, editors, *Algorithmique Parallèle*, Etude et Recherche en Informatique, chapter 9, pages 134–147. Masson, 1992.

[30] K. Gallivan, W. Jalby, U. Meier, and A. Sameh. The Impact of Hierarchical Memory Systems on Linear Algebra Design. Technical Report 625, Center for Supercomputing Research and Development - University of Illinois at Urbana-Champaign, March 1987.

[31] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine: A Users'Guide and Tutorial for Networked Parallel Computing.* The MIT Press, 1996.

[32] G.A. Geist and M.T. Heath. Matrix Factorization on a Hypercube Multiprocessor. Technical report, Oak Ridge National Laboratory, 1985.

[33] M. Gengler, S. Ubéda, and F. Desprez. *Initiation au Parallélisme : Concepts, Architectures et Algorithmes.* Manuels informatiques. Masson, 1995. 2-225-85014-3.

[34] J.A. Gerber. Distribution et Redistribution Automatiques des Données pour le Calcul Scientifique. Technical report, Laboratoire LIP, Juin 1998.

[35] D. Gimenez, R. van de Geijn, and V. Hernandez amd A. M. Vidal. Exploiting the symmetry on the Jacobi method on a mesh of processors. In *4th EUROMICRO Workshop on Parallel and Distributed Processing*, 1996.

[36] James W. Demmel and Kresimir Veselic. Jacobi's method is more accurate than QR. *SIAM : Journal Matrix Anal. Appl.*, 13(4):1204–1245, 1992.

[37] J.Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R.C. Whaley. LAPACK Working Note: A Proposal for a Set of Parallel Linear Algebra Subprograms. Technical Report 100, Department of Computer Science - University of Tennessee, 1995.

[38] B. Kågström, P. Ling, and C. Van Loan. GEMM-Based Level 3 BLAS: High Performance Model Implementations and Performance Evaluation Benchmark. Technical Report UMINF-95.18, Umeå University, October 1995.

[39] K. Kennedy, N. Nedeljkovic, and A. Sethi. Efficient address generation for block-cyclic distributions. In *1995 ACM/IEEE Supercomputing Conference.* `http://www.supercomp.org/sc95/proceedings`, 1995.

[40] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction To Parallel Computing: Design and Analysis of Algorithms.* The Benjamin /Cummings Publishing Company, Inc., 1994.

[41] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transaction on Mathematical Software*, 5:308–323, 1979.

[42] C. Lee, Y.-F. W., and T. Yang. Global Optimization for Mapping Parallel Image Processing Tasks on Distributed Memory Machines. *Journal of Parallel and Distributed Computing*, 45:29–45, 1997.

[43] P.A.R. Lorenzo, A. Muller, Y. Murakami, and B.J.N. Wylie. High Performance Fortran Interfacing to ScaLAPACK. Technical Report TR-96-13, Swiss Center for Scientific Computing (CSCS), Manno, Switzerland, 1996.

[44] Makan Pourzandi. *Etude de l'impact des recouvrements calcul/communication sur des algorithmes parallèles de calcul matriciel.* PhD thesis, Université Claude Bernard de Lyon, 1995.

[45] M. Pourzandi and B. Tourancheau. Integrated communication and computation for a Jacobi-like eigemsolver on a 2D grid. In *HPCS*, 1995.

[46] L. Prylli and B. Tourancheau. Efficient block cyclic array redistribution. *Journal of Parallel and Distributed Computing*, (45):63–72, 1997.

[47] Loïc Prylli and Bernard Tourancheau. Efficient block-cyclic data redistribution. In *EuroPar'96*, volume 1123 of *Lectures Notes in Computer Science*, pages 155–164. Springer Verlag, 1996.

[48] B.V. Purushotham, A. Basu, P.S. Kumar, and L.M. Patnaik. Performance Estimation of LU Factorisation on Message Passing Multiprocessors. *Parallel Processing Letters*, 2(1):51–60, 1992.

[49] R. Schreiber and J. Dongarra. Automatic Blocking of Nested Loops. Technical Report CS-90-108, Department of Computer Science - University of Tennessee, May 1990.

[50] B.S. Siegel and P.A. Steenkiste. Controlling Application Grain Size on a Network of Workstations. In *Supercomputing'95*, 1995. http://www.cs.cmu.edu/afs/cs/project/nectar/WWW/gnectar_papers.html.

[51] Georges-André Silber. Nestor : un système de parallélisation automatique. In *RenPar' 10 : dixièmes rencontres francophones du parallélisme*, 1998.

[52] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI the complete reference.* The MIT Press, 1996.

[53] J. M. Stichnoth, D. O'Hallaron, and T. R. Gross. Generating communication for array statements: design, implementation, and evaluation. *Journal of Parallel and Distributed Computing*, 21(1):150–159, 1994.

[54] François Tisseur. *Méthode Numériques Pour Le Calcul D'éléments Spectraux.* PhD thesis, Université Jean Monnet, St-Etienne, 1997.

[55] K. van Reeuwijk, W. Denissen, H. J. Sips, and E. M.R.M. Paalvast. An implementation framework for HPF distributed arrays on message-passing parallel computer systems. *IEEE Trans. Parallel Distributed Systems*, 7(9):897–914, 1996.

[56] A. Wakatani and M. Wolfe. Redistribution of block-cyclic data distributions using MPI. *Parallel Computing*, 21(9):1485–1490, 1995.

[57] David W. Walker and Steve W. Otto. Redistribution of block-cyclic data distributions using MPI. *Concurrency: Practice and Experience*, 8(9):707–728, 1996.

[58] Lei Wang, James M. Stichnoth, and Siddhartha Chatterjee. Runtime performance of parallel array assignment: an empirical study. In *1996 ACM/IEEE Supercomputing Conference*. `http://www.supercomp.org/sc96/proceedings`, 1996.

[59] R. Whaley and J. Dongarra. LAPACK Working Note 131: Automatically Tuned Linear Algebra Software. Technical Report CS-97-366, Department of Computer Science - University of Tennessee, 1997. `http://www.netlib.org/atlas/`.

[60] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, Cambridge MA, 1989.

Contribution à l'Ecriture et à l'Extension d'une Bibliothèque d'Algèbre Linéaire Parallèle.

Stéphane Domas.

Octobre 1998.

Résumé

Cette thèse est constituée de plusieurs travaux autour de la bibliothèque de calcul scientifique ScaLAPACK et de ses composants.

La première partie montre comment optimiser une chaîne d'appel aux routines BLAS de niveau 3, en utilisant le pipeline et le recouvrement calcul/communication. Nous donnons un modèle théorique du temps d'exécution de ces routines afin de calculer le découpage optimal pour le pipeline.

La deuxième partie est une étude de la factorisation LU de ScaLAPACK (méthode par bloc en parallèle). Un modèle théorique du temps d'exécution est donné afin de calculer la taille de bloc optimale pour la distribution de la matrice à factoriser. Ensuite, nous proposons une optimisation basée sur le pipeline et le recouvrement calcul/communication.

La troisième partie concerne la génération des redistributions entre des appels à ScaLAPACK dans un programme Fortran 77. Nous donnons d'abord un modèle du temps d'exécution de la redistribution de matrices distribuées par blocs cycliques. Pour cela, nous présentons un algorithme d'ordonnancement des messages nécessaires pour redistribuer. Ensuite, avec l'aide du modèle théorique des routines de ScaLAPACK, nous déterminons l'ensemble des redistributions qui minimise le temps d'exécution total du programme.

La dernière partie est consacrée à la parallélisation de deux méthodes pour le calcul de valeurs propres, en utilisant les routines de ScaLAPACK.

Abstract

This thesis contains several works dealing with the scientific computations library, ScaLAPACK.

The first part shows how to optimize several calls to the BLAS 3 routines executed on different processors, using pipelining and computation/communication overlap. We give a theoretical model of the execution time of these routines, in order to compute the optimal blocking for the pipeline.

The second part presents a study on LU factorization in ScaLAPACK (parallel block method). A theoretical model of the execution time is given to compute the optimal block size for the matrix distribution. We also propose an optimization based on pipelining and computation/communication overlap.

The third part deals with the redistribution generation between ScaLAPACK calls in a Fortran 77 code. First, we give an execution time model of the redistribution of block cyclic distributed matrices. To achieve this, we present an algorithm that schedules the messages needed to redistribute. Then, we use the execution time model of ScaLAPACK routines to determine the set of the redistributions that minimize the total execution time of the code.

The fourth part presents the parallelization of two methods to compute eigenvalues, using the routines proposed in ScaLAPACK.