Calcul numérique sur données de grande taille

Eddy Caron

THÈSE

présentée par

Eddy Caron

pour obtenir le titre de

DOCTEUR

de l'**Université de Picardie - Jules Verne**

spécialité : Informatique

Calcul numérique sur données de grande taille

Date de soutenance: 14 Décembre 2000

Composition du Jury :

Président	: Jean-Frédéric	Myoupo
Directeurs	: Dominique	Lazure
	: Gil	Utard
Rapporteurs	: Pierre	Manneback
	: Jean-François	Méhaut
Examinateurs	: Frédéric	Desprez
	: Vincent	Villain

Thèse préparée au sein du
Laboratoire de Recherche en Informatique d'Amiens (LaRIA)

À mon père.

Remerciements

En premier lieu, je tiens à remercier chaleureusement les membres du jury, Jean-Frédéric Myoupo, Frédéric Desprez et Vincent Villain; ainsi que Pierre Manneback et Jean-François Méhaut pour avoir accepté la lourde charge de rapporteur. Leurs avis et conseils éclairés me furent précieux.

Je profite également, de cette traditionnelle page de remerciement pour témoigner de ma gratitude et de mon profond respect pour mes deux directeurs de thèse. Dominique Lazure et Gil Utard ont su au long de ces années me guider et m'accompagner en m'offrant un aperçu de leurs nombreuses connaissances. Le support scientifique fut de grande qualité à l'image des rapports humains que nous avons pu entretenir.

Je tiens également à remercier les membres permanents du LaRIA et plus particulièrement tous ceux qui ont dû, de par leur fonction ou de par leur gentillesse, me consacrer un peu de leur temps.

Dans le même esprit, je tenais à adresser mes remerciements à toutes ces personnes rencontrées en groupe de travail, collaborations ou conférences. Au travers de ces contacts vous m'avez permis de m'intégrer dans la communauté du parallélisme. Je vous en remercie.

Le plus amical des remerciements pour les thésards du LaRIA. L'ambiance particulièrement enrichissante, à bien des égards, dans laquelle s'est déroulée cette thèse est votre œuvre. "Merci. Et puis bonne chance surtout." [7]

Je tiens également à citer les organismes qui ont fourni l'indispensable support financier : le ministère (Bourse MENRT) et le Pôle Modélisation de la région Picarde.

Tous ces remerciements ne se trouveraient pas en préambule de ce manuscrit sans trouver en amont le soutien infaillible de ma famille. Puisse cet aboutissement être aussi le vôtre.

Un remerciement tout particulier à mon père. Outre le fait que cette thèse lui est dédicacée, je tiens à souligner son aide concernant les nombreuses relectures et corrections orthographiques indispensables pour permettre la lecture de ce manuscrit.

Il me reste une personne à remercier. Une personne vraiment particulière à mes yeux. Elle a su durant toutes ces années (et cela n'est pas le moindre des travaux effectués durant cette thèse) être présente à mes côtés. Je lui dois tant. Isabelle, merci.

Table des matières

Introduction **5**

I Approche système **9**

1 Gestion de la mémoire virtuelle pour le calcul *out-of-core* **11**
1.1 Mémoire virtuelle et calcul *out-of-core* 12
1.2 Les mécanismes de gestion de la mémoire virtuelle 12
1.2.1 La pagination 13
1.2.2 Politiques de gestion de pagination 14
1.3 Influence de l'ordonnancement sur la gestion de la mémoire virtuelle 15
1.3.1 Ordonnancement vertical 15
1.3.2 Ordonnancement horizontal 15
1.3.3 Analyse des résultats 16
1.4 Les failles de la politique LRU 16
1.4.1 L'effet trou 17
1.4.2 Politique LRU et factorisation LU 18
1.5 Gestion de la mémoire virtuelle au niveau utilisateur 19
1.5.1 La librairie MMUM et le module MMUSSEL 20
1.5.2 Un exemple d'application : la factorisation LU 21
1.6 Coordonnancement pagination et multi-threading 24
1.7 Mémoire virtuelle et parallélisme 26
1.7.1 La pagination distribuée 26
1.7.2 Techniques de réordonnancement pour la pagination distribuée 27
1.7.3 Communications et l'algorithme de convolution 27
1.8 Autres approches 29
1.9 Conclusion 30

II Approche algorithmique **31**

2 Une bibliothèque de calcul *out-of-core* : `ScaLAPACK` **33**
2.1 Architecture de `ScaLAPACK` 34
2.1.1 Les `BLAS` 34
2.1.2 `LAPACK` 35
2.1.3 Les BLACS 36
2.1.4 Les `PBLAS` 36

2.2 ScaLAPACK et le prototype *out-of-core* . . . 36
2.3 Contribution au prototype *out-of-core* de ScaLAPACK . . . 38
2.4 Conclusion . . . 39

3 La factorisation LU *out-of-core* 41
3.1 La factorisation LU . . . 41
3.2 Parallélisation de la factorisation LU . . . 43
3.3 Factorisation LU parallèle *out-of-core* . . . 44
3.4 Prédiction des performances . . . 45
3.4.1 Modèle architectural . . . 45
3.4.2 Modélisation . . . 45
3.4.3 Validation expérimentale . . . 49
3.5 Analyse des surcoûts *out-of-core* . . . 49
3.5.1 Réduction du surcoût des communications . . . 49
3.5.2 Réduction du surcoût des Entrées/Sorties . . . 52
3.6 Conclusion . . . 54

4 Inversion matricielle *out-of-core* 57
4.1 Inversion matricielle par factorisation LU . . . 57
4.2 Parallélisation de l'inversion . . . 59
4.3 Inversion matricielle *out-of-core* . . . 62
4.3.1 Principe général . . . 62
4.3.2 Résolution triangulaire *out-of-core* . . . 62
4.4 Prédiction des performances . . . 63
4.4.1 Modélisation de l'inversion matricielle . . . 63
4.4.2 Validation expérimentale . . . 65
4.5 Analyse des surcoûts *out-of-core* . . . 65
4.5.1 Réduction du surcoût des communications . . . 65
4.5.2 Réduction du surcoût des Entrées/Sorties . . . 65
4.6 Optimisation par variation de la taille des super-blocs . . . 69
4.7 Conclusion . . . 69

5 Recouvrement des accès disques sur grappes 71
5.1 Un mécanisme de recouvrement des E/S . . . 71
5.2 Factorisation LU parallèle *out-of-core* avec recouvrement des accès disques . . . 72
5.2.1 Principe algorithmique . . . 72
5.2.2 Expérimentation et résultats . . . 72
5.3 Inversion matricielle parallèle *out-of-core* avec recouvrement des accès disques . . . 76
5.4 Conclusion . . . 76

III Application 79

6 Scilab$^{ooc}_{//}$ 81
6.1 Scilab . . . 82
6.1.1 L'interfaçage dans Scilab . . . 82
6.2 Scilab$_{//}$ . . . 84
6.3 Interfaçage des PBLAS dans Scilab : niveau applicatif . . . 85

6.3.1 Nouveaux types Scilab// . . . 86
6.4 Interfaçage du prototype *out-of-core* dans Scilab// : niveau applicatif . . . 87
6.4.1 Nouveaux types de Scilab//^ooc . . . 87
6.5 Optimisations pour les traitements *out-of-core* . . . 88
6.5.1 Opérateurs et fonctions purs . . . 88
6.5.2 Persistance des matrices *out-of-core* . . . 88
6.5.3 Promotions, contagions et dégradations des matrices distribuées . . . 90
6.6 Conclusions et travaux futurs . . . 91

Conclusion **93**

IV Annexe **95**

A Commandes ScaLAPACK **97**
A.1 Commandes BLACS . . . 97
A.2 Commandes PBLAS . . . 98
A.3 Commandes du prototype *out-of-core* . . . 99
A.4 Contribution au prototype *out-of-core* . . . 102

B Prototypage de Scilab **105**
B.1 Prototypage de Scilab// . . . 105
B.1.1 Type contexte . . . 105
B.1.2 Type distribution . . . 106
B.1.3 Type sous-matrice . . . 106
B.1.4 Type matrice distribuée . . . 106
B.2 Prototypage de Scilab//^ooc . . . 107
B.2.1 Type distribution *out-of-core* . . . 107
B.2.2 Type matrice distribuée *out-of-core* . . . 108

Liste des figures **109**

Bibliographie **111**

Introduction

Dans de nombreux domaines, tels que l'astronomie [80], les simulations de *crash test* automobiles [26], l'électromagnétisme [2] ou encore les modèles climatiques, et, plus généralement dans le calcul numérique hautes performances [74], le Méga-octets vaut de l'or. Dans la pratique, il est le facteur fixant la taille maximale du problème que l'utilisateur pourra traiter. L'utilisateur considérera qu'il est inconcevable de recourir au système de pagination ou aux disques du fait de l'importante chute des performances. Et pourtant, le paradoxe est là d'un côté des capacités mémoire rapide mais de petite taille et coûteuses, de l'autre des capacités mémoire à bon marché et de grande taille, plusieurs Téra-octets, mais très lente d'accès. Malgré cet handicap, on retrouve dans le calcul scientifique trois champs d'application qui sollicitent de manière importante les entrées/sorties :

- Le checkpointing ou reprise sur erreur : la reprise de longs calculs à partir d'un résultat intermédiaire peut éviter parfois de grandes pertes de temps ; exemple, reprise d'un calcul débuté il y a plusieurs jours et interrompu par un crash. Cependant pour être efficace cette méthode nécessite une sauvegarde régulière de résultats intermédiaires sur disques ce qui peut s'avérer très coûteux pour le temps de calcul.
- Les entrées/sorties en temps réel : dans certains domaines comme par exemple dans le cadre de l'imagerie médicale, il est parfois très utile de pouvoir réaliser un affichage en temps réel ; exemple, l'affichage des données obtenues à partir d'une spectroscopie par résonance magnétique nucléaire [58]).
- Le calcul *out-of-core* : nombreuses sont les applications qui nécessitent le traitement de données importantes, bien souvent au delà des capacités mémoire des machines. L'axe de recherche s'orientant vers un calcul en dehors de la mémoire est appelé calcul *out-of-core*.

Le calcul *out-of-core*

La problématique du calcul *out-of-core* n'est pas nouvelle. Les machines vectorielles avaient par le passé introduit cette notion. Les récentes avancées scientifiques dans le domaine des réseaux hauts débits et l'accroissement de la puissance des processeurs définissent ou redéfinissent, à juste titre, les accès disques comme le facteur le plus pénalisant dans le cadre d'applications parallèles de très grande taille. Le concept du calcul *out-of-core* est d'utiliser au mieux les ressources des mémoires externes. Deux approches différentes mais complémentaires existent :

- soit on utilise une technique de swap mémoire distribuée [10] qui est viable grâce aux nouvelles technologies réseaux sur des architectures type grappe. Cependant cette approche reste très coûteuse et même si elle étend les capacités mémoire globale de l'architecture cela peut s'avérer encore insuffisant.

– soit on utilise les disques des différents nœuds comme un swap disque distribué. Cette solution s'avère d'un faible coût mais encore une fois est fortement pénalisée par des accès disques très lents. Cependant, il est possible de trouver des solutions permettant de réduire les accès disques et par conséquent de limiter la chute des performances. C'est sur cet objectif que nous nous focaliserons.

Situation par rapport à l'existant

Dans le cadre du traitement de problèmes parallèles *out-of-core*, nous pouvons dissocier deux approches. La première plus générique via la compilation *out-of-core* et l'autre approche plus spécifique via l'algorithmique *out-of-core*.

- La compilation *out-of-core*, propose des solutions sans avoir à réécrire le code. Cependant comme tout outil de compilation, les performances restent loin d'un traitement spécifique. Bordawekar, Choudhary, Kennedy, Koelbel et Paleczny proposent un modèle et des stratégies pour le calcul *out-of-core* dans le cadre d'applications data-parallèles [8]. L'analyse de code est basée sur les méthodes utilisées dans le cadre de la compilation type HPF. On peut distinguer trois phases distinctes lors de la compilation :

  (a) *La phase data-parallèle*
  - Analyse du parallélisme et modèle de distribution des entrées/sorties de chaque tableau.
  - Distribution du calcul en fonction de la distribution des données.
  - Déterminer les communications nécessaires.
  - Déterminer les limites locales.

  (b) *La phase out-of-core*
  - Distribution du calcul en fonction de la distribution des E/S.
  - Déterminer les entrées/sorties nécessaires pour l'accès aux tableaux.
  - Déterminer les limites mémoire.

  (c) *La phase de génération de code*
  - Séquentialisation du code local.
  - Insérer les communications et les entrées/sorties.
  - Optimiser les communications et les entrées/sorties.

  Notons également que Peter Brezany, Thomas A. Mueck, et, Erich Schikuta, proposent dans le même esprit un langage et un compilateur permettant la gestion des accès disques dans les applications hautes performances [9].

- L'algorithmique *out-of-core* est la seconde approche de résolution *out-of-core*. Elle implique du programmeur, contrairement à la compilation *out-of-core*, la réécriture de son code. Cet inconvénient est compensé par des solutions dont la granularité est plus fine et qui permettent donc de proposer des résolutions spécifiquement adaptées aux traitements à effectuer. C'est le cas de la résolution de la transformée rapide de Fourier *out-of-core* [29] sur le système de disques parallèles de Vitter et Shriver [84]. La résolution de la factorisation LU et de l'inversion matricielle seront basées sur une approche algorithmique.

Cadre de travail

Un nouveau type de calculateurs parallèles est né. Les évolutions techniques des processeurs alliés aux progrès réalisés dans le domaine des réseaux à haut débit ont permis l'émergence et le succès des grappes de station ou *cluster*. Il est désormais possible de réaliser une machine parallèle pour un coût nettement inférieur à une architecture parallèle dédiée et offrant de bonnes performances.

Le concept de la grappe est basé sur une architecture SIMD à mémoire distribuée ; Cela consiste à relier via un réseau rapide un ensemble de stations de travail. L'ensemble des ressources est distribué : mémoire locale, disques locaux, unités de traitement local.

Les premières grappes ont été mises au point et commercialisées par *DIGITAL Equipment* à la fin des années 80 sous le nom de VAXCluster [55]. Une des premières réalisations concrètes de grappe date de 1995. Afin de simuler des phénomènes physiques et de réaliser l'acquisition de données, la NASA a mis au point une grappe basée sur 16 PCs de type 486DX4 à 100MHz avec chacun deux interfaces Ethernet à 10Mbps et des disques durs de 250Mo.

Aujourd'hui, les grappes apparaissent dans le TOP-500 des super-calculateurs les plus puissants au monde, et peuvent aller jusqu'à plusieurs dizaines de stations chacune multiprocesseurs et totalisant des Téra-octets de capacité de disque et plusieurs dizaines de Giga-octets de mémoire RAM. Les grappes se présentent donc comme une solution économique et efficace pour le traitement de problèmes *out-of-core*.

Approche Système

La première approche étudiée sera liée aux mécanismes de pagination et à son comportement face aux problèmes parallèles *out-of-core*. Nous montrerons que les programmes de pagination proposés par le système ne sont pas adaptés au calcul numérique et nous illustrerons par diverses expérimentations les facteurs responsables de cette inadéquation. Nous évoquerons l'importance des réordonnancements dans le calcul *out-of-core*. Enfin, nous présenterons une nouvelle librairie de gestion de la mémoire virtuelle au niveau utilisateur. À l'aide de cet outil, nous proposerons une optimisation de la factorisation LU *out-of-core* par gestion des mécanismes de pagination.

Approche Algorithmique

L'approche algorithmique est basée sur la librairie de calcul d'algèbre linéaire `ScaLAPACK` présentée au chapitre 2. À partir de cette librairie nous avons mené une étude des performances de la factorisation LU *out-of-core* et proposé une modélisation de cette dernière (chapitre 3). Un travail similaire a été mené sur l'inversion matricielle (chapitre 4). Nous avons montré qu'il est théoriquement possible d'effectuer une factorisation LU *out-of-core* ou une inversion matricielle *out-of-core* avec des performances proches du traitement en mémoire. Afin d'expérimenter ce résultat nous avons réalisé pour ces deux problèmes une version effectuant le recouvrement des accès disques (chapitre 5).

Intégration dans un outil de calcul numérique

Une dernière partie présentera l'intégration de ces travaux dans `Scilab`$_{//}$ un outil de calcul numérique. La participation à ce projet de l'INRIA, permet d'offrir à la librairie *out-of-core* une interface plus conviviale et accroît ainsi le champ d'application de `Scilab`$_{//}$.

Première partie

Approche système

Chapitre 1

Gestion de la mémoire virtuelle pour le calcul *out-of-core*

Sommaire

1.1 Mémoire virtuelle et calcul *out-of-core* 12
1.2 Les mécanismes de gestion de la mémoire virtuelle 12
1.2.1 La pagination 13
1.2.2 Politiques de gestion de pagination 14
1.3 Influence de l'ordonnancement sur la gestion de la mémoire virtuelle . 15
1.3.1 Ordonnancement vertical 15
1.3.2 Ordonnancement horizontal 15
1.3.3 Analyse des résultats 16
1.4 Les failles de la politique LRU 16
1.4.1 L'effet trou 17
1.4.2 Politique LRU et factorisation LU 18
1.5 Gestion de la mémoire virtuelle au niveau utilisateur 19
1.5.1 La librairie MMUM et le module MMUSSEL 20
1.5.2 Un exemple d'application : la factorisation LU 21
1.6 Coordonnancement pagination et multi-threading 24
1.7 Mémoire virtuelle et parallélisme 26
1.7.1 La pagination distribuée 26
1.7.2 Techniques de réordonnancement pour la pagination distribuée 27
1.7.3 Communications et l'algorithme de convolution 27
1.8 Autres approches 29
1.9 Conclusion 30

Pour combler les manques de mémoire vive face aux applications scientifiques, le calcul *out-of-core* offre une solution économique utilisant les ressources disponibles. En effet, le calcul *out-of-core* se propose d'exploiter les grandes capacités de la mémoire secondaire : les disques. Cependant, le temps d'accès aux informations est l'inconvénient majeur de ce type de calcul. Deux approches ont émergé dans ce cadre de travail. La première développée section 1.1 consiste à proposer des solutions de gestion de mémoire virtuelle, adaptées aux calculs *out-of-core* ; nous parlerons de mémoire virtuelle *out-of-core*. La seconde approche consiste à proposer des solutions

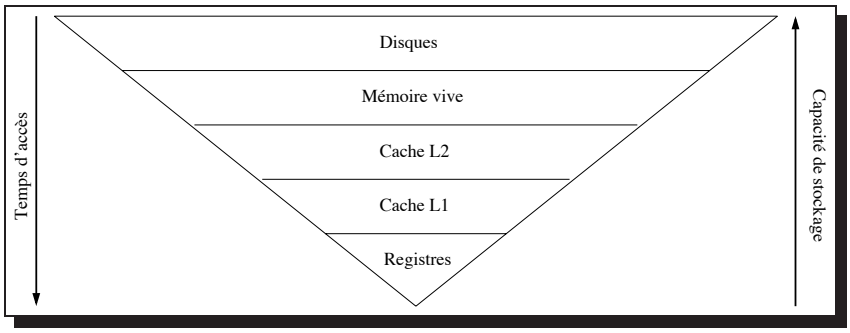

FIG. 1.1 – *Représentation de la hiérarchie mémoire.*

algorithmiques réduisant les accès disques en proposant une réorganisation des calculs ou une distribution des données adaptées (section 1.4).

1.1 Mémoire virtuelle et calcul *out-of-core*

Dans cette section nous montrerons que des techniques de réordonnancement permettent de diminuer le nombre d'accès disques dans le cas d'un traitement sur machine à mémoire partagée, puis nous proposerons une approche globale de la gestion d'une mémoire virtuelle parallèle distribuée.

L'architecture des stations de travail actuelle renferme plusieurs niveaux mémoire, dont l'ordre est déterminé par le temps d'accès moyen. On observe généralement un volume de stockage inversement proportionnel au temps d'accès : registres, cache de premier niveau, de second niveau, mémoire vive.... Les techniques de génération de code ou les bibliothèques (cf chapitre 2) permettent aujourd'hui, dans le meilleur des cas, d'exploiter au mieux un niveau avant d'accéder au niveau inférieur.

Dans le cadre de la gestion mémoire pour le calcul intensif sur grappe de stations, les performances du disque (dernier étage de la hiérarchie mémoire (cf figure 1.1)) constituent un obstacle fortement contraignant. L'utilisateur doit alors limiter la taille de ses données à la mémoire physique disponible ; la mémoire virtuelle est *de facto* bannie de l'univers du calcul scientifique, alors que les volumes de données à traiter sont toujours plus importants. Pour franchir cette frontière matérielle, il convient d'ordonnancer le calcul de façon à réduire au maximum le nombre des accès disques et/ou de recouvrir les temps d'accès disques et de calcul [54].

1.2 Les mécanismes de gestion de la mémoire virtuelle

Lorsque les ressources mémoire nécessaires à un calcul dépassent la capacité disponible, un mécanisme de gestion de la mémoire virtuelle (également appelé mécanisme de pagination disque) permet de poursuivre le traitement. Dès lors, on se retrouve dépendant du système de gestion de la mémoire virtuelle du système d'exploitation. Il est donc intéressant de rappeler les mécanismes de fonctionnement de la pagination et plus particulièrement de la pagination sous Linux sur lequel nous avons effectué nos expérimentations.

1.2.1 La pagination

L'organisation de la mémoire sous Linux est réalisée par l'intermédiaire d'un système de pagination. Le système a pour charge de maintenir une table des pages contenant diverses informations [11] comme le bit validité, l'adresse physique de la page et les informations de contrôles d'accès . À partir de cette table le système effectue la correspondance entre l'adresse virtuelle et l'adresse physique mémoire. Lorsque le processus requiert l'accès à une donnée, trois cas de figure peuvent alors survenir :

- L'adresse est transmise au système, ce dernier dispose pour celle-ci d'une adresse physique valide en mémoire. La donnée est alors transmise au processus. L'accès à la page `NPV0` figure 1.2 illustre l'**accès mémoire classique**.
- Dans un autre cas, cette adresse est référencée dans la table des pages du processus mais l'adresse physique associée est invalide. La donnée n'est donc plus disponible en mémoire. Le système informe le processeur que l'adresse virtuelle est invalide suite à un défaut de page. La figure 1.2 visualise la requête au disque par une flèche en pointillé. L'accès à la page `NPV2` illustre le **défaut de page**.
- L'adresse virtuelle transmise au système n'est pas présente dans la table des pages du processus. Le système informe ce dernier qu'un adressage invalide vient d'être provoqué. L'accès à la page `NPV9` provoque pour le processus un **adressage invalide**.

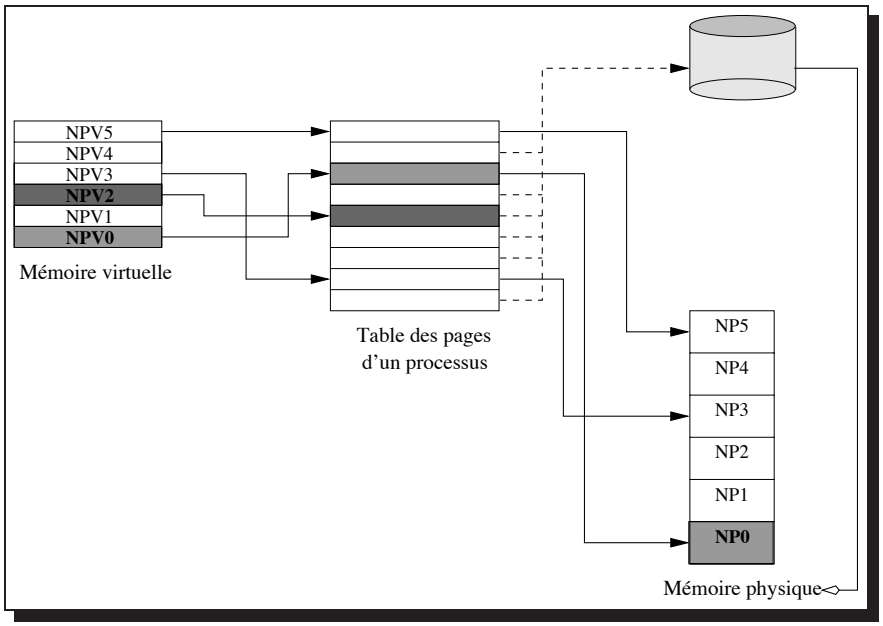

FIG. 1.2 – *Modèle de gestion mémoire avec pagination, NPV correspond au Numéro de la Page Virtuelle (adresse virtuelle) et NP au Numéro de la Page (adresse physique).*

Le coût prohibitif engendré par le mécanisme de défaut de page découle implicitement des accès disques. Pour obtenir de bonnes performances de calcul, il est indispensable de minimiser les accès disques et de minimiser ainsi les défauts de pages. L'exécution est optimale lorsque l'ensemble de travail (*working set*) est disponible en mémoire. L'ensemble de travail correspond aux pages couramment utilisées [75]. Cependant, dans le cas de traitement de grandes données, l'exécution

nécessite de parcourir un grand nombre de pages et l'utilisation du système de pagination est inévitable. On cherche alors à minimiser le nombre d'accès aux pages et de favoriser au maximum les accès à une même page. La gestion des défauts de pages est déterminée par la politique de pagination.

1.2.2 Politiques de gestion de pagination

Lorsque le système génère un défaut de page, la page doit être transférée du disque vers la mémoire. Lorsqu'il n'y a plus de page mémoire disponible, le système doit libérer l'espace mémoire nécessaire en déchargeant certaines pages mémoire. L'état de la page est pris en compte lors de cette opération. En effet, si la page a été modifiée, il est indispensable d'inscrire son contenu dans la zone de pagination avant de libérer son espace, il s'agit des *dirty pages*. Dans le cas contraire la page peut être simplement évincée de la mémoire.

Cependant, le choix de la page qui va être déchargée de la mémoire influence considérablement les performances. Comme nous le montrent les expérimentations menées en section 1.3 si l'on se retrouve à évincer la page mémoire qui va être accédée lors de l'opération suivante, les performances s'écroulent. Afin d'éviter ce genre de situation, différentes politiques de pagination existent :

- la politique **optimale** : elle consiste à décharger la page qui sera réutilisée le plus tard. Cette stratégie est optimale mais impossible à mettre en place dans le cas général puisqu'il ne nous est pas possible de prévoir les références aux futures pages.
- une politique basée sur l'algorithme **FIFO**[1] : les pages sont déchargées selon leur ordre d'arrivée. L'inconvénient majeur de cette stratégie réside dans le fait que l'on ne tient aucun compte de l'utilisation de chaque page. De plus, cette stratégie présente l'anomalie de Belady [6] c'est-à-dire que le nombre de défauts de pages peut augmenter avec la taille du cache mémoire.
- la politique de **la seconde chance** : lorsqu'une page est chargée ou utilisée on positionne son bit de la seconde chance à 1. Lorsqu'une page est candidate (selon l'ordre FIFO) au déchargement de la mémoire, on vérifie la valeur de son bit de la seconde chance. Si son bit est à 1 alors on met le bit à 0 et on n'évince pas la page ; dans le cas contraire la page est déchargée. Cette méthode peut être considérée comme une extension de la politique FIFO. Elle diminue les défauts de cette méthode sans les supprimer.
- une politique de pagination basée sur la **fréquence** appelée également **LFU**[2] : dans ce cas le système mémorise la quantité de fois où une page est accédée. Lorsque l'on doit décharger de la mémoire une page, ce sont les pages les moins fréquemment utilisées qui sont évincées en premier lieu. L'inconvénient de cette méthode est que lorsqu'une page a été fortement Utilisée, elle va s'attarder en mémoire alors que sa période d'utilisation est terminée.
- Une politique de pagination basée sur le temps appelée également algorithme de la **LRU**[3] : le système associe à chaque page un âge en fonction du dernier accès à celle-ci. Les premières pages à être évincées de la mémoire sont les plus anciennes. Reconnu comme un bon algorithme, il ne présente pas l'anomalie de Belady, mais cependant cette politique nécessite une implémentation plus complexe puisqu'elle doit connaître la date de la dernière utilisation de chaque page et gérer une pile dont le sommet contient le numéro de la page la plus récemment utilisée. Cette stratégie est celle adoptée par Linux.

1. **F**irst **I**n **F**irst **O**ut
2. **L**east **F**requently **U**sed
3. **L**east **R**ecently **U**sed.

1.3 Influence de l'ordonnancement sur la gestion de la mémoire virtuelle

Dans le cadre de la programmation (ou de la génération de code) multi-threadée ou SMP, la multiplicité des flux d'exécution et leurs mauvais ordonnancement peut induire la multiplicité des défauts de pages. Afin d'illustrer ces propos nous avons étudié le comportement d'une convolution simple *out-of-core* (moyenne des quatre voisins). La méthode classique de calcul consiste à découper la matrice en blocs, pour améliorer la localité des accès aux données.

Pour maîtriser l'ordonnancement des flux d'exécution, nous simulons une exécution sur architecture SMP dans laquelle chaque processeur est en charge du traitement d'une tranche de la matrice commune. Les processeurs sont considérés comme uniformément performants (i.e. ils restent synchronisés).

Nous avons considéré deux types de découpage de la matrice en tranches : un découpage vertical (dans le sens de la mémoire), l'autre horizontal. Pour observer le comportement du système de pagination dans ces deux cas nous avons mesuré le nombre de défauts de pages provoquant un accès disque, en fonction du nombre de flux d'exécution parallèle.

1.3.1 Ordonnancement vertical

Chaque processeur accède linéairement à la mémoire, pour la tranche verticale qui lui a été attribuée (cf. figure 1.3). L'expérience a été réalisée sur une matrice double précision de taille 4096x4096, i.e. 128 Mo sur un Celeron disposant de 32 Mo de mémoire réelle et 700 Mo de mémoire paginée. On mesure le nombre d'accès disques effectué par le système de pagination. Lorsque le nombre de flux d'exécution est trop important (de l'ordre de 256 et plus, cf. figure 1.5) le nombre d'accès disque augmente brutalement.

```
/* M matrice N*N de doubles    */
/* simulation de P processeurs */

    for(i=0;i<N/P;i++)
      for (j=0;j<N;j++)
        for (p=0;p<P;p++)
            calcul(M[j+(i+p*N/P)*N]);
```

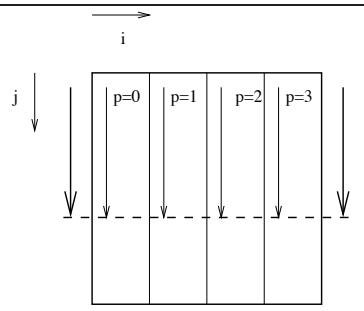

FIG. 1.3 – *Découpage vertical, le front d'accès parallèle à la mémoire est représenté en pointillé. Le sens du parcours du front mémoire est parallèle au sens du parcours de la matrice par les flux d'exécution.*

1.3.2 Ordonnancement horizontal

Dans le second cas, la matrice est découpée par bandes horizontales. Chaque processeur simulé accède colonne par colonne aux éléments de sa bande (cf. figure 1.4).

```
/* M matrice N*N de doubles    */
/* simulation de P processeurs */

    for(i=0;i<N;i++)
      for (j=0;j<N/P;j++)
        for (p=0;p<P;p++)
            calcul(M[j+p*N/P+i*N]);
```

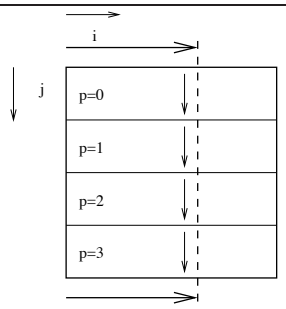

FIG. 1.4 – *Découpage horizontal, le front d'accès parallèle à la mémoire est représenté en pointillé. Le sens du parcours du front mémoire est perpendiculaire au sens du parcours de la matrice par les flux d'exécution.*

La deuxième courbe de la figure 1.5 montre l'indépendance du nombre de pages paginées par rapport au nombre de flux d'exécution. Le nombre de pagination est ici constant et correspond à la première expérimentation sans le phénomène d'augmentation.

Le temps d'exécution de la convolution pour 4096 flux d'exécution est alors 6,66 fois inférieur que pour l'ordonnancement vertical.

1.3.3 Analyse des résultats

Ces résultats s'expliquent en considérant le *front* d'accès parallèle à la mémoire. Dans le cas du découpage vertical, on constate un accès parallèle à un nombre de zones mémoire discontiguës correspondant au nombre de flux d'exécution. Dans le cas du découpage horizontal, l'accès parallèle s'effectue dans une zone contiguë de la mémoire (la colonne), et ce quel que soit le nombre de flux d'exécution. On obtient donc de meilleures performances dans cette dernière situation.

Dans le cas du traitement des données de grande taille, la génération de code pour machine à mémoire partagée ne doit pas être considérée de la même façon que la génération de code pour machine à mémoire distribuée, sous peine de mauvaise gestion mémoire. En effet, l'expérience a montré que le cache processeur est plus efficace pour des accès contigus en mémoire. Dans le cadre des calculs *out-of-core*, la problématique diffère : le coût d'un défaut de page prédomine largement celui d'un défaut de cache.

1.4 Les failles de la politique LRU

Dans cette section nous allons montrer que la gestion de la mémoire virtuelle proposée par les systèmes d'exploitation n'est pas adaptée au calcul *out-of-core*.

La section 1.2.2 décrit les principes des politiques de gestion de mémoire virtuelle. L'objectif d'une politique de pagination est de favoriser la localité temporelle, c'est-à-dire de préserver en mémoire les pages qui ont été récemment utilisées. La politique utilisée (LRU) considère que les pages qui ont le plus de chance d'être accédées par la suite sont celles qui viennent d'être récemment utilisées.

Nous allons illustrer par un exemple les répercutions de cette politique dans le cadre du calcul *out-of-core*. L'exemple présenté dans cette section met en évidence les limites de la politique LRU

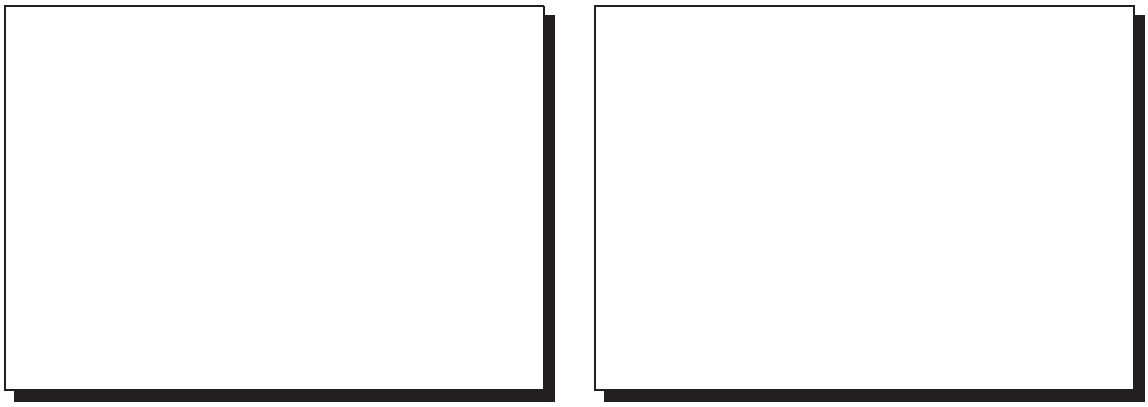

FIG. 1.5 – *Nombre de pagination en fonction du nombre de flux d'exécution : découpage par colonne et découpage par ligne.*

lors de calcul séquentiel.

1.4.1 L'effet trou

La politique LRU n'est pas adaptée aux accès linéaires en mémoire qui apparaissent souvent dans les calculs générés par des applications data-parallèles. Lorsque la taille des données est plus grande que la mémoire physique, le système de pagination est pris à défaut et l'on voit apparaître un *effet trou*. Dans cette partie nous allons définir cet *effet trou* sur un exemple simple puis nous verrons ces répercutions dans le cas d'une factorisation LU sur une matrice de grande taille.

```
real V[N];
for j = 1 to P
   for i = 1 to N
     V[i] = f(i,j);
```

FIG. 1.6 – *Accès linéaire en mémoire.*

Considérons le programme figure 1.6. Il y a P accès linéaires à un vecteur V de N éléments. Soit M le nombre de pages mémoire physique disponible en mémoire, et B la taille de la page. Considérons le cas où $\lceil N/B \rceil > M$, par exemple $N = M \times B + 1$, et $V[1..N-1]$ est initialement dans la mémoire physique. Les accès à $V[N]$ provoquent un défaut de page. Afin de répondre à la requête, le système de gestion de mémoire virtuelle va donc ôter de la mémoire physique la page qui contenait $V[1]$. Cependant, le programme va effectuer un nouveau parcours du vecteur V et par conséquent accéder à $V[1]$. Ce qui va provoquer un défaut de page. Ce défaut de page va alors évincer de la mémoire les pages qui contiennent $V[B+1]$, c'est-à-dire les pages qui seront nécessaires à la poursuite de l'algorithme. Ce phénomène se répercute à chaque page accédée. On parle alors d'*effet trou* (figure 1.7).

Le nombre d'accès disques est égal à $2 \times \lceil N/B \rceil \times (P-1) + 1$, et est dépendant de la taille de la mémoire physique M (lorsque $N > M \times B$).

L'exemple présenté pour illustrer l'effet trou est trivial. Cependant, on retrouve ce phénomène dans de nombreuses applications de calcul numérique nécessitant un accès linéaire en mémoire,

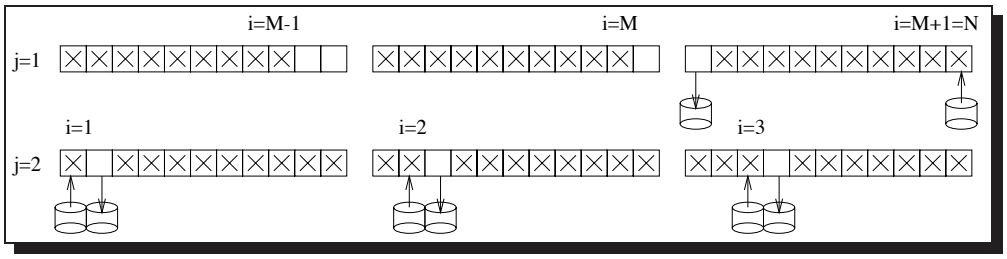

FIG. 1.7 – *Principe de l'effet trou.*

comme par exemple dans le cas de factorisation LU.

1.4.2 Politique LRU et factorisation LU

```
real A(N,N);
for j = 1 to N
  for i = j+1 to N
    for k=1 to j
       A(i,j)=A(i,j)-A(i,k)*A(k,j)
```

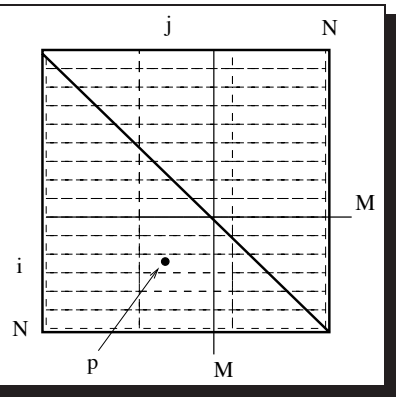

FIG. 1.8 – *Factorisation LU. La matrice est stockée par ligne en mémoire. La page p possède les éléments $A(i,j)$. La matrice d'ordre N est plus grande que M le nombre de pages disponibles en mémoire physique.*

Considérons le cas d'une factorisation LU sur une matrice d'ordre N. L'algorithme est présenté figure 1.8. Dans ce programme, chaque élément $A(i,j)_{i>j}$ est le résultat du vecteur $A(i,1:j)$ par le vecteur $A(1:j,j)$. Nous estimons le nombre de défauts de pages lorsque N est supérieur au nombre de pages physiques en mémoire M. Considérons pour cela l'itération $(i,j)_{i>j}$ et p la page qui contient $A(i,j)$. Par dépendance des données, nous montrons que la prochaine utilisation de la page p s'effectuera à l'itération $(i,j+1)$, comme le montre la figure 1.9a. Le nombre de pages accédées avant le prochain accès à la page p, est supérieur ou égal à $N-1 \geq M$. En appliquant la politique LRU, la page p n'est pas disponible suffisamment longtemps en mémoire. Cela se vérifie pour toutes les itérations où $(i,j)_{i>j}$. En fait dans ce cas, on produit un effet trou lors de l'accès à deux éléments contiguës sur la même ligne. On a donc au moins $(\frac{N}{2}-1)N$ défauts de pages.

À présent considérons les itérations $(i,j)_{i>j}$ dans le cas où $j > M$. On effectue alors $(N-j)$ accès linéaires aux pages j, associés au vecteur $A(1:j,j)$ dans la boucle interne. Le calcul de $A_{(i,j)}$ effectue trop de chargement de pages pour conserver en mémoire les premières pages utilisées pour le calcul de $A_{(i+1,j)}$ (cf figure 1.9b). On se retrouve alors dans *l'effet trou* qui va provoquer j défauts de pages. Globalement les deux effets trou impliquent un nombre de défauts de pages $p(N,M)$ qui

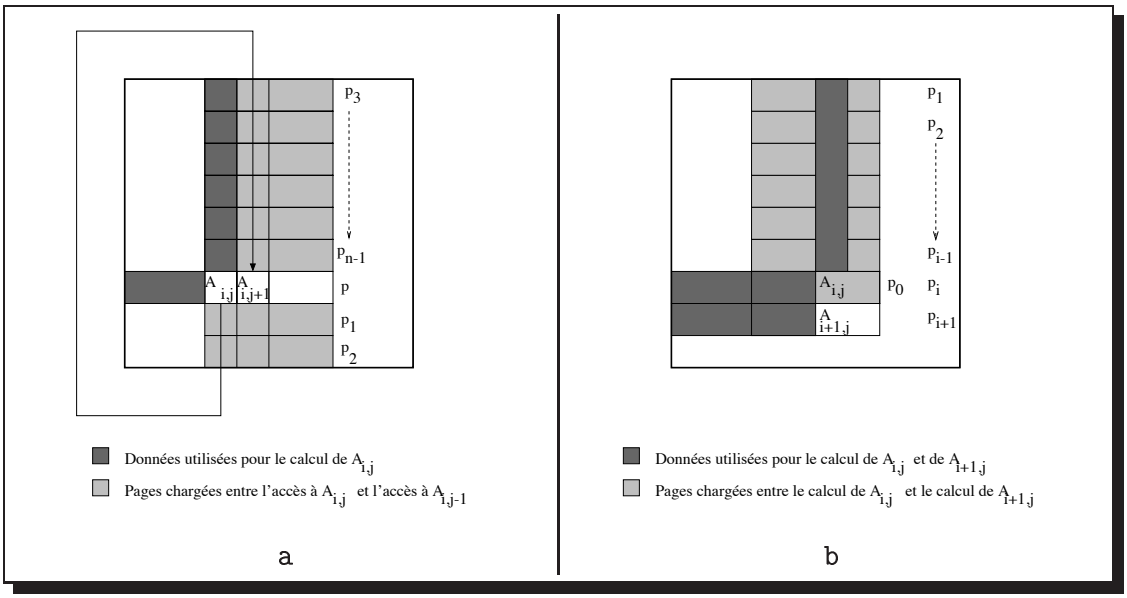

FIG. 1.9 – *L'effet trou dans la factorisation LU. Entre deux éléments de deux colonnes consécutives (a) et entre deux éléments consécutifs (b).*

est supérieur à $(\frac{N}{2} - 1)N + \sum_{j=M+1}^{j=N} j(N-j)$.

Lorsque M est constant, p(N,M) est en $O(N^3)$. Dans la section suivante nous présentons une nouvelle stratégie de pagination qui permet d'éviter l'écroulement provoqué par *l'effet trou*.

1.5 Gestion de la mémoire virtuelle au niveau utilisateur

Nous avons montré que les politiques de gestion de mémoire virtuelle, mises en œuvre dans les systèmes d'exploitation, ne sont pas adaptées au schéma d'exécution des applications data-parallèle. Deux approches sont alors envisageables :

- soit on s'oriente vers une solution algorithmique. Dans ce cas, l'utilisateur doit réécrire son code. Les performances d'un code *out-of-core* pourront être améliorées si le programmeur peut mettre à profit ses connaissances sur le système de pagination. En effet, le programmeur connaissant la politique de pagination peut, si cela lui est possible, réordonnancer l'accès aux données afin de diminuer les défauts de pages. C'est ce type de solution qui est développé dans la partie II.
- soit on s'oriente vers une solution système en modifiant la politique de gestion de la mémoire virtuelle. Malheureusement, il n'est pas aisé de modifier cette politique. Une des rares approches consiste en général à réécrire des parties du noyau du système d'exploitation qui gère la stratégie de gestion mémoire. Dans la suite de ce chapitre, nous présentons un nouvel outil qui permet l'implémentation de gestion de la mémoire virtuelle au niveau utilisateur. À l'aide de cet outil nous pourrons proposer une nouvelle stratégie de gestion de la mémoire virtuelle pour la factorisation LU.

1.5.1 La librairie MMUM et le module MMUSSEL

Comme nous venons de le démontrer les politiques de gestion de mémoire virtuelle ne sont pas adaptées aux applications numériques de grande taille. Nous avons également énoncé (section 1.2.2) la stratégie de pagination optimale. Son implémentation est, dans la réalité, irréalisable mais en offrant la possibilité à l'utilisateur de pouvoir modifier la méthode de pagination en fonction de son application ou même de redéfinir sa propre stratégie de pagination en fonction de son application, on tend à se rapprocher de la solution optimale puisque spécifique à une application donnée. C'est à partir de ce constat qu'un nouvel outil de gestion de la mémoire virtuelle au niveau utilisateur a vu le jour.

Description

L'outil de gestion de la mémoire virtuelle est composé d'une librairie appelée MMUM[4], et d'un module noyau appelé MMUSSEL[5] [13]. Le module noyau est écrit pour LINUX, mais pourrait être écrit pour n'importe quel Unix utilisant une organisation de la mémoire virtuelle de type MACH [1] (BSD, OSF/1, Hurd, ...). MMUM est une librairie consacrée aux mécanismes d'écriture en gestion de mémoire virtuelle au niveau applicatif. La librairie peut être combinée avec le module MMUSSEL. Ce module interagi avec le système d'exploitation pour effectuer la gestion des pages en mémoire physique. Une description plus approfondie du module MMUSSEL est disponible dans [32]

Cette librairie permet au programmeur de définir une nouvelle région mémoire à laquelle on peut associer une politique de pagination particulière. Une nouvelle zone mémoire est créée par l'appel à la fonction `mmum_create(size,init,nopage).`

```
(void *) mmum_create(size_t size,
                     void (*init)(void *,int),
                     void (*nopage) (void *))
```

Le paramètre `size` est la taille en *bytes* de la nouvelle région. La fonction `init` est appelée lorsqu'une référence est réalisée sur une nouvelle région. La fonction `nopage` est appelée à chaque fois qu'un défaut de page est provoqué dans cette zone mémoire.

Afin de définir une nouvelle stratégie de gestion de la mémoire virtuelle, il faut dans un premier temps écrire les deux procédures `init` et `nopage`. Plusieurs fonctions de MMUM sont conçues pour permettre la gestion de la mémoire physique avec la nouvelle région créée.

- `mmum_newpage(void *a)`: associe une nouvelle page de la mémoire physique à la page référencée par l'adresse (virtuelle) `a`. Le bit lecture/écriture de la nouvelle page est positionné à 0.
- `mmum_releasepage(void *a)`: libère la page physique associée à l'adresse (virtuelle) `a`.
- `mmum_xchg(void *a,void *b)`: échange les deux pages physiques associées respectivement aux adresses (virtuelles) `a` et `b`. Il n'est pas nécessaire d'associer une page physique à l'adresse virtuelle. Pour le moment, si l'adresse `a` est associée avec la page physique et qu'il n'y a pas de page associée à l'adresse `b`, après l'appel à `mmum_xchg(a,b)`, l'adresse `b` est associée avec la page précédemment pointée par l'adresse `a`, et aucune page n'est associée à l'adresse `a`.
- `mmum_use_rw(void *a)`: retourne la valeur du bit de lecture/écriture de la page associée à l'adresse (virtuelle) `a` et la remet à zéro.

4. Memory Management in User Mode.
5. Memory Management at USer SpacE Level.

```
void main() {
    double *v;
    int i;

    ...
    /* Le fichier A est projeté sur le vecteur v */
    v = mmum_create(N,fm_init,fm_nopage);

    for (i=0;i<N;i++) v[i]=f(v,i);
    ...
}
```

FIG. 1.10 – *Application nécessitant l'accès à un fichier projeté dans une région mémoire.*

Projeter un fichier dans la mémoire virtuelle

Pour illustrer MMUM/MUSSEL, nous présentons une gestion de la mémoire virtuelle qui permet de projeter un fichier à une nouvelle zone mémoire[6].

Le système de gestion de mémoire virtuelle est utilisé par une application qui effectue quelques calculs sur un vecteur de taille N, comme le décrit le programme en figure 1.10. Le contenu du vecteur est en fait le contenu du fichier placé dans une nouvelle zone mémoire. La variable `V` est un pointeur sur le début de cette nouvelle zone. Les itérations du programme modifient les valeurs du vecteur par la fonction `f`. Les accès au fichier sont transparents et sont effectués par le système de gestion de la mémoire virtuelle, c'est-à-dire par la fonction `fm_init` et `fm_nopage`.

Figure 1.11 on peut voir le code source des fonctions `fm_init` et `fm_nopage` qui implémente les mécanismes pour projeter le fichier.

Dans cette implémentation, il n'y a qu'une seule page présente à la fois. Par initialisation, cette page est la première page du fichier (fonction `fm_init`). Quand un défaut de page survient, c'est-à-dire qu'une autre page de la zone mémoire est accédée, la page courante est écrite sur le disque si elle a été modifiée, et la nouvelle page est chargée en mémoire (fonction `fm_nopage`).

1.5.2 Un exemple d'application : la factorisation LU

Nous allons définir une nouvelle stratégie de pagination pour la factorisation LU précédemment introduite dans la section 1.4.2. Nous proposons de modifier dynamiquement la politique de pagination en cours d'exécution. Nous définissons pour cela différentes phases de calcul.

La matrice est stockée en ligne, les colonnes sont regroupées par blocs de taille B. Durant le calcul de tous les éléments d'une colonne de bloc, N pages de la mémoire sont accédées plusieurs fois. Au lieu d'utiliser un algorithme LRU comme politique de gestion de la mémoire virtuelle, nous pouvons utiliser un algorithme LIFO (Last In First Out), avec la page $M-1$ comme première page en mémoire. Ainsi un défaut de page est provoqué pour chaque accès à un élément de $A(i,j)$ lorsque $i >= M$, cette stratégie permet d'éviter l'*effet trou* et de réduire le nombre de défauts de pages.

À chaque itération i, du fait de la localité spatiale, on a une utilisation étendue des pages associées au vecteur $A(i, 1:j)$. Lorsque ces pages sont présentes en mémoire physique, c'est-à-dire

6. Notons que dans les systèmes d'exploitation actuels la fonction `mmap` permet d'effectuer cette opération.

```
void *fm_start;                /* Adresse de la nouvelle région mémoire    */
void *fm_current;              /* Adresse virtuelle de la page courante    */
                               /* Chargement dans la région                */
int fm_size;                   /* Taille de la nouvelle région mémoire     */
int fm_fd;                     /* Descripteur de fichier                   */

void fm_init(void *v,int size) /* La nouvelle zone mémoire est initialisée */
{
  fm_start=v;                  /* Enregistre l'adresse de la nouvelle zone */
  fm_size=size;                /* Enregistre la taille de la nouvelle zone */
  fm_fd=fopen(FILENAME,"rw");  /* Ouvre le fichier à projeter dans la zone */
                               /*  mémoire                                 */

  new_page(fm_start);          /* Association d'une nouvelle page physique */
                               /*  avec le début de la zone                */

                               /* Lit la première page de la région        */
  fread(fm_start,PAGE_SIZE,1,fm_fd);
  fm_current=fm_start;         /* Enregistre l'adresse virtuelle courante  */
                               /*  de la page physique                     */
}

void fm_nopage(void *a)        /* La page adressée n'est pas présente en   */
                               /*  mémoire physique                        */
{
                               /* Vérifie si la page a été modifiée        */
                               /* Si oui, elle est inscrite sur le disque  */
  if (WRITTEN(mmum_use_rw(fm_current)))
    {
      fseek(fm_fd,(fm_current-fm_start)/PAGE_SIZE);
      fwrite(fm_start,PAGE_SIZE,1,fm_fd);
    }
                               /* La page physique associée à l'adresse    */
                               /*  virtuelle fm_current est à présent      */
                               /*  associée à l'adresse virtuelle a        */

                               /* Lit la page correspondant à l'adresse    */
                               /*  virtuelle a                             */
  fseek(fm_fd,(a-fm_start)/PAGE_SIZE);
  fread(a,PAGE_SIZE,1,fm_fd);
  fm_current=a;                /* Enregistre l'adresse virtuelle courante  */
                               /*  de la page physique                     */
}
```

FIG. 1.11 – *Fonctions associées à la nouvelle zone mémoire pour l'implémentation de la projection de fichier.*

Ordre de la mémoire	LRU	Optimisée	Efficacité
512	41 194 616	22 761 454	55%
768	7 477 734	3 024 878	40%
921	1 651 317	281 996	17%

TAB. 1.1 – *Comparaison entre la factorisation LU version LRU et la version optimisée. Expérimentation sur une matrice d'ordre 1024.*

suite à un défaut de page, le système de gestion de mémoire virtuelle ne les remplace pas.

Lorsqu'un bloc de colonnes est atteint, on doit remplacer le bloc de colonnes précédent, et plus précisément ses pages physiques par de nouvelles. Pour cela, le mécanisme de remplacement des pages est changé pour une stratégie LRU. Cela permet d'accéder aux M premiers éléments de la première colonne du nouveau bloc. Une fois ces accès effectués, on reprendra la stratégie LIFO.

Pour implémenter dynamiquement ce changement de stratégie, on réalise un programme de factorisation LU avec plusieurs directives de gestion de la mémoire virtuelle comme le montre la figure 1.12.

- `LRU(A)`: quand un défaut de page est provoqué sur la matrice A, on utilise la stratégie de remplacement des pages LRU.
- `LIFO(A)`: quand un défaut de page est provoqué sur la matrice A, on utilise la stratégie de remplacement des pages LIFO.
- `Spatial(A(i,1:j))`: les pages qui contiennent le vecteur $A(i, 1 : j)$ ne sont pas remplacées par le système de gestion de la mémoire virtuelle.
- `NoSpatial(A(i,1:j))`: les pages qui contiennent le vecteur $A(i, 1 : j)$ peuvent être remplacées par le système de gestion de la mémoire virtuelle.

```
real A(N,N);
LRU(A)
for j = 1 to N
   if (j>M) and (j mod B)=0 then LRU(A)
   for i = j+1 to N
      Spatial(A(i,1:j))
      for k=1 to j
          if (j>M) and (i=M or k=M) then
                                        LIFO(A)
          A(i,j)=A(i,j)-A(i,k)*A(k,j)
      NoSpatial(A(i,1:j))
```

FIG. 1.12 – *Factorisation LU avec les directives de pagination.*

Toutes les données dans la sous matrice `A(M:N,1:N)` sont gérées par une stratégie de remplacement de type LIFO, excepté pour la zone définie pour bénéficier de la localité spatiale que nous appellerons *zone spatiale*. Toutes les autres données sont accédées suivant la stratégie LRU. L'intérêt de cette solution est de supprimer les défauts de pages dans la sous matrice `A(1:M,1:N)`.

Lorsque l'on expérimente cette stratégie sur un PC sous Linux pour une matrice d'ordre $N = 1024$, les résultats avec différentes valeurs pour M sont présentés dans le tableau 1.1. Plus le rapport M/N est grand, plus l'optimisation proposée est efficace.

1.6 Coordonnancement pagination et multi-threading

Les expérimentations et les conclusions obtenues précédemment sont basées sur le postulat que les processeurs sont considérés comme uniformément performants. L'exécution sur architecture SMP est totalement asynchrone ce qui implique des conséquences néfastes sur la pagination. Nous proposons alors une approche multithreadée. Le parcours selon un découpage vertical est alors soumis à des perturbations significatives comme le montre la figure 1.13.

FIG. 1.13 – *Nombre de pagination en fonction du nombre de threads : découpage vertical.*

Ces phénomènes de perturbations s'expliquent par la désynchronisation des threads. Lorsqu'un défaut de page est provoqué, le thread est mis en attente et le système effectue un changement de contexte afin de permettre à un autre processus de s'exécuter. Prenons, par exemple, deux threads qui désirent accéder à une même page. Le temps de latence entre ces deux accès est dépendant de la charge mémoire occasionnée par l'exécution des autres threads qui se réalisera entre ces deux accès. Si cette charge mémoire est supérieure à la mémoire disponible la page aura été évincée de la mémoire. La désynchronisation des threads conduit à la propagation de ce phénomène et entraîne l'écroulement des performances. En fait, les accès multiples à la mémoire par les différents flux, reproduisent le phénomène de l'effet trou. Le facteur déclencheur est différent mais le résultat est similaire.

Pour améliorer les performances nous avons dans un second temps proposé une solution axée sur le *data-driving*. Dans ce cas, le flux d'exécution n'est plus déterminé par le flux de contrôle mais par le flux de données. Tant qu'une page est disponible en mémoire, on exploite ses données. On effectue en fait une distribution cyclique du flux d'exécution sur les données comme le montre la figure 1.14. Nous avons constaté que la désynchronisation qui nous pénalisait dans l'exécution précédente, n'apparaît plus. En effet, lorsqu'un défaut de page est occasionné par un thread celui-ci est mis en attente jusqu'à ce que la page soit chargée en mémoire physique. Ce laps de temps

associé à la priorité de traitement des threads est suffisant pour absorber une désynchronisation significative.

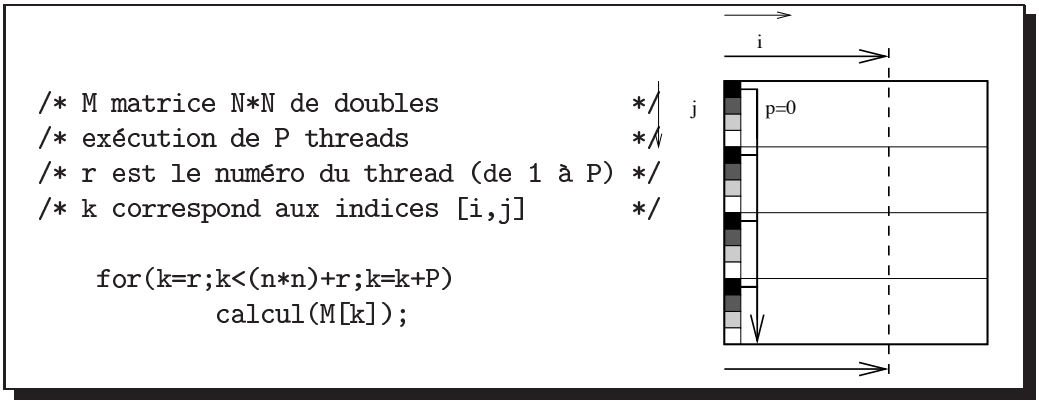

FIG. 1.14 – *Découpage data-driven, les threads sont distribués de manière cyclique sur les données. Le front d'accès parallèle à la mémoire est représenté par les pointillés. Le sens du parcours du front mémoire est cyclique mais reste globalement perpendiculaire au sens du parcours de la matrice pour les flux d'exécution.*

En adoptant cette méthode, les résultats obtenus ne subissent plus de perturbations pour un faible nombre de threads. Mais on continue à constater un écroulement des performances lorsque le nombre de threads augmente. Ce phénomène semble en contradiction avec la solution algorithmique adoptée. En effet, chaque page mémoire peut contenir jusqu'à 512 éléments de type double, soit 512 threads dans notre cas. Les threads *posix* [85] de Linux sont des threads système, mais la gestion de leur contexte d'exécution ne peut justifier une telle consommation de pagination. Afin de comprendre les causes d'un tel écroulement, nous avons mené diverses expérimentations qui ont mis en exergue un problème système.

Lorsqu'un processus déclenche un défaut de page le système prend en compte sa demande et place le processus en attente. Dans le même temps où la page est extraite du disque et chargée en mémoire physique, un ou plusieurs autres threads peuvent provoquer le même défaut de page. Si l'on analyse de près le comportement du noyau système on constate qu'une file d'attente des pages à charger est créée. Une entrée dans cette file ne tient pas compte du fait qu'une demande similaire est déjà présente. Lorsque le chargement d'une page donnée a été effectué et que le processus demandeur reprend son exécution, il met à jour la table des pages. L'information est alors disponible pour les autres processus. Cependant, les processus ayant effectué la requête sur cette page vont occasionner exactement le même accès disque. Deux pages ne peuvent être stockées dans la mémoire physique à des adresses différentes. Par conséquent, ce traitement est inutile et catastrophique pour les performances.

Ce problème système n'est pas une erreur conceptuelle puisqu'une telle optimisation ne se justifiait pas avant l'avènement des threads. En effet, chaque processus dispose de sa table de pages et par conséquent le cas évoqué précédemment correspondait à un cas pathologique très spécifique. L'émergence des threads associée à l'évolution du calcul *out-of-core* nous incite à concevoir une politique de coordonnancement pagination-threads. Les résultats obtenus en sont l'illustration directe.

1.7 Mémoire virtuelle et parallélisme

La résolution de problèmes de grande taille est souvent liée au parallélisme. L'approche parallèle permet d'introduire une nouvelle technique de gestion de la mémoire virtuelle: la pagination distribuée, mémoire distante.

1.7.1 La pagination distribuée

La pagination distribuée est également appelée pagination en mémoire distante. Le principe est d'utiliser, pour un nœud surchargé, la mémoire d'un nœud voisin afin de stocker les pages paginées [10]. Les technologies haut-débit permettent ainsi de diminuer le temps de chargement d'une page, mais ce mécanisme ne résout qu'une situation localement dégradée : le volume total des données est toujours limité à la mémoire physique répartie. Cette technique équivaut à traiter la surcharge mémoire locale par un rééquilibrage de charge.

La gestion de la pagination distribuée ainsi définie implique la prise en considération ou la reconsidération de critères comme la cohérence de cache ou encore l'importance de la localité des données. Dans le cadre d'une pagination distribuée, deux stratégies de contrôle peuvent être mises en place [30] : le contrôle centralisé ou le contrôle distribué.

Contrôle centralisé

Dans cette configuration, une unité est désignée comme serveur entre les nœuds et le (ou les) disque(s). La distribution des données est alors réalisée par le serveur selon une fonction de hachage prédéfini. Aucune cohérence de cache n'est à envisager car aucune recopie des données n'est autorisée par le serveur. Cette solution est efficace pour un nombre de nœuds peu élevé. La localité des données n'est pas un argument primordial si l'on considère un réseau à large bande passante. On peut envisager un recouvrement communication/calcul pour combler un manque de localité des données.

Contrôle distribué

Un exemple de ce type de pagination est fourni au travers de la politique de pagination développée pour une architecture PAFS (Parallel/distributed File System) [30]. Dans cette configuration, on dispose d'un ou plusieurs serveurs de traitement dont la fonction est d'interfacer les nœuds de calcul avec les serveurs de disques. Les serveurs de disques sont quant à eux la passerelle entre les nœuds de calcul et les unités disques. Dans ces conditions, deux types de distribution des données entrent en jeu, dans un premier temps la distribution des données sur les disques et dans un second temps leur distribution sur les nœuds de calcul. Afin de limiter les communications, on envisage d'attribuer à un nœud les blocs qui sont référencés par son serveur. La politique de pagination associée à ce type d'architecture est l'algorithme `LRU-Interleaved`.

Une politique de gestion de pagination distribuée : LRU-Interleaved

la pagination **LRU-Interleaved** appelée également **PG-LRU** est une extension de la politique LRU, pour les systèmes de fichiers distribués [31]. Cette stratégie intègre une structure de données capable de gérer la distribution des blocs. Cette structure donne une vision partielle des pages les moins récemment utilisées pour l'ensemble du système. Le premier critère pour décharger une page de la mémoire consiste pour un nœud demandeur à vérifier si un bloc de ce nœud est présent dans

cette liste. Si c'est le cas, la pagination restera locale à ce nœud. Dans le cas contraire, on utilise l'algorithme LRU sans se préoccuper du nœud destinataire.

1.7.2 Techniques de réordonnancement pour la pagination distribuée

L'ordonnancement des flux d'exécution sur mémoire partagée

Avec l'utilisation d'une machine à mémoire commune, l'accès à une page se trouvant sur un disque est toujours très pénalisant. Tout comme dans le cas séquentiel, pour obtenir de meilleurs performances le code devra tenir compte de ce facteur. Comme nous l'avons vu dans le cas séquentiel, la solution consistant à réordonnancer le flux d'instruction (cf section 1.1) reste applicable dans le cadre d'une mémoire distribuée.

L'ordonnancement de la pagination distribuée

Lorsque l'utilisation mémoire est très supérieure à la mémoire physique totale, chaque nœud du réseau est contraint d'utiliser la pagination sur disque. Cette charge ne peut être laissée au système : pour minimiser le nombre de pages accédées sur disque, il faut ordonnancer les calculs de façon à utiliser les données distantes dès que celles-ci sont chargées en mémoire. La synchronisation des diverses paginations doit permettre d'utiliser efficacement une mémoire virtuelle distribuée dont la taille dépasse celle de la mémoire physique. Le code exécutable doit alors prendre en charge explicitement la pagination afin d'éviter les accès disques inutiles.

1.7.3 Communications et l'algorithme de convolution

Le traitement parallèle, ne résout en rien les problèmes liés à l'effet trou. De plus, d'autres facteurs peuvent s'ajouter à son déclenchement, comme par exemple les communications. Cette partie illustre les interférences entre les communications et le système de pagination. Pour cela, nous basons notre approche sur l'algorithme de convolution. Le cœur de cet algorithme consiste à déterminer chaque élément en effectuant la moyenne de ses voisins. Le programme HPF et le code généré correspondant sont fournis figure 1.15.

```
     REAL A(N,N)
!HPF DISTRIBUTE A(BLOCK,*)
           ...
     FORALL (I=2:N-1, J=2:N-1)
             A(I,J) = (A(I,J+1)
                    + A(I+1,J)
                    + A(I,J+1)
                    + A(I-1,J))/4
           ...
```

```
Send(A(1:n,1),left)
Send(A(1:n,n),right)
Recv(A(1:n,0),left)
Recv(A(1:n,n+1),right)

FORALL (i=1:n, j=1:n)
     A(i,j)=(A(i,j+1)
            +A(i,j-1)
            +A(i+1,j)
            +A(i-1,j))/4
```

FIG. 1.15 – *À gauche, un programme HPF classique d'un calcul d'une convolution à deux dimensions. Le tableau A est distribué par blocs de colonnes. À droite, le code généré à partir du programme HPF sur une architecture à p processeurs. Les communications entre les sections du tableau sont introduites par le compilateur. La constante $n = N/p$ est l'ordre de la matrice A sur chaque processeur.*

Le tableau est distribué par blocs de colonnes sur les processeurs. Les données nécessaires pour les processeurs voisins sont envoyées ou reçues dans la première et la dernière colonne (appelée généralement zone fantôme).

Nous avons exécuté le code généré sur un réseau de PC avec 8192×4KB par page. L'ordre de la matrice sur chaque processeur est égal à 4096. La partie gauche du tableau 1.2 donne le nombre de défauts de pages obtenu dans les différentes étapes de l'algorithme. Toute la mémoire physique est utilisée dans les étapes de communication. Lors du calcul la plupart des pages envoyées et reçues sont écrites sur le disque. Ces pages sont alors chargées à nouveau lorsqu'elles sont utilisées par la phase de calcul.

	Sans tiling			Avec tiling		
	Proc. 0	**Proc. 1**	Proc. 2	Proc. 0	**Proc. 1**	Proc. 2
`Send(A[...],left)`		**9024**	9106		**10237**	10220
`Send(A[...],right)`	9218	**9547**		10085	**10155**	
`Recv(A[...],left)`		**3**	9291	27	**53**	
`Recv(A[...],right)`	9576	**5715**			**0**	31
`FORALL`	79713	**78513**	79272	72288	**64925**	72111
Total	98595	**102858**	97734	82400	**85370**	82362
Temps écoulé pour l'exécution	*720,80 s*	***771,84 s***	*777,54 s*	*571,55 s*	***536,44 s***	*576,98 s*

TAB. 1.2 – *Nombre de défauts de pages dans le cas du programme de convolution.*

Une solution pour éviter les effets de pagination inutile consiste à coupler le calcul et les communications. L'idée est de communiquer uniquement les données qui se trouvent en mémoire pour les besoins du calcul. On peut obtenir ce résultat en effectuant un découpage (***tiling***) sur la matrice (cf figure 1.16).

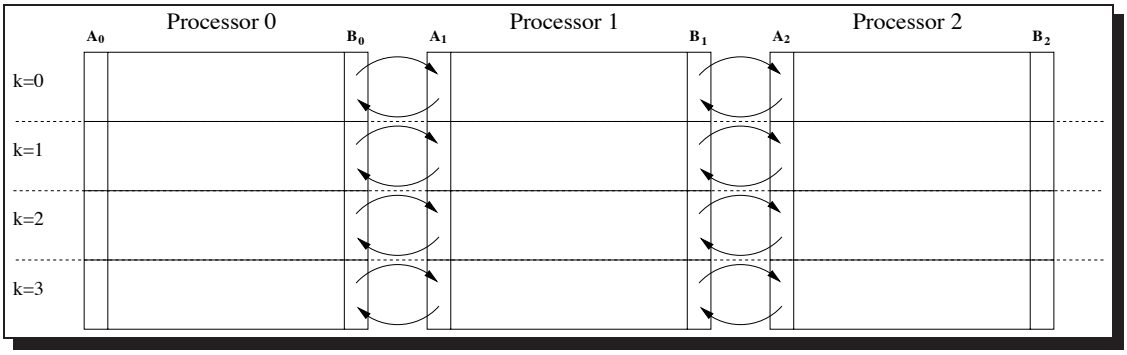

FIG. 1.16 – *Produit de convolution avec une décomposition du calcul.*

L'espace mémoire nécessaire à la phase de calcul est décomposé en tuiles. Les tuiles sont déterminées de manière à ce que le nombre de pages accédées, lors des communications et lors du calcul, soit égal au nombre de pages mémoire physique disponible. Le pseudo-code de la version de la convolution en tiling est présenté figure 1.17.

Le tableau 1.2 donne le nombre de défauts de pages pour chaque étape de l'algorithme. Il n'y a pas de défaut de page redondant ce qui permet d'obtenir un gain de 30% pour le temps d'exécution.

```
p=n/t;
FOR k=1,t
   Send(A((k-1)*p:k*p,1),left);
   Send(A((k-1)*p:k*p,n),right)
   Recv(A((k-1)*p:k*p,0),left);
   Recv(A((k-1)*p:k*p,n+1),right);

   FORALL(i=(k-1)*p:k*p,j=1:n)
      A[i][j]= (A[i][j+1]
             +A[i][j-1]
             +A[i+1][j]
             +A[i-1][j])/4;
```

FIG. 1.17 – *Convolution version décomposée.*

1.8 Autres approches

Bordawekar and Choudhary proposent différentes stratégies de communication pour les programmes *out-of-core* parallèles dans [8]. La première appelée méthode de communication *out-of-core*, où les communications et les calculs sont séparés. La seconde méthode qui a servi d'inspiration à notre solution, est appelée méthode de communication *in-core*. Dans ce cas, les communications sont associées avec le calcul de la même tuile.

Une analyse plus étendue de l'activité du système de pagination dans les programmes parallèles *out-of-core* est proposée par Marinescu et Wang dans [60, 61]. Dans ces travaux, ils ont effectué une analyse statistique de l'activité de pagination de plusieurs applications sur machines parallèles. Ils observent la non-corrélation entre la pagination des différents nœuds dans les programmes SPMD. Aucune explication n'est fournie à ce sujet. Cependant, cette non-corrélation introduit une nouvelle difficulté dans la dérivation des algorithmes de *gang scheduling* [62] (technique d'ordonnancement de différents calculs parallèles indépendants, avec de meilleures performances [69]).

La seconde partie de ce chapitre présente une nouvelle librairie qui permet de définir des algorithmes de gestion de la mémoire virtuelle au niveau de l'application. Cet outil a été utilisé pour proposer une nouvelle stratégie de gestion de la mémoire virtuelle dans le cadre de la factorisation LU. Nous obtenons des améliorations significatives en nombre de défauts de pages.

Il existe deux autres approches qui permettent de redéfinir les stratégies de gestion de la mémoire virtuelle pour les applications scientifiques de grande taille. La première, ce sont les travaux de Park, Scott and Sechrest sur une architecture Cenju-3 de NEC [71]. Sur cette architecture, le système d'exploitation dispose d'une interface utilisateur pour définir la gestion du système de pagination. Ils ont écrit des gestionnaires de mémoire virtuelle pour deux applications et obtiennent de bons résultats. La seconde approche, ce sont les travaux de Krueger, Loftesness, Vahdat and Anderson [56] qui présentent une interface orientée objet pour la gestion de la mémoire virtuelle et un outil pour étudier les accès à la mémoire virtuelle. L'idée maîtresse est de posséder assez d'informations de la part de l'outil d'étude de la mémoire virtuelle pour définir une stratégie de pagination générale pour des applications spécifiques.

Dans ces deux approches, l'utilisateur doit écrire un système de gestion de la mémoire virtuelle comme nous l'avons fait pour la factorisation LU. Ce n'est pas une tâche aisée et ne doit pas être laissé à la charge de l'utilisateur final.

Dans le cas de programmes data-parallèles *out-of-core*, il est intéressant de réutiliser les informations extraites lors de la compilation. Dans les travaux de Mowry, Demke et Krieger, à partir d'une analyse statique du code séquentiel, le compilateur peut extraire des informations concernant la localité spatiale et temporelle des données. Ces informations sont utilisées pour insérer des préchargements et des directives durant le calcul [65]. Dans les travaux de Collard et Utard [27] le compilateur détermine le découpage du calcul en accord avec la mémoire physique et l'agencement des données. L'espace de travail de chaque partie est de taille égale ou inférieure à la mémoire physique disponible; le compilateur essaie de réutiliser les données entre les parties successives.

1.9 Conclusion

Dans ce chapitre nous observons le comportement du système de gestion de la mémoire virtuelle standard du système d'exploitation lors d'une exécution type *out-of-core*, comme une factorisation LU. La première conclusion est que la politique LRU du système de gestion de la mémoire virtuelle n'est pas adaptée aux calculs numériques sur des données de grande taille. Le chapitre illustre ensuite l'importance du réordonnancement des données. À partir de ces résultats, nous proposons une optimisation, sur notre exemple de factorisation LU, avec un outil de gestion de la mémoire virtuelle au niveau utilisateur.

Nous avons ensuite étudié les solutions de pagination dans le cadre d'exécutions parallèles et souligné les nouvelles contraintes introduites. Ce qui nous a permis de constater que les communications dans les programmes data-parallèles introduisent de nouveaux threads d'accès mémoire. Ces nouveaux threads peuvent être incompatibles avec les threads d'accès mémoire utilisés pour la phase de calcul. Une solution consiste à "*tiller*" le calcul et les communications.

Deuxième partie

Approche algorithmique

Chapitre 2

Une bibliothèque de calcul *out-of-core* : ScaLAPACK

Sommaire

2.1 Architecture de ScaLAPACK 34
2.1.1 Les BLAS 34
2.1.2 LAPACK 35
2.1.3 Les BLACS 36
2.1.4 Les PBLAS 36
2.2 ScaLAPACK et le prototype *out-of-core* 36
2.3 Contribution au prototype *out-of-core* de ScaLAPACK 38
2.4 Conclusion 39

En matière de calcul scientifique la librairie ScaLAPACK[1] est devenue une référence. Elle fut co-développée au laboratoire national d'Oak Ridge et à l'université du Tennessee. L'objectif de cette librairie, est de fournir un ensemble d'outils dédiés aux calculs linéaires algébriques, qui soit à la fois puissant et portable. Les architectures ciblées sont les machines MIMD à mémoire distribuée avec passage de messages comme la Paragon d'Intel, les machines type SP de chez IBM, les T3 de chez Cray, etc. Par extension ScaLAPACK peut également être utilisé sur des grappes munies de MPI[2] [49, 48, 50, 78, 70, 45] ou PVM[3] [5, 46, 59].

Le projet a été basé sur différentes unités. Les liens entre les différentes unités sont standardisés, ce qui assure la portabilité de la librairie. L'efficacité est assurée par une conception de chaque unité, optimisée spécifiquement à l'architecture cible. Dans le même esprit il est possible de créer de nouvelles unités, permettant l'extension des outils fournis. C'est le cas du prototype *out-of-core* de ScaLAPACK.

Dans une première partie, ce chapitre présente la structure de ScaLAPACK et ses différentes unités, puis nous nous intéresserons plus particulièrement au prototype *out-of-core* avant de présenter les travaux effectués dans le cadre du développement de ce prototype.

1. **Sca**lable **L**inear **A**lgebra **PACK**age
2. **M**essage **P**assing **I**nterface
3. **P**arallel **V**irtual **M**achine

2.1 Architecture de ScaLAPACK

ScaLAPACK est composé de quatre unités majeures :

1. Les BLAS : Basic Linear Algebra Subprograms.
2. LAPACK : Linear Algebra PACKage.
3. Les BLACS : Basic Linear Algebra Communication Subprograms.
4. Les PBLAS : Parallel Basic Linear Algebra Subprograms.

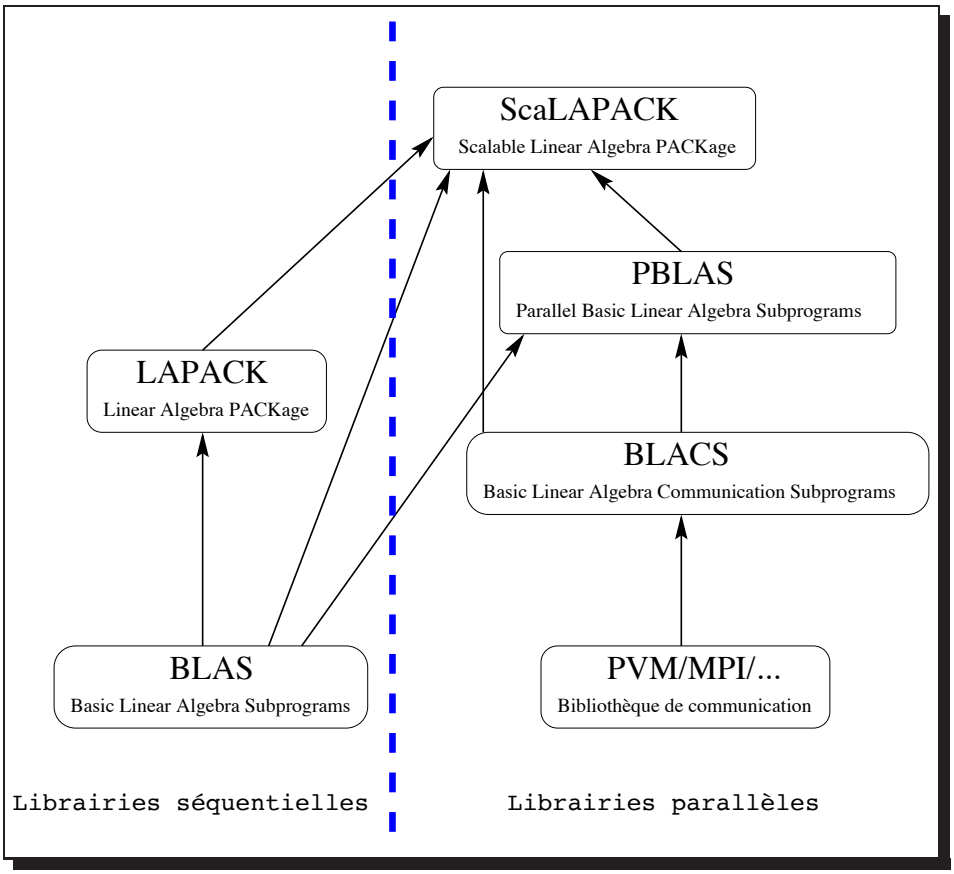

FIG. 2.1 – *La hiérarchie de ScaLAPACK.*

La figure 2.1 montre l'architecture de ScaLAPACK ainsi que les interactions entre les différentes unités.

2.1.1 Les BLAS

Les BLAS constituent l'unité dédiée au calcul. Elles contiennent les fonctions de base du calcul linéaire algébrique[4]. Ces fonctions sont divisées selon les opérateurs en trois niveaux :

- Les BLAS-1 regroupant les opérations vectorielles ou scalaires.

4. Liste des fonctions BLAS : http://www.mcc.ac.uk/software/mathematical/blas/summary.html.

- Les BLAS-2 regroupant les opérations matrices/vecteurs.
- Les BLAS-3 regroupant les opérations matrices/matrices.

Pour obtenir de meilleures performances les unités de calcul de ScaLAPACK (principalement les BLAS) exploitent la spécificité architecturale des machines cibles. Par exemple l'utilisation optimisée des différents niveaux de cache ou encore la spécificité du jeu d'instructions. Nous avons testé et évalué différentes implémentations des BLAS sur différentes architectures : une grappe de Celeron, une grappe d'Alpha et une Origin 2000. Parmi les BLAS testés on retrouve celle de Netlib, celle du projet ATLAS [87] et des BLAS spécifiques à chaque architecture. Le tableau 2.1 nous donne l'évaluation de ces différentes versions pour une multiplication matricielle effectuée à l'aide de la fonction DGEMM et une résolution triangulaire effectuée via l'appel à la fonction DTRSM.

Architecture	Type de BLAS	DGEMM (Mflops)	DTRSM (Mflops)
Alpha	Version Netlib	10.46	8.53
	Version Atlas	495.26	358.87
	Version Alpha	762.60	[a]
PC Celeron	Version Netlib	14.17	12.17
	Version Atlas	250.50	132.07
	Version PII 1.2c 11/99	283.91	158.93
	Version Xeon 1.2f 03/00	311.35	160.78
Origin 2000	Version BLAS Origin 2000	250	187

TAB. 2.1 – *Benchmark BLAS : Mflops pour l'exécution d'une multiplication matricielle (DGEMM) et d'une résolution triangulaire (DTRSM) sur une matrice double de taille* 1000×1000.

[a] version non disponible

Un choix judicieux des BLAS est donc un facteur déterminant quant aux performances globales du calcul effectué. À titre d'exemple, les BLAS dédiées aux processeurs Alpha nous permettent de réaliser une factorisation de Cholesky **80 fois plus rapidement** qu'avec la version native de Netlib. Afin de tirer le meilleur parti de la grappe d'Alpha nous avons combiné la version Alpha (qui se limite au DGEMM) avec la version Atlas qui nous donnent les meilleurs résultats pour les autres fonctions, dans l'attente d'une version complète des BLAS pour alpha.

2.1.2 LAPACK

Cette bibliothèque contient un ensemble de routines destinées à résoudre des systèmes linéaires, les problèmes de moindres carrés et les problèmes de valeurs propres. LAPACK [3] s'appuie sur LINPACK [40] et EISPACK [77]. Ces performances sont obtenues en privilégiant les appels aux BLAS et plus particulièrement à la multiplication matricielle. Les solutions algorithmiques proposées sont basées sur une approche par blocs. Ce partitionnement permet alors d'effectuer les appels aux BLAS selon ce critère. LAPACK est une librairie séquentielle, cependant les appels aux BLAS ainsi définis laissent entrevoir de manière inhérente un potentiel de parallélisation.

Par ailleurs, notons que c'est de ce *package* qu'est issu le principe de factorisation LU par bloc que nous étudierons dans le chapitre 3.

2.1.3 Les BLACS

Les BLACS [41] forment une librairie de communications, à passage de messages, dédiée à l'algèbre linéaire. L'objectif des BLACS est de fournir à l'utilisateur de ScaLAPACK, une librairie de communication indépendante de l'architecture. Quelles que soient l'architecture cible et la librairie de communication sous-jacente, l'utilisateur n'est pas contraint de redéfinir les commandes de communication ou la distribution des données. Les BLACS sont en fait une interface qui encapsule des librairies de communication de plus bas niveau comme MPI ou PVM.

Les BLACS comprennent :

- des routines de communications synchrones d'envoi et de réception d'une matrice ou d'une sous matrice d'un processeur à un autre.
- des fonctions de diffusion.
- des fonctions de réduction (sommes, maximum, minimum).
- quelques routines destinées à la consultation de paramètres (ex : récupération de la taille de la grille, nombre de processeurs, etc.).

Parmi ces fonctions, nous présentons en annexe A.1 quelques commandes de base utilisées dans nos programmes. Pour plus d'informations sur les fonctions BLACS le lecteur consultera le guide référence des BLACS, disponible à l'adresse `http://www.netlib.org/blacs/BLACS/QRef.html`.

2.1.4 Les PBLAS

Les PBLAS [23] sont la version parallèle des BLAS. Le principe est de fournir des outils parallèles pour l'algèbre linéaire en utilisant au niveau des communications le passage de messages (développé au sein des BLACS), en utilisant pour les calculs les BLAS. Une telle approche permet d'utiliser des méthodes d'algorithmes similaires à celles définies dans LAPACK. En effet, les différents appels pour réaliser un traitement par blocs peuvent être réalisés en parallèle. Les développeurs des PBLAS ont ainsi pu mettre à profit l'approche par blocs de LAPACK.

Le type de parallélisation mis en place dans les PBLAS est le parallélisme de données.

Définition 1 *Le parallélisme de données (ou data-parallélisme) est l'exécution simultanée, sur différentes unités de calcul, d'une suite d'opérations élémentaires s'appliquant à des données homogènes [57].*

Lorsque l'on introduit la notion de parallélisme de données sur des grappes on introduit implicitement la distribution des données. Dans le cadre de ScaLAPACK, les informations concernant la distribution des données sont définies dans le type **matrice distribuée**. La distribution employée est de type bloc-cyclique. Cette distribution offre de bonnes propriétés dans le cadre du calcul linéaire et réalise l'équilibrage de charge comme il est montré dans [39].

En utilisant les notations adoptées par ScaLAPACK on parle de la distribution d'une matrice de taille $M \times N$ sur des blocs de taille $MB \times NB$ (cf. figure 2.2).

À noter que des projets transversaux de parallélisation des BLAS ont également été développés dans [25].

2.2 ScaLAPACK et le prototype *out-of-core*

Le prototype out-of-core

À partir des unités précédemment définies ScaLAPACK offre un ensemble de routines permettant de réaliser certaines factorisations (LU, QR, Cholesky) et des résolutions de systèmes linéaires sur

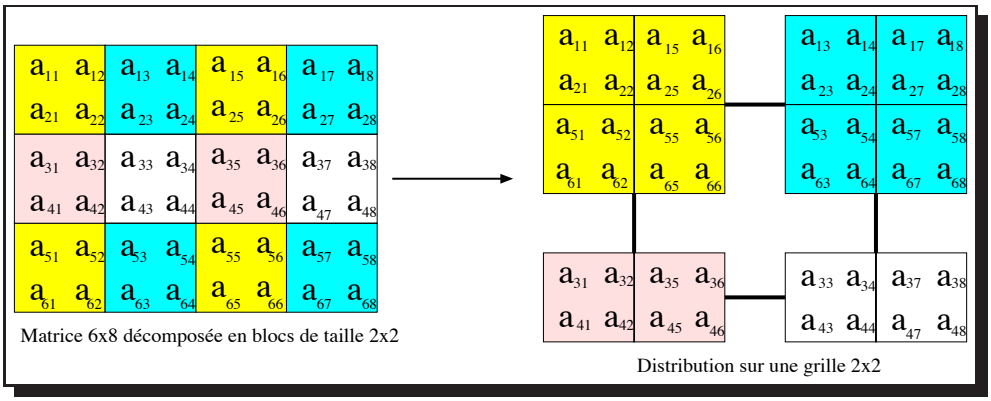

FIG. 2.2 – *Distribution blocs cycliques. M=6, N=8 et MB=NB=2.*

un large ensemble de matrices (générales, bandes, tridiagonales, hermitiennes, triangulaires). Des extensions de développement de `ScaLAPACK` ont permis la mise au point de différents prototypes pour permettre la résolution de problèmes spécifiques. Citons le prototype de la superLU [35, 36] permettant d'effectuer une factorisation LU sur des matrices creuses ou encore le prototype de l'algorithme SDC[5] [73] permettant de résoudre les problèmes de valeurs propres. Notre intérêt s'est porté sur le prototype *out-of-core* [33]. Ce prototype permet d'effectuer différents types de factorisation sur des matrices de taille supérieure à la mémoire. Les factorisations réalisées dans ce cadre sont la factorisation de Cholesky, la factorisation QR et la factorisation LU. À ces fins, une série de fonctions destinées à la gestion de fichiers sur disque ont été développés (l'annexe A.3 présente quelques fonctions issues de ce prototype).

Hiérarchie mémoire

Le prototype *out-of-core*, envisage un niveau de la hiérarchie mémoire supplémentaire avec le **super-bloc.**

Définition 2 *Un super-bloc est une sous-matrice dont la taille est suffisamment petite pour tenir en mémoire.*

Les différents niveaux hiérarchiques ainsi introduits dans `ScaLAPACK` sont assimilables à la hiérarchie mémoire. On peut effectivement faire une analogie entre la hiérarchie mémoire et les différents niveaux de décomposition présents dans `ScaLAPACK` (cf. figure 2.3).

Les matrices *out-of-core* peuvent être associées aux disques. En effet, de par leur taille les matrices sont stockées en mémoire secondaire.

La mémoire est quant à elle assimilée au super-bloc dans le cas *out-of-core* et à la matrice dans le cas *in-core*. En fait, on associe ici la mémoire à la matrice (ou sous matrice) qui va être présente en mémoire lors du calcul en cours.

Une décomposition en blocs, de tailles adaptées, permet de mettre à profit les propriétés du cache (cf la section 2.1.1). On peut associer le niveau de la hiérarchie mémoire, représenté par le cache, avec la taille du bloc.

5. **S**pectral **D**ivide and **C**onquer

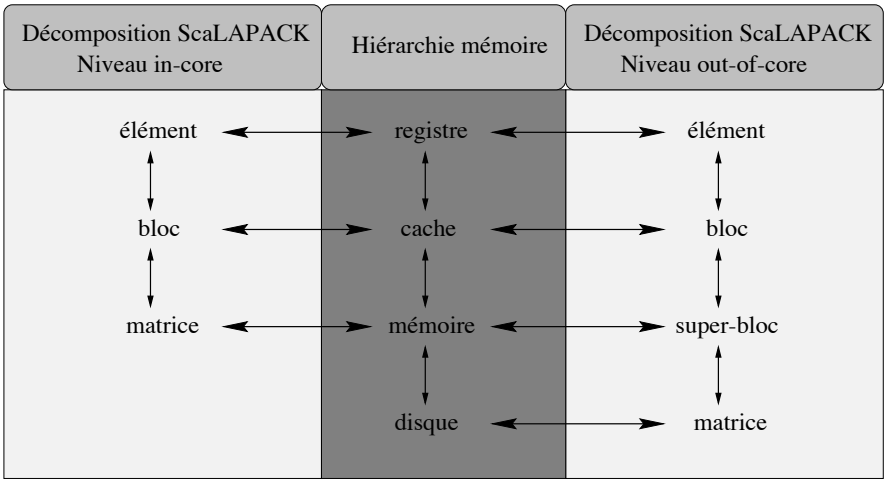

FIG. 2.3 – *Analogie hiérarchie mémoire et décomposition* `ScaLAPACK`.

L'élément est associé aux registres, en effet l'élément est le dernier maillon de la décomposition de `ScaLAPACK`, et transitera par les registres lors des calculs. Par analogie c'est le registre que l'on retrouve au dernier niveau de la hiérarchie mémoire.

Distribution *out-of-core*

Avec la notion de super-bloc, `ScaLAPACK` effectue donc un niveau de décomposition supplémentaire (cf figure 2.4). Dans les algorithmes du prototype *out-of-core*, les super-blocs correspondent à un ensemble de colonnes de taille $N \times MMB$

Le prototype *out-of-core* ajoute trois composantes supplémentaires à `ScaLAPACK`:

- Un nouveau type de matrice distribuée intégrant la notion de super-bloc et de fichier sur disque.
- Des fonctions de gestion de matrices sur disques.
- Trois factorisations *out-of-core* : la factorisation de Cholesky, la factorisation LU et la factorisation QR.

2.3 Contribution au prototype *out-of-core* de `ScaLAPACK`

À partir du prototype *out-of-core*, nous avons proposé deux nouveautés majeures, une optimisation de la factorisation LU *out-of-core* ainsi qu'une nouvelle fonctionnalité, l'inversion matricielle *out-of-core* :

- **La factorisation LU *out-of-core* avec recouvrement des accès disques :** fonction similaire à la version *out-of-core* issue du prototype (`PFDGETRF`) mais offrant un mécanisme permettant de recouvrir les accès disques. L'intérêt de ce développement est justifié dans le chapitre 3. Pour connaître les mécanismes algorithmiques et la mise en œuvre de cette optimisation à la factorisation LU *out-of-core* le lecteur pourra consulter le chapitre 5.
- **L'inversion matricielle *out-of-core* :** les détails de la réalisation de cette fonction feront l'objet du chapitre 4. Notons qu'il existe également une version intégrant les recouvrements des accès disques dont les mécanismes seront développés dans le chapitre 5.

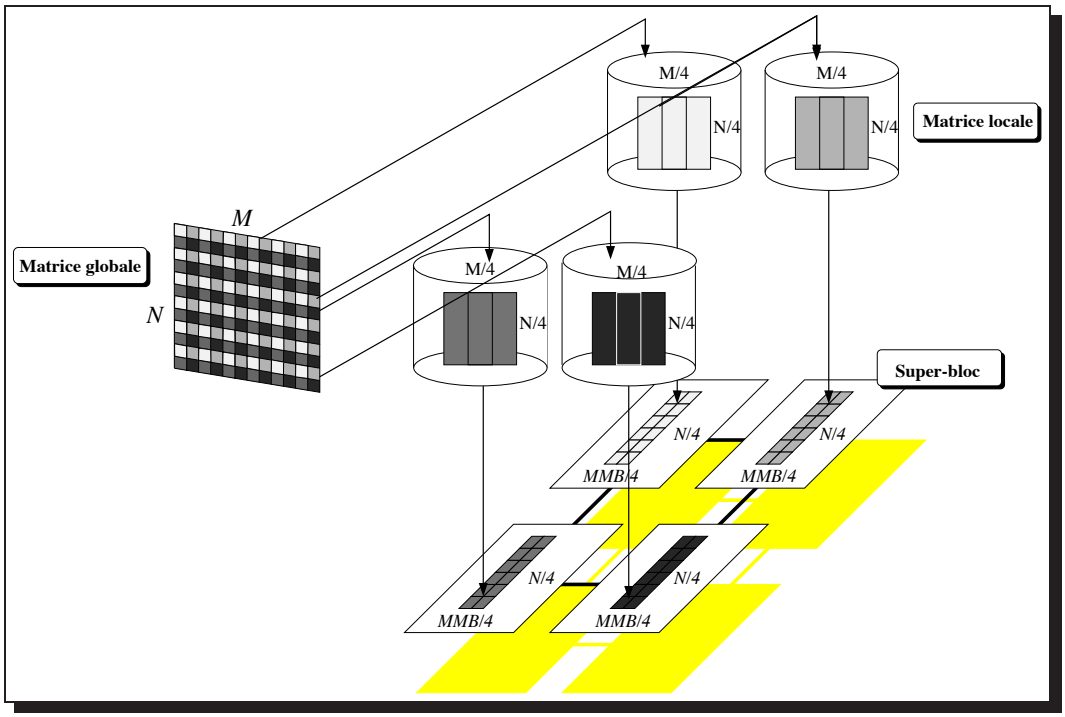

FIG. 2.4 – *Décomposition d'une matrice out-of-core sur 4 nœuds.*

Nous avons également réalisé quelques fonctions annexes afin d'atteindre nos objectifs (cf. annexe A.4).

2.4 Conclusion

Ce chapitre a posé les bases de notre cadre de travail algorithmique, en présentant la librairie d'algèbre linéaire `ScaLAPACK`. Il met ensuite l'accent sur le prototype *out-of-core* qui a servi de fondement aux résultats présentés dans les chapitres qui suivent. Notre contribution à ce prototype y est également décrit. Les différentes unités de `ScaLAPACK` la présentent souvent comme une librairie ayant une mécanique complexe. Cependant, cela n'influence en rien sur son efficacité. En effet avec des unités optimisées les performances obtenues sont correctes. D'un point de vue de la portabilité, `ScaLAPACK` tient ses promesses. Les codes réalisés ont été évalués avec succès sur différentes plates-formes sans apporter de modifications aux sources. Dans le cadre du calcul *out-of-core*, nous montrerons que les développements effectués peuvent être améliorés. De plus, pour bénéficier d'une manipulation plus vaste des matrices *out-of-core*, de nombreuses fonctions pour la manipulation de ces matrices restent à réaliser (des opérations de bases entre deux matrices *out-of-core* comme l'addition, la multiplication, ou des opérations plus complexes comme le calcul des valeurs propres, des outils de redistribution, etc.).

Chapitre 3

La factorisation LU *out-of-core*

Sommaire

3.1 La factorisation LU 41
3.2 Parallélisation de la factorisation LU 43
3.3 Factorisation LU parallèle *out-of-core* 44
3.4 Prédiction des performances 45
3.4.1 Modèle architectural 45
3.4.2 Modélisation 45
3.4.3 Validation expérimentale 49
3.5 Analyse des surcoûts *out-of-core* 49
3.5.1 Réduction du surcoût des communications 49
3.5.2 Réduction du surcoût des Entrées/Sorties 52
3.6 Conclusion 54

La factorisation LU est le cœur de nombreuses applications. Ainsi, l'importance d'optimiser cette routine n'est plus à démontrer du fait de l'accroissement des demandes de la part des applications pour le traitement des grandes données.

Dans ce chapitre, nous présentons un modèle analytique permettant d'évaluer les performances de la factorisation LU parallèle *out-of-core left-looking*. L'objectif de cette modélisation est de déceler des optimisations pour l'algorithme en question. À l'aide des expérimentations effectuées via l'algorithme de la librairie `ScaLAPACK`, nous montrerons la validité de ce modèle. Nous verrons qu'avec une distribution de la matrice correctement effectuée et en instaurant un recouvrement du surcoût des E/S par le calcul, il est possible d'obtenir des performances proches de l'algorithme en mémoire.

Dans la section 3.1 et 3.2, nous décrivons la factorisation LU et la version parallèle de `ScaLAPACK`. Dans la section 3.3 nous présentons la factorisation LU *out-of-core*. La modélisation est développée dans la section 3.4. Enfin dans la section 3.5 nous analysons le surcoût de l'algorithme et nous montrons comment le minimiser.

3.1 La factorisation LU

La factorisation LU de la matrice A correspond à la décomposition de la matrice en deux matrices $L = (l_{ij})_{1 \leq i,j \leq N}$ et $U = (u_{ij})_{1 \leq i,j \leq N}$,

$$A = LU$$

où L est une matrice triangulaire inférieure (c'est-à-dire que $l_{ij} = 0$ pour $1 \leq j < i \leq N$) et U est une matrice triangulaire supérieure (c'est-à-dire $u_{ij} = 0$ pour $1 \leq i < j \leq N$).

Les $N^2 + N$ valeurs de L et U sont le résultat de la résolution de N^2 équations :

$$1 \leq j \leq N \begin{cases} l_{jj} = 1 \\ u_{ij} = a_{ij} - \sum_{k=1}^{k=i-1} l_{ik} u_{kj} \text{ pour } 1 \leq i \leq j \\ l_{ij} = \frac{1}{u_{jj}} (a_{ij} - \sum_{k=1}^{k=j-1} l_{ik} u_{kj}) \text{ pour } j < i \leq N \end{cases} \tag{3.1}$$

À partir du graphe de dépendance de données, nous pouvons montrer que les colonnes de la matrice sont évaluées de la gauche vers la droite, et les lignes du haut vers le bas.

Une méthode utilisée pour la parallélisation de la factorisation LU est basée sur l'algorithme par blocs *right-looking*. Cet algorithme est basé sur une décomposition par blocs des matrices A, L et U :

$$\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} = \begin{pmatrix} L_{00} & 0 \\ L_{10} & L_{11} \end{pmatrix} \begin{pmatrix} U_{00} & U_{01} \\ 0 & U_{11} \end{pmatrix}$$

La décomposition par blocs donne les équations suivantes :

$$A_{00} = L_{00} U_{00} \tag{3.2}$$
$$A_{01} = L_{00} U_{01} \tag{3.3}$$
$$A_{10} = L_{10} U_{00} \tag{3.4}$$
$$A_{11} = L_{10} U_{01} + L_{11} U_{11} \tag{3.5}$$

À partir de ces équations on peut dériver l'algorithme récursif suivant :

1. Effectuer la factorisation LU selon l'équation $A_{00} = L_{00} U_{00}$ (3.2) (cette factorisation peut être réalisée par une autre méthode).
2. Calculer L_{01} (respectivement U_{10}) à partir de l'équation (3.3) (respectivement (3.4)). La méthode de calcul utilisée est la résolution triangulaire (L_{00} et U_{00} sont des matrices triangulaires).
3. Calculer L_{11} et U_{11} à partir de l'équation (3.5):
   (a) Calcul d'une nouvelle matrice $A' = A_{11} - L_{10} U_{01}$.
   (b) Factorisation LU récursive appliquée sur cette nouvelle matrice : $A' = L_{11} U_{11}$.

Cet algorithme est appelé *right-looking* du fait du calcul de la matrice A'. La partie gauche (L_{00} et L_{01}) de la matrice n'est pas utilisée dans le calcul récursif. Cela est également vrai pour la partie inférieure (U_{00} et U_{10}). De plus, à partir des équations (en 3.1), il est facile de montrer que le calcul peut être effectué dans la même zone mémoire. En effet, quelque soit la valeur de i et de j, les points u_{ij} et l_{ij} ne requièrent qu'une valeur de A (a_{ij}). Cette valeur est utilisée une seule fois lors du calcul. Une seule matrice peut donc être utilisée pour A et le résultat L et U.

La décomposition LU est calculée en utilisant un pivotement partiel pour améliorer la stabilité numérique. Les informations sur ce pivotement sont contenues dans la matrice de permutation P, ce qui nous donne en réalité

$$PA = LU$$

Dans l'algorithme *right-looking* avec pivotement partiel, la factorisation de A_{00} et le calcul de L_{01} sont réalisés dans la première étape. Pour simplifier la présentation, nous présentons un algorithme avec pivotement partiel où l'échange des lignes est effectué en deux passages.

La figure 3.1 montre les différentes étapes pour le second appel récursif de la factorisation *right-looking* :

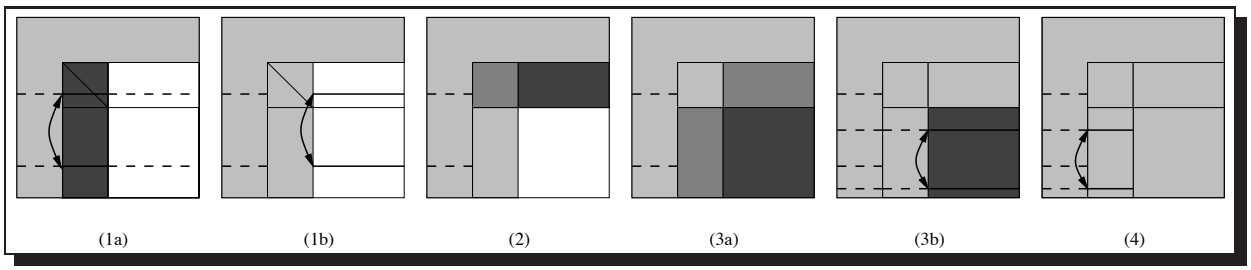

FIG. 3.1 – *Un appel récursif à l'algorithme right-looking. Les lignes horizontales représentent les lignes pivotées. Les lignes hachurées représentent les lignes qui n'ont pas encore été pivotées.*

1a. Calcule la factorisation $P \begin{pmatrix} A_{00} \\ A_{10} \end{pmatrix} = \begin{pmatrix} L_{00} \\ L_{10} \end{pmatrix} U_{00}$ où P est une matrice permutation qui représente le pivotement partiel : la partie gauche de la matrice A (c'est-à-dire $\begin{pmatrix} A_{00} \\ A_{10} \end{pmatrix}$) est factorisée.

1b. Applique le pivot P à la partie droite de la matrice A (c'est-à-dire $\begin{pmatrix} A_{01} \\ A_{11} \end{pmatrix}$)

2. Calcule U_{01} à partir de l'équation (3.3).

3a. Calcule la nouvelle matrice $A' = A_{11} - L_{10}U_{01}$.

3b. Calcule L_{11}, U_{11} et P' par un appel récursif à la factorisation $P'A' = L_{11}U_{11}$ (P' est la matrice de permutation).

4. Applique le pivot P' à la partie inférieure gauche de A (c'est-à-dire L_{10} calculé lors de la première étape. Finalement l'algorithme retourne la composition de P et P'.

3.2 Parallélisation de la factorisation LU

La décomposition par blocs de l'algorithme correspond à la distribution par blocs de la matrice par `ScaLAPACK` (cf section 2.1.4). L'étape 1a. de l'algorithme est calculée par une colonne de p processeurs. L'étape 2. est calculée par une ligne de q processeurs. L'étape 3a. est calculée par l'ensemble de la grille. Le pivotement de l'étape 1b. (respectivement 4.) est exécuté en même temps que le calcul de l'étape 1a. (respectivement 3b.).

À présent, décrivons plus précisément les différentes étapes de l'algorithme. L'étape 1a. est implémentée dans ScaLAPACK par la fonction `pdgetf2` qui factorise le bloc de colonne. Pour chaque élément de la diagonale du bloc supérieur (A_{00}) l'algorithme réalise les opérations suivantes :

1. calcule le pivot par des primitives de communication de type *reduce* et des échanges entre la ligne du pivot et la ligne courante.
2. *broadcast* la ligne de pivot sur la colonne de processeurs.

3. divise la colonne sous le pivot par la valeur du pivot et met à jour les éléments de la matrice à droite de la colonne.

L'étape 2. de l'algorithme est implémentée par la fonction de `ScaLAPACK` `pdtrsm` : le bloc gauche supérieur (A_{00}) est diffusé à la ligne de processeurs puis l'algorithme effectue une résolution triangulaire en utilisant les `BLAS`.

L'étape 3a. est implémentée par la fonction `pdgemm` de `ScaLAPACK` : le bloc U_{01} est diffusé sur les colonnes de processeurs. Le bloc L_{10} est diffusé sur les lignes de processeurs. Ces deux blocs sont alors multipliés entre eux pour mettre à jour A'.

Les performances de l'algorithme dépendent de la taille des blocs et de la topologie de la grille virtuelle de processeurs. La taille des blocs détermine le degré de parallélisme ainsi que les performances via les routines `BLAS`-3 mises à contribution par `ScaLAPACK`. La topologie de la grille détermine le coût des communications. Dans [22], nous avons montré que les meilleures performances sont obtenues avec une grille de quelques lignes : l'étape 1a. de l'algorithme est à grain fin et ne nécessite que de petites communications (mais avec de nombreuses latences) pour le pivotement et pour le calcul de L_{10}.

3.3 Factorisation LU parallèle *out-of-core*

À présent, considérons la situation où la matrice A est trop grande pour tenir en mémoire. Nous présentons l'algorithme de factorisation LU *left right looking* parallèle *out-of-core* utilisé par la routine `pfdgetrf` du prototype *out-of-core* de `ScaLAPACK` [43]. Des algorithmes de même principe sont décrits dans [81, 72].

L'algorithme parallèle *out-of-core* est une extension de l'algorithme parallèle en mémoire. La matrice est factorisée de gauche à droite super-bloc par super-bloc. Chaque fois qu'un nouveau super-bloc de la matrice (appelé **super-bloc actif**) est chargé en mémoire, l'ensemble des pivotements à réaliser et des mises à jour sont appliqués sur le super-bloc actif. Ces opérations sont basées sur l'historique de l'algorithme *right-looking* générées par les étapes précédentes. Pour effectuer les mises à jour, chaque super-bloc à gauche du super-bloc actif (appelé **super-bloc courant**) est relu. Lorsque ces mises à jour sont terminées le super-bloc actif est écrit sur le disque après avoir été factorisé. Une fois, le dernier super-bloc factorisé, la matrice est entièrement relue afin d'appliquer les pivots générés par les phases récursives (étape 4).

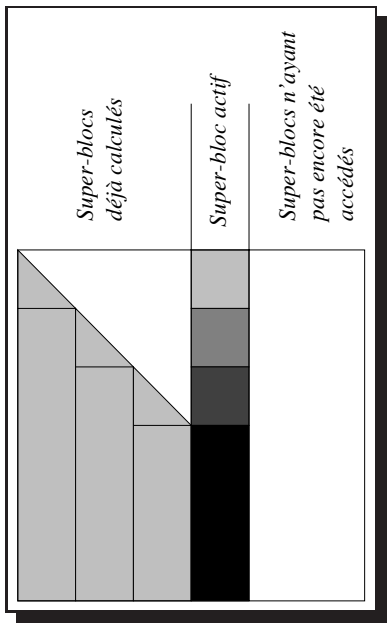

FIG. 3.2 – *Algorithme out-of-core left-looking avec super-blocs*

La mise à jour de chaque super-bloc est résumée figure 3.2. Un super-bloc gauche et la mise à jour consistent à appliquer le pivot des lignes sur le super-bloc actif, puis :

1'. lecture des blocs en dessous de la diagonale du super-bloc courant.

2'. Calcul de la partie U_{01} du super-bloc actif par une résolution triangulaire (fonction `pdtrsm`);

3'. Mise à jour de A_{11} par le produit de U_{01} du super-bloc actif par L_{10} du super-bloc courant (fonctions `pdgemm`).

3.4 Prédiction des performances

Dans cette section nous présentons un modèle architectural et une prédiction des temps d'exécution de l'algorithme parallèle *out-of-core left-right looking*.

3.4.1 Modèle architectural

L'objectif de la modélisation à travers un modèle architectural est de permettre la prédiction des temps d'exécution. Notre modèle architectural est basé sur une grappe. Chaque nœud stocke ses blocs sur son disque. Nous caractérisons cette architecture en définissant quelques constantes fixant le temps de calcul, le temps de communication et le temps d'accès disque.

Temps de calcul. Il est classiquement fixé sur le temps de calcul d'une opération flottante sur un processeur, il est représenté par la constante α. En fait, ce temps n'est pas réellement constant et dépend de la hiérarchie mémoire du processeur et du type de calcul. Par exemple, une multiplication de matrice bénéficie d'une bonne réutilisation du cache mémoire alors que le produit d'un vecteur par un scalaire entraîne une faible localité temporelle. C'est pour cette raison que nous distinguons trois temps de calcul pour les opérations flottantes présentes dans notre traitement : α_g pour la multiplication matricielle, α_t pour la résolution triangulaire et α_s pour la multiplication vecteur/scalaire.

Temps de communication. Le temps de communication est généralement représenté par un modèle du type $\beta + V\tau$, où β est le temps de latence et τ est le temps de transmission d'une unité de données et V est le volume de données communiqué. Dans notre modèle, seule la diffusion est effectuée au niveau des communications. Les constantes β et τ sont dépendantes de la topologie de la grille virtuelle : β_p^q est le temps de latence pour une colonne de p processeurs pour diffuser les données[1] sur leurs lignes. Dans ce cas $1/\tau_p^q$ représente le débit. De la même façon, β_q^p et τ_q^p représentent le temps pour une ligne de q processeurs pour diffuser les données sur leurs colonnes. Cette fonction dépend directement de l'implémentation des communications sur le réseau. Par exemple, pour une grappe de stations de travail avec un switch, la diffusion peut-être implémentée selon un arbre de diffusion. Dans ce cas, $\beta_p^q = \log_2 q \times \beta$ et $\tau_p^q = \log_2 q \times \frac{\tau}{p}$ où β est la latence de communication pour un nœud et $1/\tau$ le débit moyen. Avec un hub (c'est-à-dire une technologie type bus), le modèle nous donne : $beta_p^q = p(q-1) \times \beta$ et $\tau_p^q = \tau$ si $q > 1$, $\tau_p^q = 0$ si $q = 1$.

Temps d'accès disque. Le temps d'accès disque est basé sur le débit du disque. τ^{io} correspond au temps pour lire ou écrire un mot sur un disque. Dans ce cas, $\tau_p^{io} = \frac{\tau^{io}}{p}$ est le temps pour écrire ou lire p mots en parallèle pour p disques.

3.4.2 Modélisation

Pour modéliser l'algorithme, nous estimons le temps utilisé par chaque fonction. Pour chacune de ces fonctions, nous distinguons le temps de calcul et le temps de communication. Nous distinguons

1. Les données sont distribuées de manière uniforme sur les processeurs.

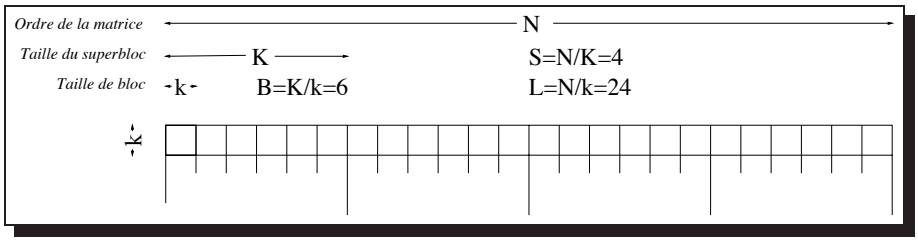

FIG. 3.3 – *Exemple de distribution des données pour la décomposition de la factorisation LU out-of-core.*

également le temps de la résolution en mémoire et le surcoût de temps introduit par la gestion *out-of-core*.

Soit N l'ordre de la matrice, K la largeur du super-bloc, la taille des blocs est de $k \times k$. La grille des processeurs est composée de p lignes de q colonnes. Nous avons les contraintes suivantes pour nos paramètres : N est un multiple de K et K est un multiple de k et q. Soit $L = \frac{N}{k}$ la taille en blocs de la matrice, $S = \frac{N}{K}$ le nombre de super-blocs, et $B = \frac{K}{k}$ la taille en blocs du super-bloc (cf figure 3.4.2).

Par simplification, nous ne considérons pas le coût de la procédure de pivot dans notre analyse. Cet dernier est par ailleurs insignifiant par rapport au reste de l'exécution. Ce coût correspond à une division pour chaque élément de la matrice et au coût d'échange des lignes auquel s'ajoute une lecture et une écriture de la matrice. La figure 3.4 montre l'utilisation des blocs dans les trois fonctions principales pour le super-bloc en cours et le super-bloc actif.

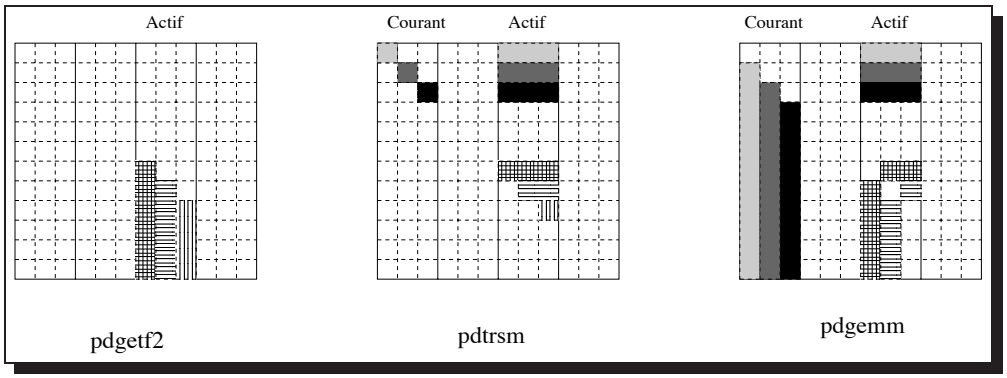

FIG. 3.4 – *Blocs impliqués dans les 3 fonctions principales de la factorisation LU. On distingue les blocs du super-bloc actif et du super-bloc courant (la matrice contient 4 super-blocs).*

Coût de la factorisation LU (`pdgetf2`)

L'étape 1 de l'algorithme (fonction `pdgetf2` de `ScaLAPACK`) est appliquée sur un bloc de colonnes (de taille k) sous la diagonale. Il y a L blocs de taille k. Ce calcul est indépendant de la taille du super-bloc.

Pour le coût de calcul, nous distinguons le calcul du bloc de la diagonale et le calcul des blocs sous la diagonale. Le coût de calcul pour le bloc diagonal correspond au coût d'une factorisation

LU sur une matrice de taille $k \times k$:

$$\alpha_s \sum_{j=1}^{j=k} \left(\sum_{i=1}^{i=j-1} (2i-2) + \sum_{i=j+1}^{j=k} (2j-1) \right) = \alpha_s (\frac{2k^3}{3} - \frac{k^2}{2} - \frac{k}{6})$$

En posant H comme étant la hauteur on a pour le bloc sous la diagonale :

$$\alpha \left(k \times H + H \sum_{j=2}^{j=k} 2(j-1) \right) = \alpha \times k^2 \times H$$

Ce qui nous donne un temps total de calcul pour la fonction `pdgetf2` sur la matrice complète de taille $N \times N$:

$$\begin{aligned} & \frac{\alpha_s}{p} \sum_{j=1}^{j=L} \left((\frac{2k^3}{3} - \frac{k^2}{2} - \frac{k}{6}) + (k^2 \times (L-i) \times k) \right) \\ = \; & \frac{\alpha_s}{p} \left(\frac{Nk^2}{6} + \frac{N^2 k}{2} - \frac{Nk}{2} - \frac{N}{6} \right) \end{aligned} \tag{3.6}$$

Pour les communications réalisées dans `pdgetf2`, pour chaque bloc de la diagonale et pour chaque élément de la diagonale, la partie droite est diffusée sur la colonne de processeurs :

$$L \times \left(k\beta_1^p + \frac{k(k+1)}{2} \tau_1^p \right) = N \times (\beta_1^p + \frac{(k+1)}{2} \tau_1^p) \tag{3.7}$$

Coût de la résolution triangulaire (`pdtrsm`)

L'étape 2 de l'algorithme (calcul de U_{01}) est appliquée sur chaque bloc de ligne à droite de la diagonale. Cette étape consiste à réaliser une résolution triangulaire pour chaque bloc à droite du bloc diagonal. Le coût de calcul de la résolution triangulaire entre deux blocs de taille $k \times k$ est de $\alpha_t k^3$.

Le coût de calcul pour tous les `pdtrsm` effectués par l'algorithme nous donne :

$$\frac{1}{q} \sum_{i=1}^{i=L-1} \alpha_t k^3 \times i = \frac{\alpha_t}{2q} \times (N^2 k - N k^2) \tag{3.8}$$

Le coût de communication pour le `pdtrsm` correspond à la diffusion du bloc diagonal sur la ligne de processeur. Une diffusion est effectuée durant le calcul du super-bloc actif, et une autre est effectuée pour les mises à jour. Dans ce cas le coût total des diffusions est de :

$$\begin{aligned} & (S(B-1) + \sum_{i=1}^{S-1} (i \times B)) \times (\beta_p^q + k^2 \tau_p^q) \\ = \; & S(B-1)(\beta_p^q + k^2 \tau_p^q) + & (3.9) \\ & B \frac{S(S-1)}{2} (\beta_p^q + k^2 \tau_p^q) & (3.10) \end{aligned}$$

Coût de la multiplication matricielle (`pdgemm`)

L'étape 3 de l'algorithme met à jour la sous-matrice A'. La fonction réalise une multiplication matricielle avec des diffusions. Pour la sous-matrice d'ordre H, il y a $(\frac{H}{k})^2$ multiplications par bloc. Le coût de chaque multiplication est $2\alpha_g k^3$. Le coût de calcul pour les appels de `pdgemm` est :

$$\frac{1}{pq}\sum_{i=1}^{i=L-1} i^2 \times 2\alpha_g k^3 \quad = \quad \frac{\alpha_g}{pq}\left(\frac{N^3+Nk^2}{3} - N^2k\right) \tag{3.11}$$

Pour les coûts de communication, nous distinguons le coût de la factorisation du super-bloc actif et le coût de la mise à jour par les super-blocs courants.

Pour la factorisation du super-bloc actif, le coût correspond à la diffusion du bloc ligne de la diagonale et la diffusion des blocs colonnes sous la diagonale.

$$\begin{aligned} & S((B-1)\beta_q^p + \frac{B(B-1)}{2}k^2\tau_q^p) \\ + & \sum_{i=1}^{i=S}((B-1)\beta_p^q + \frac{B(B-1)}{2}k^2\tau_p^q + (i-1)B(B-1)k^2\tau_p^q) \end{aligned} \tag{3.12}$$

La figure 3.2 illustre les mises à jour successives pour un super-bloc actif. Chaque bloc de colonnes sous la diagonale à gauche du super-bloc actif sont lus et diffusés.

$$\begin{aligned} & \sum_{i=1}^{i=S-1} i \times (B\beta_p^q + \frac{B(B-1)}{2}k^2\tau_p^q + iB^2k^2\tau_p^q) \\ = & \frac{N(N-K)(6\beta_p^q + (Kk+4Nk-3k^2)\tau_p^q)}{12Kk} \end{aligned} \tag{3.13}$$

De la même façon, les blocs de lignes du super-bloc courant sont diffusés.

$$\frac{S(S-1)}{2}(B\beta_q^p + K^2\tau_q^p) \tag{3.14}$$

Coût des accès disques

Les Entrées/Sorties correspondent à l'écriture et à la lecture du super-bloc actif :

$$2SNK\tau_{pq}^{io}$$

et à la lecture des super-blocs à gauche :

$$\begin{aligned} & \sum_{i=1}^{i=S-1} i \times (\frac{B(B-1)}{2}k^2\tau_{pq}^{io} + (i-1)B^2k^2\tau_{pq}^{io}) \\ = & \frac{N(N-K)(Kk+4Nk+3k^2)\tau_{pq}^{io}}{12Kk} \end{aligned} \tag{3.15}$$

3.4.3 Validation expérimentale

Pour valider notre modèle de prédiction, nous exécutons la factorisation LU *out-of-core* de `ScaLAPACK` sur une grappe de 8 PC-Celeron sous Linux interconnectés par un switch Fast-Ethernet. Chaque nœud dispose de 96 Mo de mémoire physique. Le modèle décrit dans la section précédente est instancié avec les constantes suivantes mesurées expérimentalement :

$$1/\alpha_g = 311\ Mflops, 1/\alpha_t = 160\ Mflops, 1/\alpha_s = 16\ Mflops,$$

$$\beta_p^q = \beta_q^p = 1.7\ ms, 1/\tau_p^q = 1/\tau_q^p = 1.1\ Mo/s, 1/\tau_{io} = 1.8\ Mo/s$$

Les tableaux 3.5 et 3.6 montrent la comparaison entre le temps d'exécution et le temps estimé par la modélisation (en *italique*). Nous mesurons le temps des Entrées/Sorties, le temps de calcul et les temps de communication durant la factorisation du super-bloc actif et durant la phase de mise à jour. Pour le calcul et les communications, la modélisation est proche de la réalité. Pour les temps des accès disques, on constate quelques différences. Cela est principalement dû à notre modélisation des Entrées/Sorties. En effet, les performances disques sont plus difficiles à modéliser. Les accès disques ont été estimés linéaires, hors, dans la réalité ce n'est pas le cas. La fragmentation du fichier sur le disque est un des facteurs qui influe sur la non linéarité des performances disques.

3.5 Analyse des surcoûts *out-of-core*

En comparaison avec la version de l'algorithme en mémoire, le surcoût de la version *out-of-core* correspond au coût des accès disques et au coût de diffusion (des colonnes) lors de la mise à jour du super-bloc actif. Pour chaque super-bloc actif, les super-blocs de gauche doivent être lus et diffusés à nouveau.

Ce surcoût est défini par les équations (3.10) et (3.13) pour les communications et (3.15) pour les E/S. Il est aisé de montrer que si $K = N$ (c'est-à-dire $S = 1$) alors ce coût est égal à zéro : on retrouve l'exécution *in-core* de l'algorithme.

Le surcoût non négligeable est de l'ordre de $O(N^3)$. Dans la suite de ce chapitre, nous verrons comment réduire ce surcoût. Soit $O_C = (3.10) + (3.13)$ le surcoût des communications et $O_{IO} = (3.15)$ le surcoût des E/S.

3.5.1 Réduction du surcoût des communications

Comme le montre la modélisation et les résultats expérimentaux, la topologie de la grille de processeurs a une grande influence sur les surcoûts des communications :

Lemme 1 *Si le nombre de colonnes q est égal à 1, alors* $O_C = 0$ *!*

S'il n'y a qu'une seule colonne de processeurs, il n'y a pas de diffusion des colonnes lors de la mise à jour. Pour ce qui est des communications, les coûts de diffusion augmentent avec le nombre de processeurs. Plus le nombre de colonnes est important, plus O_C sera important.

La figure 3.7 montre l'influence de la topologie sur les performances. Dans la même figure, on peut voir les performances de la version *right looking* en mémoire. Les constantes sont celles d'une grappe de PC-Celeron. Nous montrons ainsi que la meilleure topologie pour les performances in-core est la moins efficace pour la version *out-of-core*. La figure 3.8 montre le rapport entre les accès disques, les communications et le temps total d'exécution.

M	K	$p \times q$	E/S	Calcul	Comm. Active	Comm. mise à jour	Temps d'exécution
12288	3072	1x8	5m 23s/*4m 19s*	12m 24s/*15m 54s*	32m 33s/*29m 54s*	51m 19s/*1h 04m*	1h 41m/*1h 54m*
1,2 Gb / S=4							204 Mflp/*179 Mflp*
		2x4	4m 51s/*4m 19s*	11m 15s/*13m 27s*	10m 47s/*11m 00s*	18m 31s/*23m 23s*	45m 26s/*52m 85s*
							453 Mflp/*395 Mflp*
		4x2	5m 13s/*4m 19s*	11m 32s/*12m 21s*	6m 07s/*5m 46s*	10m 25s/*12m 57s*	33m 27s/*35m 22s*
							616 Mflp/*582 Mflp*
		8x1	5m 37s/*4m 19s*	12m 39s/*12m 11s*	11m 23s/*8m 36s*	18m 54s/*22m 40s*	48m 33s/*47m 36s*
							424 Mflp/*433 Mflp*
20480	2048	1x8	34m 11s/*19m 53s*	1h 08m/*1h 04m*	1h 30m/*1h 22m*	8h 29m/*8h 37m*	11h 41m /*11h 23m*
3,3 Gb / S=10							136 Mflp/*139 Mflp*
		2x4	34m 54s/*19m 53s*	1h 00m/*57m 31s*	28m 00s/*28m 38s*	2h 55m/*2h 59m*	4h 57m/*4h 44m*
							321 Mflp/*335 Mflp*
		4x2	35m 34s/*19m 53s*	1h 00m/*54m 26s*	9m 57s/*10m 53s*	1h 07m/*1h 08m*	2h 51m/*2h 33m*
							558 Mflp/*612 Mflp*
		8x1	35m 39s/*19m 53s*	1h 03m/*53m 32s*	10m 16s/*10m 09s*	1h 14m/*1h 15m*	3h 03m/*2h 39m*
							521 Mflp/*600 Mflp*
27648	1024	1x8	0 h 52m /*1h 16m*	2h 28m/*2h 29m*	2h 40m/*2h 24m*	32h/*32h*	26h /*26h*
6,1 Gb / S=27							77 Mflp/*77 Mflp*
		2x4	0h 40m/*1h 16m*	2h 27m/*2h 16m*	48m 25s/*49m 24s*	14h 57m/*15h 06m*	18h 53m/*19h 28m*
							205 Mflp/*200 Mflp*
		4x2	0h 50m/*1h 16s*	2h 03m/*2h 11m*	13m 47s/*15m 37s*	4h 29m/*4h 33m*	7h 35m/*8h 16m*
							506 Mflp/*473 Mflp*
		8x1	0h 51m/*1h 16s*	2h 00m/*2h 09m*	6m 21s/*8m 04s*	2h 27m/*2h 28m*	5h 24m/*6h 01m*
							714 Mflp/*649 Mflp*

FIG. 3.5 – *Comparaison entre l'exécution expérimentale et le modèle théorique (en italique) du temps d'exécution et des performances de l'algorithme de factorisation LU out-of-core. M est l'ordre de la matrice, K la taille du super-bloc, p le nombre de processeurs sur une ligne de la grille, q le nombre de colonnes, S est le nombre de super-blocs. La taille de la matrice est en Gigaoctets et se trouve dans la première colonne. Les temps sont donnés en heures (h), minutes (m) et secondes (s). La dernière colonne présente en Mflops (Mflp), les performances de l'exécution expérimentale, et, les prévisions du modèle théorique.*

M	K	$p \times q$	E/S	Calcul	Comm. Active	Comm. mise à jour	Temps d'exécution
12288	3072	1x8	5m 06s/*4m 19s*	16m 42s/*10m 37s*	3m 9s/*3m 00s*	4m 39s/*6m 28s*	29m 44s/*24m 25s*
1,2 Gb / S=4							693 Mflp/*844 Mflp*
		2x4	4m 20s/*4m 19s*	13m 43s/*10m 32s*	59s/*1m 16s*	1m 41s/*2m 21s*	20m 58s/*18m 30s*
							983 Mflp/*1114 Mflp*
		4x2	4m 44s/*4m 19s*	12m 23s/*10m 36s*	34s/*55s*	58s/*1m 18s*	18m 56s/*17m 10s*
							1088 Mflp/*1200 Mflp*
		8x1	4m 46s/*4m 19s*	12m 08s/*10m 53s*	1m 06s/*1m 22s*	1m 42s/*2m 16s*	20m 13s/*18m 51s*
							1019 Mflp/*1093 Mflp*
20480	2048	1x8	31m 01s/*19m 52s*	1h 02m/*48m 36s*	10m 18s/*8m 12s*	52m 9s/*51m 51s*	2h 36m/*2h 8m*
3,3 Gb / S=10							610 Mflp/*742 Mflp*
		2x4	33m 32s/*19m 52s*	1h 19m/*48m 22s*	2m 51s/*3m 10s*	18m 31s/*17m 59s*	2h 15m/*1h 29m*
							706 Mflp/*1067 Mflp*
		4x2	33m 10s/*19m 52s*	1h 04m/*48m 34s*	1m 07s/*1m 40s*	8m 14s/*6m 55s*	1h 49m/*1h 17m*
							1208 Mflp/*1238 Mflp*
		8x1	30m 01s/*19m 52s*	1h 03m/*49m 19s*	1m 10s/*1m 52s*	7m 00s/*7m 37s*	1h 44m/*1h 18m*
							917 Mflp/*1212 Mflp*
27648	1024	1x8	53m 03s/*1h 16m*	2h 45m/*1h 59m*	14m 42s/*14m 26s*	4h 3m/*4h 28m*	7h 57m/*7h 58m*
6,1 Gb / S=27							492 Mflp/*490 Mflp*
		2x4	49m 48s/*1h 16m*	2h 17m/*1h 58m*	4m 25s/*5m 21s*	1h 21m/*1h 30m*	4h 34m/*4h 51m*
							857 Mflp/*805 Mflp*
		4x2	50m 38s/*1h 16m*	2h 01m/*1h 58m*	1m 18s/*2m 20s*	25m 23s/*25m 36s*	3h 20m/*3h 45m*
							1174 Mflp/*1040 Mflp*
		8x1	51m 16s/*1h 16m*	1h 53m/*2h 00m*	52s/*1m 58s*	13m 30s/*15m 00s*	3h 02m/*3h 34m*
							1290 Mflp/*1097 Mflp*

FIG. 3.6 – *Comparaison entre l'exécution expérimentale et le modèle théorique (en italique) du temps d'exécution et des performances de l'algorithme de factorisation LU out-of-core avec un réseau Fast-Ethernet. M est l'ordre de la matrice, K la taille du super-bloc, p le nombre de processeurs sur une ligne de la grille, q le nombre de colonnes, S est le nombre de super-blocs. La taille de la matrice est en Gigaoctets et se trouve dans la première colonne. Les temps sont donnés en heures (h), minutes (m) et secondes (s). La dernière colonne présente en Mflops (Mflp), les performances de l'exécution expérimentale, et, les prévisions du modèle théorique.*

FIG. 3.7 – *Performances théoriques de la factorisation LU sur une grappe de 16 PC-Celeron/Linux (64Mo) interconnectés par un switch Fast-Ethernet : comparaison entre la version parallèle in-core (IC) right-looking et la version parallèle out-of-core (OoC) left-right looking , avec trois topologies différentes.*

3.5.2 Réduction du surcoût des Entrées/Sorties

Une méthode courante pour réduire les surcoûts des E/S consiste à effectuer un recouvrement des accès disques par le calcul. Dans l'algorithme *left-right looking*, durant la mise à jour du super-bloc actif, les super-blocs à gauche sont lus de la gauche vers la droite. Un schéma de recouvrement serait de lire le prochain super-bloc durant la phase de mise à jour du super-bloc actif. Ce recouvrement n'est complet que si le temps de mise à jour du super-bloc est plus important que le temps de lecture du prochain super-bloc.

À présent, considérons les ressources nécessaires pour effectuer le recouvrement total. Soit M la quantité de mémoire allouée pour un super-bloc sur un processeur. Pour une matrice d'ordre N la taille du super-bloc est alors de $K = \frac{pqM}{N}$. Soit O^o_{IO} le surcoût des IO non recouverts, dans ce cas :

Théorème 1 *Si le nombre de colonnes du processeur q est égal à 1 et si*

$$pM \geq N\frac{t_{io}}{2\alpha}$$

alors

$$O^o_{IO} = 0$$

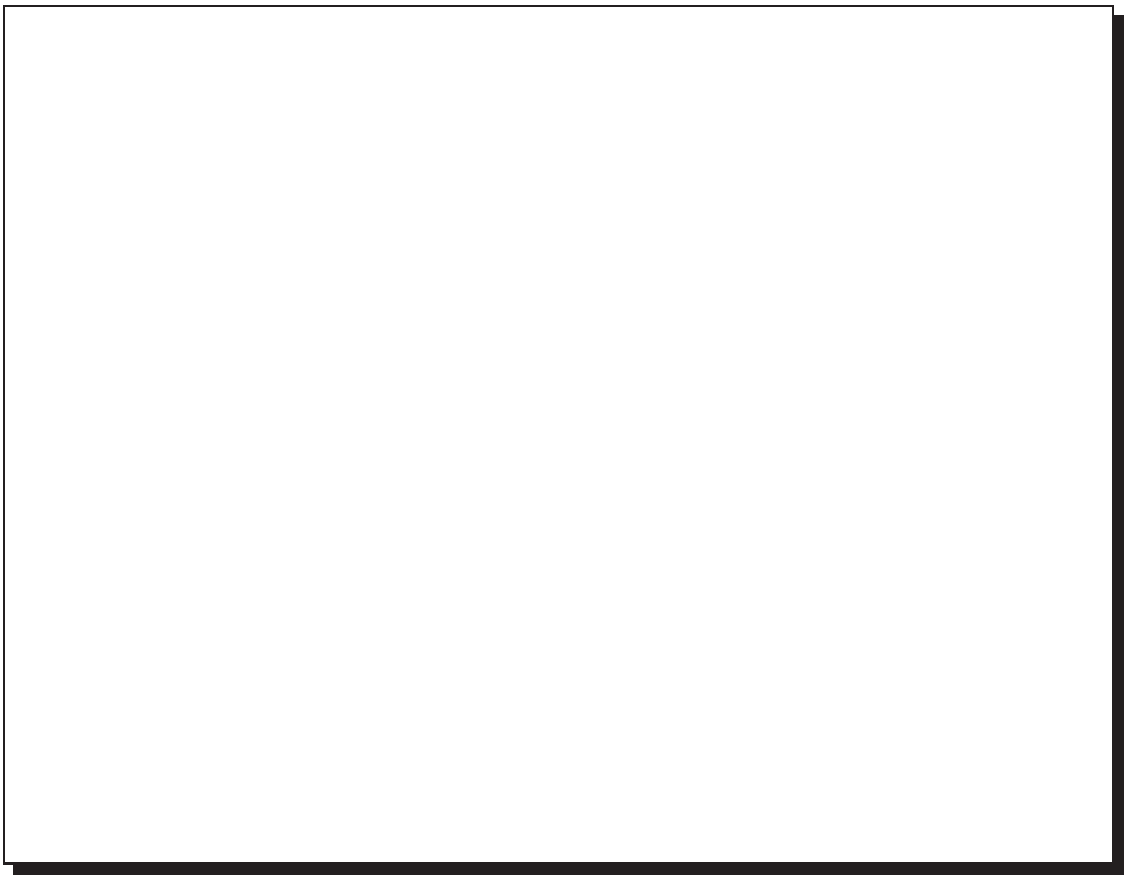

FIG. 3.8 – *Rapports des surcoûts de communication et des E/S.*

Preuve 1 *Revenons à la partie de mise à jour de l'algorithme (figure 3.2). Par simplification, nous sous-estimons le temps de calcul, et ne considérons que le coût le plus important : celui du* `pdgemm`*. Pour chaque super-bloc courant à gauche du super-bloc actif, le temps de mise à jour est égal au temps de communication de B blocs de lignes dans le super-bloc actif ($q = 1$ implique qu'il n'y ait pas de communication de bloc de colonnes pour le super bloc courant) plus le temps de calcul de la mise à jour du super-bloc actif. La partie des communications est égale à :*

$$B^2k^2\tau_q^p + B\beta_q^p \tag{3.16}$$

Soit H la hauteur du super-bloc courant (en nombre de $k \times k$ blocs). Le temps de calcul pour la mise à jour du super-bloc actif avec le super-bloc courant est de :

$$((H-B) \times B + \frac{B(B-1)}{2}) \times (2\frac{\alpha_t}{pq}k^3B) \tag{3.17}$$

Le temps de lecture du prochain super-bloc courant est :

$$((H-2B) \times B + \frac{B(B-1)}{2}) \times (k^2\tau_{pq}^{io})) \tag{3.18}$$

À présent, considérons la situation où les E/S sont recouvertes par le calcul (c'est à dire $O^o_{IO} = 0$), alors $\frac{(3.17)+(3.16)}{(3.18)} \geq 1$. Notons que si $\frac{(3.17)}{(3.18)} \geq 1$ alors $\frac{(3.17)+(3.16)}{(3.18)} \geq 1$. Ainsi le problème est ramené à déterminer la taille de B :

$$\frac{((H-B)B + \frac{B(B-1)}{2})}{((H-2B)B + \frac{B(B-1)}{2})} \times \frac{(2\frac{\alpha_t}{pq}k^3B)}{k^2\tau^{io}_{pq}} \geq 1$$

La première partie de l'expression est toujours supérieure à 1. Nous déterminons pour quelle taille de super-bloc la seconde partie de l'expression est supérieure à 1. Par définition $K = k * B$ (K est la taille du super-bloc en nombre de colonnes), et $\tau^{io}_p q = \frac{t_{io}}{pq}$. Nous avons

$$\frac{(2\frac{\alpha}{pq}k^3B)}{k^2\tau^{io}_{pq}} \geq 1 \Leftrightarrow 2\frac{\alpha}{t_{io}}K \geq 1 \Leftrightarrow K \geq \frac{t_{io}}{2\alpha}$$

Ainsi $K = pqM/N$ et $q = 1$, si $pM \geq N\frac{t_{io}}{2\alpha}$ alors $\frac{(3.17)}{(3.18)} \geq 1$, c'est-à-dire $O^o_{IO} = 0$.

Notons que dans l'algorithme, la taille du super-bloc actif et la taille du super-bloc courant sont égales. Une idée serait de réduire la mémoire physique pour spécifier des tailles différentes pour le super-bloc courant et le super-bloc actif durant la mise à jour : en augmentant la taille du super-bloc actif (c'est-à-dire le temps de calcul) et en réduisant la taille du super-bloc courant (c'est-à-dire le temps de lecture).

3.6 Conclusion

Dans ce chapitre nous avons présenté un modèle de prédiction des performances de l'algorithme parallèle de factorisation LU *left-right looking out-of-core* utilisé par `ScaLAPACK`. Cet algorithme est une extension de l'algorithme parallèle de factorisation LU *right-looking*. Fort de cette modélisation, nous avons isolé les surcoûts introduits par la version *out-of-core*. Nous avons observé que la meilleure topologie de grille virtuelle pour supprimer le surcoût des communications est une seule colonne de processeurs. Nous montrons que le recouvrement des E/S par le calcul nous permet de réduire le surcoût des accès disques de l'algorithme. Nous déterminons la taille mémoire nécessaire pour permettre ce recouvrement [21]. La taille mémoire nécessaire est proportionnelle à l'ordre de la matrice. Pour mettre en évidence la pertinence de ce résultat, considérons une grappe de PC-Celeron avec 16 nœuds et un réseau Fast-Ethernet. Pour factoriser une matrice de 80Go (une matrice d'ordre 100000) nous avons besoin de 26Mo de mémoire par super-bloc (actif, courant et préchargé) par nœud, c'est-à-dire 78Mo par nœud ! Le modèle nous donne un temps d'exécution pour la factorisation de cette matrice en 4,5 jours sans recouvrement, et 2,5 jours dans l'autre cas. Si nous substituons les processeurs Celeron d'Intel à 237Mflops par des processeurs AXP de Digital/Alpha à 757 Mflops, dans ce cas la mémoire nécessaire par nœud est de 252 Mo. Le temps de calcul estimé est d'environ 36 heures sans recouvrement et environ 21 heures avec (soit 1,7 fois plus rapide). Ce dernier temps est le temps estimé pour l'algorithme avec la même topologie (c'est-à-dire une colonne de processeurs). Avec une meilleure topologie pour la version en mémoire (4 colonnes de 4 processeurs), l'algorithme en mémoire prendra environ 18h pour factoriser la matrice, mais la mémoire nécessaire par nœud devra être de 5Go, soit 20 fois plus que la quantité de mémoire nécessaire pour la version *out-of-core*.

Nous proposons une implémentation du schéma de recouvrement dans la factorisation parallèle LU *left-right* looking *out-of-core* dans le chapitre 5, et nous validerons expérimentalement les

théories exposées dans ce chapitre. Nous pourrons également intégrer le recouvrement des communications en bénéficiant des travaux réalisés dans le cadre *in-core* [37].

Chapitre 4

Inversion matricielle *out-of-core*

Sommaire

4.1 Inversion matricielle par factorisation LU 57
4.2 Parallélisation de l'inversion 59
4.3 Inversion matricielle *out-of-core* 62
4.3.1 Principe général 62
4.3.2 Résolution triangulaire *out-of-core* 62
4.4 Prédiction des performances 63
4.4.1 Modélisation de l'inversion matricielle 63
4.4.2 Validation expérimentale 65
4.5 Analyse des surcoûts *out-of-core* 65
4.5.1 Réduction du surcoût des communications 65
4.5.2 Réduction du surcoût des Entrées/Sorties 65
4.6 Optimisation par variation de la taille des super-blocs 69
4.7 Conclusion 69

Forts des performances obtenues sur la factorisation LU (cf. chapitre 3) nous avons élaboré une inversion matricielle parallèle *out-of-core*. L'inversion matricielle est utilisée dans de nombreuses applications effectuant des résolutions de système linéaire par méthode directe.

Ce chapitre propose un algorithme d'inversion matricielle *out-of-core* par extension des travaux réalisés sur la factorisation LU *out-of-core* `ScaLAPACK` [42]. Puis par extension du modèle du chapitre précédent, nous évaluerons les performances théoriques de cet algorithme. Par ailleurs, ce modèle permettra de mettre en évidence les surcoûts dus au traitement *out-of-core*. Nous montrerons comment réduire ces surcoûts afin d'obtenir une inversion *out-of-core* pouvant atteindre des performances proches d'une résolution en mémoire.

4.1 Inversion matricielle par factorisation LU

Soit A une matrice à deux dimensions. L'inverse de la matrice A, A^{-1} peut être obtenue à partir de sa décomposition LU. La matrice A^{-1} est obtenue par deux résolutions triangulaires (appelées également substitutions triangulaires) appliquées sur le résultat de la décomposition LU. Il existe deux types de résolutions triangulaires :

- **La résolution triangulaire avant** qui résout des équations du type $LX = B$ où L est une matrice triangulaire inférieure. La résolution triangulaire correspond au calcul de la substi-

tution avant appliquée à chaque colonne de X (appelé x) et B (appelé b). Cela revient à résoudre N fois l'équation $Lx = b$ que l'on peut réécrire sous la forme :

$$\begin{array}{lllll} L_{1,1}x_1 & & & = & b_1 \\ L_{1,1}x_1 + L_{2,2}x_2 & & & = & b_2 \\ & & & & \vdots \\ L_{1,1}x_1 + L_{2,2}x_2 + \ldots + L_{1,n-1}x_{n-1} & & & = & b_{n-1} \\ L_{1,1}x_1 + L_{2,2}x_2 + \ldots + L_{1,n-1}x_{n-1} + L_{1,n}x_n & & & = & b_n \end{array}$$

Les valeurs se déterminent donc de x_1 à x_n, c'est pour cette raison que l'on parle de substitution avant (cf figure 4.1).

– **La résolution triangulaire arrière** qui résout des équations du type $UX = B$ où U est une matrice triangulaire supérieure. De la même façon que pour le cas triangulaire avant, cela revient à résoudre N fois l'équation $Ux = b$ que l'on peut réécrire sous la forme :

$$\begin{array}{lllll} U_{1,1}x_1 + U_{1,2}x_2 + \ldots + U_{1,n-1}x_{n-1} + U_{1,n}x_n & = & b_1 \\ U_{2,2}x_2 + \ldots + U_{1,n-1}x_{n-1} + U_{1,n}x_n & = & b_2 \\ & & \vdots \\ U_{1,n-1}x_{n-1} + U_{1,n}x_n & = & b_{n-1} \\ U_{1,n}x_n & = & b_n \end{array}$$

Les valeurs se déterminent donc de x_n à x_1, d'où le nom de substitution arrière (cf figure 4.1).

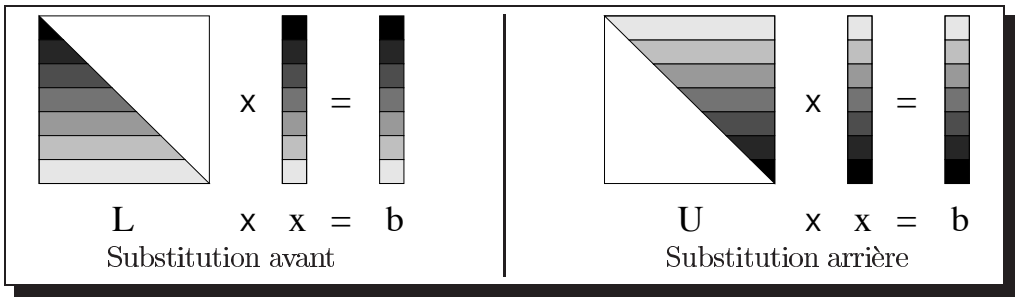

FIG. 4.1 – *Substitutions triangulaires. L'ordre de résolution s'effectue du sombre vers le clair.*

Le calcul de la matrice inverse de A est ramené à ces deux types de résolutions triangulaires selon le mécanisme suivant. Soit I la matrice identité :

$$AA^{-1} = I$$

implique que

$$LUA^{-1} = I$$

Posons

$$Y = UA^{-1}$$

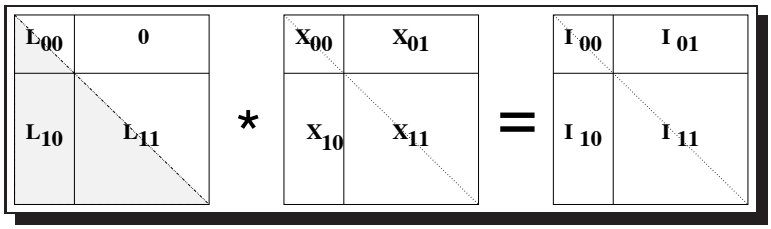

FIG. 4.2 – *Décomposition par blocs pour la résolution triangulaire.*

où Y est déterminé à l'aide d'une résolution triangulaire *arrière* à partir de l'équation

$$LY = I \tag{4.1}$$

La matrice inverse A^{-1} est calculée par une seconde résolution triangulaire en appliquant une résolution triangulaire *avant* sur l'équation

$$UA^{-1} = Y \tag{4.2}$$

Comme nous l'avons vu au chapitre 3 la décomposition LU nous est retournée par `ScaLAPACK` avec un vecteur de permutation P ce qui correspond à l'équation :

$$PA = LU$$

Dans le cas de l'inversion, A^{-1} sera alors déterminé à partir de la matrice de permutation P. Ce qui à partir des équations précédentes nous donnent :

$$\begin{cases} AA^{-1} = I \\ PAA^{-1} = PI \\ PLUA^{-1} = PI \end{cases}$$

En appliquant la fonction de factorisation LU de `ScaLAPACK`, on obtient PLU. Le vecteur pivot P est alors appliqué sur la matrice identité pour déterminer A^{-1}.

La section suivante présente la parallélisation de la substitution triangulaire, une étude analogue concernant la parallélisation de la factorisation LU a été présentée dans la section 3.2.

4.2 Parallélisation de l'inversion

Tout comme pour la factorisation LU, le principe de parallélisation est basé sur une décomposition par blocs (cf. figure 4.2). Nous avons :

$$\begin{pmatrix} L_{00} & 0 \\ L_{10} & L_{11} \end{pmatrix} \begin{pmatrix} X_{00} & X_{01} \\ X_{10} & X_{11} \end{pmatrix} = \begin{pmatrix} I_{00} & I_{01} \\ I_{10} & I_{11} \end{pmatrix}$$

Cette décomposition par blocs nous donne les équations suivantes :

$$
\begin{aligned}
L_{00}X_{00} &= I_{00} && (4.3)\\
L_{00}X_{01} &= I_{01} && (4.4)\\
L_{10}X_{00} + L_{11}X_{10} &= I_{10} && (4.5)\\
L_{10}X_{01} + L_{11}X_{11} &= I_{11} && (4.6)
\end{aligned}
$$

Ces équations déterminent alors l'algorithme récursif suivant (cf figure 4.3) :

1. On évalue X_{00} en effectuant une résolution triangulaire à partir de l'équation $L_{00}X_{00} = I_{00}$ (équation (4.3)) et X_{01} par résolution triangulaire sur l'équation $L_{00}X_{01} = I_{01}$ (équation (4.4)).
2. X_{10} et X_{11} sont déterminés respectivement à partir des équations (4.5) et (4.6). Pour cela l'algorithme calcule les matrices I'_{10} et I'_{11} avec :

$$
\begin{aligned}
I'_{10} &= I_{10} - L_{10}X_{00}\\
I'_{11} &= I_{11} - L_{10}X_{01}
\end{aligned}
$$

3. Cette résolution est alors appliquée récursivement sur les équations $L_{11}X_{10} = I'_{10}$ et $L_{11}X_{11} = I'_{11}$

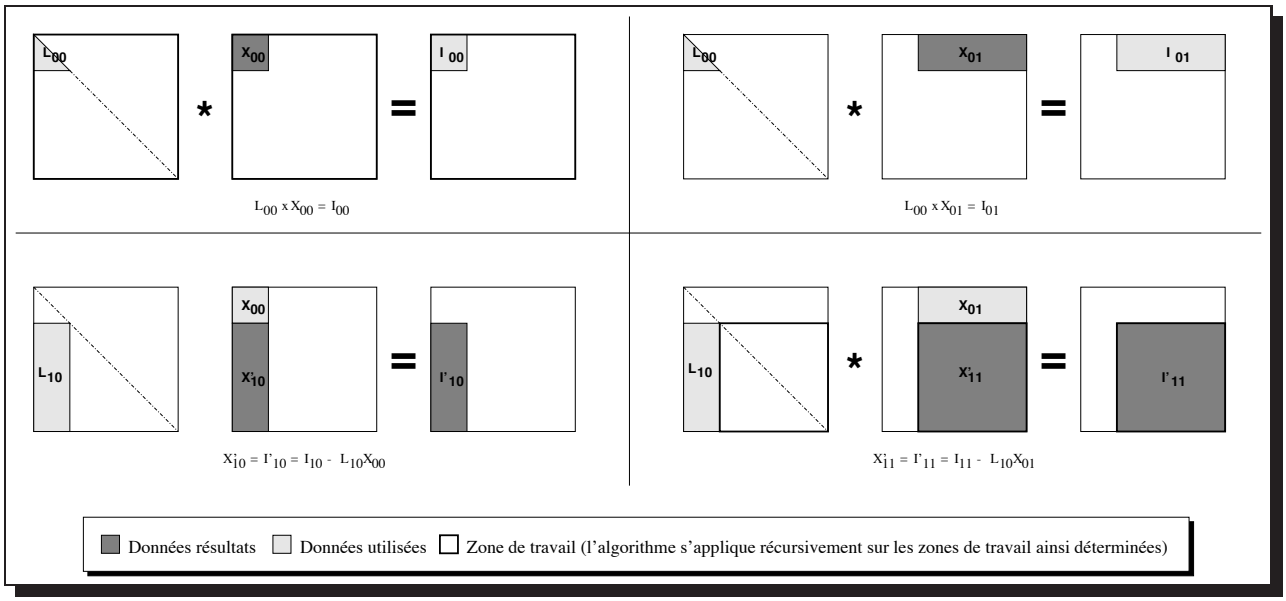

FIG. 4.3 – *Principe de la résolution triangulaire par bloc*

La méthode de résolution triangulaire parallèle utilisée pour résoudre l'équation 4.1 est ensuite appliquée de la même manière pour résoudre l'équation 4.2.

Dans ScaLAPACK la fonction pgdtrsm implémente cet algorithme. La matrice est distribuée selon le même principe que la factorisation LU, c'est-à-dire selon une distribution bloc cyclique.

L'exemple développé figure 4.4 est une substitution avant. Dans ce cas le calcul sur les blocs diagonaux s'effectue de la gauche vers la droite et du haut vers le bas. Dans le cas d'une substitution arrière l'algorithme est le même mais l'accès s'effectuera en sens inverse c'est-à-dire de la droite vers la gauche et du bas vers le haut.

L'étape 1 de l'algorithme (pdtrsm) est réalisée par une ligne de q processeurs. L'étape 2 (pdgemm) quant à elle est réalisée par l'ensemble des processeurs.

La figure 4.4 décrit plus précisément les différentes étapes de l'algorithme. L'étape 1 est implémentée par la fonction de ScaLAPACK pdtrsm, qui réalise une substitution triangulaire en parallèle sur une matrice carrée. Cette étape est effectuée sur une ligne de q processeurs. L'étape 2 est effectuée par ScaLAPACK avec la routine pdgemm, elle effectue la mise à jour du reste de la matrice par multiplication. Cette étape est réalisée par l'ensemble des processeurs. Pour chaque partie on distingue deux phases, une phase de communication et une phase de calcul.

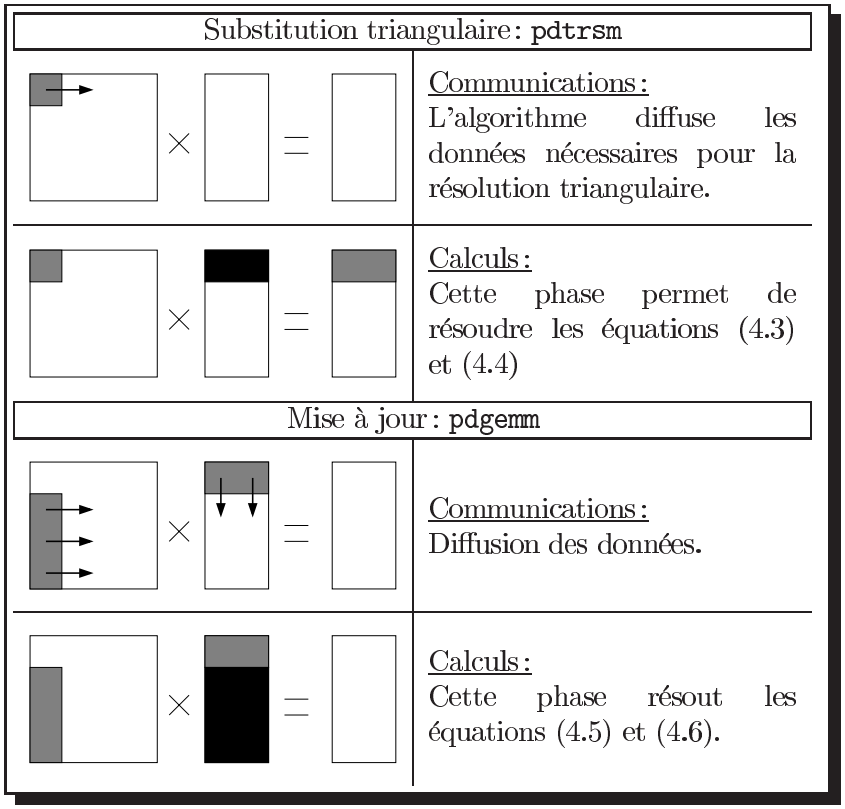

FIG. 4.4 – *Un niveau de la résolution triangulaire par bloc (substitution avant).*

4.3 Inversion matricielle *out-of-core*

4.3.1 Principe général

Le pseudo-code de cet algorithme présenté figure 4.5 fait apparaître les deux substitutions *out-of-core* utilisées dans le cadre de l'inversion. Selon le concept développé dans le prototype *out-of-core* de `ScaLAPACK` (cf. section 2.2), la résolution triangulaire *out-of-core* est effectuée super-bloc par super-bloc. L'algorithme doit effectuer deux substitutions entre deux matrices *out-of-core*. Prenons le cas de la substitution avant : la matrice X est calculée selon une décomposition par super-bloc. Pour chacun des super-blocs de X, la matrice L est entièrement lue. Cette lecture est réalisée selon le même type de décomposition. Ce qui revient à réaliser pour chaque super-bloc de X, une résolution triangulaire *out-of-core* avec la matrice L.

```
/* Programme d'inversion matricielle out-of-core */

Factorisation LU out-of-core (A=P.LU)
Création de la matrice identité I (PFDMATGENID)
Appliquer le pivot P sur la matrice I (PFDLAPIV)

Pour chaque super-bloc i de la matrice I
   Lecture du super-bloc i de la matrice I dans X
   Pour chaque super bloc L (parcours de gauche à droite)
      /* Substitution avant (L*I=X) */
      X = X/L
   Fin pour
   Pour chaque super bloc de U (parcours de droite à gauche)
      /* Substitution arrière (U*A^-1=X).*/
      X = X/U
   Fin pour
   Ecriture du super-bloc i de la matrice X
Fin pour
```

FIG. 4.5 – *Pseudo-code de l'inversion matricielle out-of-core*

Afin de minimiser l'occupation des matrices présentes en mémoire lors du traitement, seuls deux super-blocs sont nécessaires. La résolution de la décomposition LU et des deux résolutions triangulaires se réalisent dans le même espace d'adressage. En effet, les matrices X et I de l'algorithme sont en fait les mêmes matrices en mémoire lors de l'exécution.

4.3.2 Résolution triangulaire *out-of-core*

À présent nous allons décrire plus précisément la résolution triangulaire réalisée pour un super-bloc de la matrice X (figure 4.6)

Posons L_0, L_1,...,L_B les super-blocs de la matrice L. Et X_0, X_1,...,X_B les super-blocs de la matrice X. Les super-blocs sont décomposés en deux parties : le bloc diagonal (L_j^1, X_i^1 et I_i^1) et le reste du super-bloc (respectivement L_j^2, X_i^2 et I_i^2). La figure 4.6 montre une résolution triangulaire *out-of-core* sur un super-bloc.

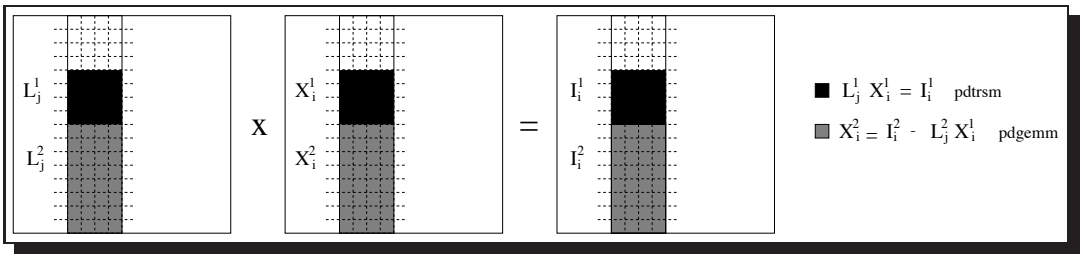

FIG. 4.6 – *Résolution triangulaire sur un super-bloc*

Dans un premier temps, l'algorithme réalise une résolution triangulaire sur la diagonale $L_j^1 X_i^1 = I_i^1$, en utilisant l'algorithme parallèle précédemment décrit dans la section 4.2. Dans un second temps, l'algorithme effectue une multiplication matricielle afin de mettre à jour X_i^2, avec $X_i^2 = I_i^2 - L_j^2 X_i^1$.

4.4 Prédiction des performances

Afin d'étudier les performances de l'inversion matricielle, nous avons réalisé une extension au modèle présenté section 3.4 en incorporant la modélisation de la résolution triangulaire.

4.4.1 Modélisation de l'inversion matricielle

Comme nous l'avons vu, pour effectuer l'inversion matricielle *out-of-core*, la factorisation LU *out-of-core* est suivie de deux résolutions triangulaires *out-of-core*. Les coûts de la substitution avant et de la substitution arrière sont les mêmes. Nous donnons ici le coût pour une résolution triangulaire donnée. Pour réaliser la substitution triangulaire, deux fonctions principales sont appelées. Une résolution triangulaire sur un bloc carré (`pgdtrsm`) et la mise à jour du reste de la matrice à l'aide d'une multiplication par bloc (`pdgemm`). Après avoir donné le coût des entrées/sorties, nous distinguerons pour chacune de ces fonctions deux coûts : les communications et le calcul.

Coût des accès disques

Les accès disques correspondent à la lecture et à l'écriture des super-blocs de la matrice résultat (4.7) et la lecture des super-blocs de la matrice triangulaire (4.8) :

$$S(NK\tau_{pq}^{io} + \beta_{pq}^{io}) \tag{4.7}$$

$$S\sum_{i=1}^{i=S}((N - K(i-1))K\tau_{pq}^{io} + \beta_{pq}^{io}) \tag{4.8}$$

Coût de la substitution triangulaire parallèle (`pgdtrsm`)

L'algorithme effectue S résolutions triangulaires pour chaque super-bloc de la matrice A^{-1}, soit S^2 substitutions triangulaires :

$$S^2 \left(\frac{B.(k^2K + kK - K)}{q} \alpha_t + \frac{\sum_{i=1}^{i=B-1} 2iKk^2}{pq} \alpha_g \right) \tag{4.9}$$

Les communications du `pgdtrsm` consistent à diffuser les blocs diagonaux sur la ligne de processeurs. L'algorithme envoie S^2 blocs diagonaux. Nous pouvons alors décomposer les coûts de communications en deux parties : le coût correspondant au super-bloc actif (4.10) et le surcoût *out-of-core* en communications (4.11).

$$S \sum_{i=1}^{i=B-1} \left(\frac{k^2}{2} \tau_p^q + \beta_p^q \right) \tag{4.10}$$

$$S(S-1) \sum_{i=1}^{i=B-1} \left(\frac{k^2}{2} \tau_p^q + \beta_p^q \right) \tag{4.11}$$

Les données communiquées pour la mise à jour du reste de la matrice sont diffusées sur l'ensemble des processeurs (4.12). Les coûts sont les mêmes que dans la version en mémoire.

$$S^2 \sum_{i=1}^{i=B-1} \left(ik^2 \tau_p^q + \beta_p^q \right) \tag{4.12}$$

Ajoutons la diffusion des colonnes :

$$S^2(B-1) \left(kK\tau_q^p + \beta_q^p \right) \tag{4.13}$$

Coût de la multiplication matricielle parallèle (`pdgemm`)

La mise à jour par multiplication matricielle est appliquée au dessous de la diagonale. Pour chaque super-bloc, la hauteur est diminuée de K :

$$S \sum_{i=1}^{i=S-1} \frac{(2iK^3)}{pq} \alpha_g \tag{4.14}$$

Les diffusions nécessaires pour la multiplication *out-of-core* sont envoyées sur la ligne du processeur. La largeur de ce bloc est de K. La hauteur est déterminée en fonction de la diagonale. Nous supposons que les diffusions s'effectuent par bloc. Le coût pour les super-blocs actifs est (4.15):

$$\sum_{i=1}^{i=S-1} \left(iBkK\tau_p^q + B\beta_p^q \right) \tag{4.15}$$

et le surcoût.*out-of-core* est (4.16):

$$(S-1) \sum_{i=1}^{i=S-1} \left(iBkK\tau_p^q + B\beta_p^q \right) \tag{4.16}$$

Le bloc de lignes $K \times K$ est diffusé sur la colonne de processeurs comme dans la version en mémoire (4.17) :

$$S \sum_{i=1}^{i=S-1} (BkK\tau_q^p + B\beta_q^p) \tag{4.17}$$

4.4.2 Validation expérimentale

Nous avons repris le même principe de validation expérimentale, présenté dans la section 3.4.3. La figure 4.7 donne les résultats comparatifs entre le temps d'exécution réel et le temps estimé par notre modèle (en *italique*) du programme. Le tableau offre également un comparatif similaire pour chacune des fonctions de l'algorithme d'inversion matricielle (dans le cas de la substitution triangulaire nous distinguons le temps de communication et de calcul).

Pour les calculs et les communications, le temps d'exécution est proche du temps estimé. Il y a quelques différences pour le temps des accès disques pour les mêmes raisons de fragmentations de fichier évoquées dans le chapitre précédent.

4.5 Analyse des surcoûts *out-of-core*

Dans le chapitre 3, nous démontrons qu'avec une distribution correcte et avec un recouvrement des accès disques par le calcul, il est possible de minimiser le surcoût *out-of-core* de l'algorithme: nous obtenons des performances d'une version en mémoire. Dans la suite de ce chapitre, nous montrerons que ce résultat s'étend à l'inversion matricielle *out-of-core*.

Le surcoût de l'inversion matricielle est donné par les équations (4.11) et (4.16) pour les communications et (4.8) pour les accès disques. Lorsque $K = N$ (c'est-à-dire $S = 1$) alors le coût est nul: c'est le temps d'exécution de l'algorithme en mémoire. Dans le paragraphe qui suit nous montrerons comment réduire ce surcoût.

4.5.1 Réduction du surcoût des communications

On retrouve le même type d'influence de la topologie sur le comportement des surcoûts des communications que pour la factorisation LU. Posons O_C le surcoût des communications avec $O_C = (4.11) + (4.16)$. La conclusion est identique que dans le chapitre précédent :

Lemme 2 *Si le nombre de colonnes q est égal à 1, alors* $O_C = 0$ *!*

La figure 4.8 montre l'influence de la topologie de la grille de processeurs sur les performances de l'inversion matricielle. La figure 4.9 illustre également nos propos avec un surcoût des communications nul pour une topologie 16×1.

4.5.2 Réduction du surcoût des Entrées/Sorties

Dans le cadre de l'inversion *out-of-core* le nombre d'accès disques est encore plus important que dans le cas de la factorisation LU. Il est donc intéressant d'appliquer les techniques de recouvrement exposés dans la section 3.5.2.

Soit $O_{IO} = (4.8)$ le surcoût des accès disques. Le surcoût *out-of-core* O^o_{IO} est recouvert dans le cas suivant :

Théorème 2 *Si le nombre de colonnes de processeurs q est égal à 1 et si*

$$pM \geq N\frac{t_{io}}{2\alpha}$$

alors

$$O^o_{IO} = 0$$

M	K	$p \times q$	LU	pgdtrsm IO	pgdtrsm Comm. Actif	pgdtrsm Comm Mise à jour	pgdtrsm Calcul	Temps d'exécution
12288	3072	1x8	29m39/*24m25*	10m16/*8m23*	3m6/*3m02*	8m13/*9m05*	21m21/*15m32*	1h36/*1h28*
1,2 Go / 4SB			693 Mflp/*844 Mflp*					856 Mflp/*928 Mflp*
		2x4	21m17/*18m30*	9m39/*8m23*	1m27/*1m32*	3m4/*3m25*	20m27/*15m41*	1h12/*1h08*
			983 Mflp/*1114 Mflp*					1128 Mflp/*1208 Mflp*
		4x2	18m50/*17m10*	8m52/*8m23*	2m5/*2m15*	2m25/*2m17*	19m50/*16m00*	1h08/*1h06*
			1088 Mflp/*1200 Mflp*					1200 Mflp/*1232 Mflp*
		8x1	19m57/*18m51*	8m9/*8m23*	5m22/*5m57*	4m5/*4m33*	19m50/*16m38*	1h21/*1h22*
			1019 Mflp/*1093 Mflp*					1016 Mflp/*1008 Mflp*
20480	2048	1x8	2h37/*2h08*	1h7/*46m37*	11m44/*8m21*	1h16/*1h15*	2h34/*1h11*	10h38/*8h06*
3,3 Go / 10SB			610 Mflp/*742 Mflp*					592 Mflp/*784 Mflp*
		2x4	1h06/*1h29*	1h2/*46m37*	4m28/*4m15*	27m27/*26m34*	2h05/*1h12*	6h23/*5h42*
			706 Mflp/*1067 Mflp*					992 Mflp/*1112 Mflp*
		4x2	1h50/*1h17*	1h2/*46m37*	5m53/*6m15*	11m55/*11m28*	1h56/*1h13*	6h21/*5h05*
			1208 Mflp/*1238 Mflp*					1000 Mflp/*1248 Mflp*
		8x1	1h43/*1h18*	55m46/*46m37*	13m45/*16m24*	13m41/*15m14*	1h49/*1h14*	6h25/*5h38*
			917 Mflp/*1212 Mflp*					984 Mflp/*1128 Mflp*
27648	1024	1x8	7h59/*7h58*	2h7/*3h25*	14m23/*14m59*	6h2/*6h38*	4h4/*2h56*	1j04h/*1j07h*
6,1 Go / 27SB			492 Mflp/*490 Mflp*					536 Mflp/*504 Mflp*
		2x4	4h34/*4h51*	1h59/*3h25*	7m16/*7m45*	2h3/*2h15*	3h51/*2h57*	16h45/*18h58*
			857 Mflp/*805 Mflp*					928 Mflp/*824 Mflp*
		4x2	3h20/*3h45*	2h2/*3h25*	9m50/*11m18*	39m41/*43m30*	3h47/*2h58*	12h43/*14h58*
			1174 Mflp/*1040 Mflp*					1224 Mflp/*1040 Mflp*
		8x1	3h02/*3h34*	2h4/*3h25*	26m35/*29m12*	26m53/*30m00*	3h46/*3h02*	12h29/*15h02*
			1290 Mflp/*1097 Mflp*					1248 Mflp/*1040 Mflp*

FIG. 4.7 – *Comparaison entre l'exécution expérimentale et le modèle théorique (en italique) du temps d'exécution et des performances de l'algorithme d'inversion matricielle out-of-core. M est l'ordre de la matrice, K la taille du super-bloc, p le nombre de processeurs sur une ligne de la grille, q le nombre de colonnes, S est le nombre de super-blocs. La taille de la matrice est en Gigaoctets et se trouve dans la première colonne. Les temps sont donnés en jours (j), heures (h), minutes (m) et secondes (s). La dernière colonne présente en Mflops (Mflp), les performances de l'exécution expérimentale, et, les prévisions du modèle théorique.*

FIG. 4.8 – *Performances théoriques de l'inversion matricielle. La courbe permet de comparer la version parallèle en mémoire (IC) et la version out-of-core (OoC).*

Nous proposons un recouvrement similaire pour la résolution triangulaire. Quand l'algorithme réalise ces calculs, c'est-à-dire la résolution triangulaire et la multiplication matricielle, entre L_j, X_i et I_i, le prochain super-bloc L_{j+1} est préchargé. Si le temps de préchargement est inférieur au temps de calcul, alors le surcoût des accès disques est nul. Nous devons cependant déterminer en fonction de nos ressources dans quelles conditions ce recouvrement est réalisable. Nous allons donc vérifier que le théorème 2 reste valide pour la résolution triangulaire et par conséquent pour l'inversion matricielle.

Preuve 2 *A partir de la modélisation et par l'équation 4.18 nous connaissons le temps de lecture d'un super-bloc à gauche. La hauteur du super-bloc est ici de H.*

$$(H-K)K\tau_{pq}^{io} + \beta_{pq}^{io} \tag{4.18}$$

Nous sous-estimons le temps de calcul de la mise à jour en ne considérant que la multiplication (`pdgemm`) qui représente la plus importante partie du temps de calcul.

$$\frac{2(H-1)K^2}{pq}\alpha_g \tag{4.19}$$

Ajoutons les coûts de communication. Nous supposons, selon le théorème 2, que nous avons une unique colonne de processeurs ($q = 1$) :

$$BkK\tau_q^p + B\beta_q^p \tag{4.20}$$

FIG. 4.9 – *Rapports des surcoûts pour les communications et les accès disques.*

À présent considérons la situation où les accès disques sont recouverts par le calcul (c'est-à-dire $O^o_{IO} = 0$*). Dans ce cas nous avons* $\frac{(4.19)+(4.20)}{(4.18)} \geq 1$*. Notons que si* $\frac{(4.19)}{(4.18)} \geq 1$ *alors* $\frac{(4.19)+(4.20)}{(4.18)} \geq 1$*. Le problème revient à déterminer pour quelle taille de super-bloc* K *le recouvrement est possible.*

$$\frac{\frac{2(H-1)K^2}{pq}\alpha_g}{(H-K)K\tau^{io}_{pq}} \geq 1 \tag{4.21}$$

$$\frac{H-1}{H-K} \times \frac{2K\alpha_g}{pq\tau^{io}_{pq}} \geq 1 \tag{4.22}$$

La première partie de l'expression est toujours supérieure à 1. Nous savons que $q = 1$ *et* $\tau^{io}_{pq} = \frac{t_{io}}{pq}$*. Nous pouvons fixer une borne pour* K *:*

$$\frac{2K\alpha_g}{t_{io}} \geq 1 \tag{4.23}$$

c'est-à-dire

$$K \geq \frac{t_{io}}{2\alpha_g} \tag{4.24}$$

Pour résumer, nous avons $K = pqM/N$ *et* $q = 1$*, si* $pM \geq N\frac{t_{io}}{2\alpha}$ *alors* $\frac{(4.19)}{(4.18)} \geq 1$*, en d'autres termes* $O^o_{IO} = 0$*.*

4.6 Optimisation par variation de la taille des super-blocs

Dans l'algorithme, la largeur du super-bloc actif et la largeur du super-bloc à gauche sont égaux. Une idée pour réduire la quantité de mémoire nécessaire serait de spécifier des tailles différentes pour ces super-blocs durant la phase de mise à jour, augmentant ainsi la largeur du super-bloc actif (et par conséquent le temps de calcul) et réduisant la largeur du super-bloc à gauche (et par conséquent réduire le temps d'accès disque). Le modèle précédemment écrit a été étendu pour permettre de spécifier des tailles différentes pour le super-bloc actif et le super-bloc de mise à jour. L'idée que nous avons avancée se vérifie. Plus la taille du super-bloc actif est importante, plus le temps pour réaliser l'inversion diminue (cf figure 4.10).

FIG. 4.10 – *Performances théoriques de l'inversion matricielle sur une grille 16x1. Le pourcentage représente la quantité de mémoire allouée pour le super-bloc actif (le reste étant destiné au super-bloc de mise à jour).*

4.7 Conclusion

Dans ce chapitre, nous avons présenté un algorithme d'inversion matricielle *out-of-core* ainsi que sa modélisation. Cet algorithme est dérivé de la factorisation LU *out-of-core* de `ScaLAPACK`. À partir de cette modélisation, nous avons pu isoler les surcoûts introduits par le traitement *out-of-core*. Comme pour la factorisation LU, nous avons observé que la meilleure topologie de grille virtuelle, pour supprimer les surcoûts en communication, était une colonne de processeurs. Nous avons ensuite montré qu'un recouvrement des accès disques par le calcul permettait de supprimer

le surcoût des accès disques. Nous avons déterminé que la mémoire nécessaire pour réaliser ce recouvrement est proportionnelle à la racine carrée de la taille de la matrice [19].

Afin de concrétiser ces résultats, considérons une petite grappe de 16 nœuds de PC-Celeron avec un réseau Fast-Ethernet. Pour inverser une matrice de 80 Go (une matrice d'ordre 100000) nous avons besoin de 26 Mo de mémoire par super-bloc (actif, à gauche et de pré-chargement) et par nœud, soit 78Mo par nœud. La modélisation prédit un temps de factorisation d'environ 11 jours sans recouvrement et de 9 jours avec. Si nous considérons qu'à la place des Celeron à 237 Mflops, nous disposons de processeurs AXP de Digital Alpha à 757 Mflops, alors dans ce cas la mémoire nécessaire par nœud est de 252 Mo. Le modèle prévoit alors 6 jours de calcul sans recouvrement et 4 jours avec (1,5 fois plus rapide). Avec la même topologie (c'est-à-dire une colonne de 16 processeurs), la version in-core donne les mêmes performances que la version *out-of-core* avec recouvrement. Avec une meilleure topologie (c'est-à-dire 4 colonnes et 4 lignes), l'algorithme in-core réalise l'inversion en 3 jours mais la mémoire nécessaire à ce traitement est de 5 Go soit 20 fois plus que la version *out-of-core*.

Chapitre 5

Recouvrement des accès disques sur grappes

Sommaire

5.1 Un mécanisme de recouvrement des E/S 71
5.2 Factorisation LU parallèle *out-of-core* avec recouvrement des accès disques 72
5.2.1 Principe algorithmique 72
5.2.2 Expérimentation et résultats 72
5.3 Inversion matricielle parallèle *out-of-core* avec recouvrement des accès disques 76
5.4 Conclusion 76

La problématique à l'étude de ce chapitre est de fournir un système de recouvrement des Entrées/Sorties par le calcul et les communications. Nous avons vu dans les chapitres 3 et 4 l'intérêt d'intégrer un tel mécanisme dans la factorisation LU *out-of-core* et dans l'inversion matricielle *out-of-core*. En effet, le recouvrement permet d'obtenir des performances équivalentes à un traitement en mémoire. La méthode de recouvrement adoptée est basée sur les sémaphores. Dans un premier temps nous expliquerons les raisons de ce choix, puis nous évoquerons le principe de mise en œuvre. La suite de ce chapitre expose alors la démarche scientifique entreprise pour obtenir un réel recouvrement des Entrées/Sorties. Dans un second temps, nous présenterons la factorisation LU *out-of-core* et l'inversion matricielle *out-of-core* dans lesquelles nous avons mis en place le système de recouvrement ainsi que les résultats obtenus.

5.1 Un mécanisme de recouvrement des E/S

La méthode de recouvrement est basée sur les sémaphores. Cette solution nous permet d'éviter l'utilisation des signaux qui entrent en conflit avec la bibliothèque de communication MPI. La même argumentation est utilisée quant à l'utilisation des threads. Pour des raisons conflictuelles liées aux signaux, les versions actuelles de MPI ne sont pas *thread-safe*. La commande système `clone` utilise des sémaphores pour communiquer. Le principe général consiste à cloner le processus courant afin d'obtenir deux processus. L'un dédié à la lecture ou à l'écriture disque, l'autre consacré aux calculs

et aux communications. La fonction de `clonage` (similaire à un `fork`) va dupliquer le processus en cours vers la fonction `handler_semlaread` qui elle-même lancera la commande d'accès disques.

```
__clone(handler_semlaread,malloc(65536)+65532,CLONE_VM|CLONE_FS|CLONE_FILES,NULL);
```

Les paramètres de la fonction `clone` sont les suivants :

- `CLONE_VM` précise que l'exécute du processus père et du processus fils se réaliseront dans le même espace d'adressage mémoire.
- `CLONE_FS` indique que le processus père et le processus fils partagent les mêmes informations sur le système de fichiers.
- `CLONE_FILES` précise que le processus père et le processus fils disposent de la même table de descripteurs de fichiers.

5.2 Factorisation LU parallèle *out-of-core* avec recouvrement des accès disques

5.2.1 Principe algorithmique

La factorisation LU *out-of-core* de `ScaLAPACK` peut se résumer en deux phases bien distinctes :

1. une phase d'Entrées/Sorties avec la lecture des données sur le disque.
2. une phase de calcul qui effectue les calculs à partir des données lues sur les disques.

Ces deux phases ont donc une dépendance forte mais s'effectuent en alternance. Comme nous l'avons démontré dans le chapitre 3 il est envisageable de se placer dans le cas où le temps de lecture est inférieur au temps de calcul. Le principe de recouvrement des Entrées/Sorties est illustré figure 5.1. Plus le nombre de phases à réaliser est important plus le gain offert par le recouvrement sera conséquent. Soit t_c le temps de calcul d'une phase et t_{io} le temps de lecture sur disque d'une phase. Pour effectuer la lecture et le calcul de n colonnes on aura un temps d'exécution de $T = n(t_c + t_{io})$ dans le cas synchrone. Et $T = t_{io} + nt_c$ avec $t_c \leq t_{io}$ dans le cas avec recouvrement. Soit $2n$ phases dans le premier cas et $n + 1$ phases dans le second cas.

5.2.2 Expérimentation et résultats

Premiers résultats

Afin de contrôler l'efficacité du recouvrement nous avons implémenté le mécanisme précédemment décrit. Deux types d'accès disques sont alors à différencier :

- Les Entrées/Sorties synchrone. Ce sont les accès disques qu'il n'est pas possible de recouvrir (c'est le cas par exemple du premier super-bloc de la matrice).
- Les Entrées/Sorties asynchrone. Ce sont les accès disques que nous pouvons recouvrir.

Comme nous l'avons vu, pour effectuer les préchargements le code doit diviser en deux le super-bloc. À partir de cette implémentation, nous avons étudié deux cas :

- une version synchrone. Le principe algorithmique est celui de la version avec recouvrement mais le mécanisme de recouvrement est inhibé. Cette version sert de code de comparaison pour évaluer la version asynchrone.
- une version asynchrone. C'est la version mettant en œuvre le concept présenté section 5.2.1.

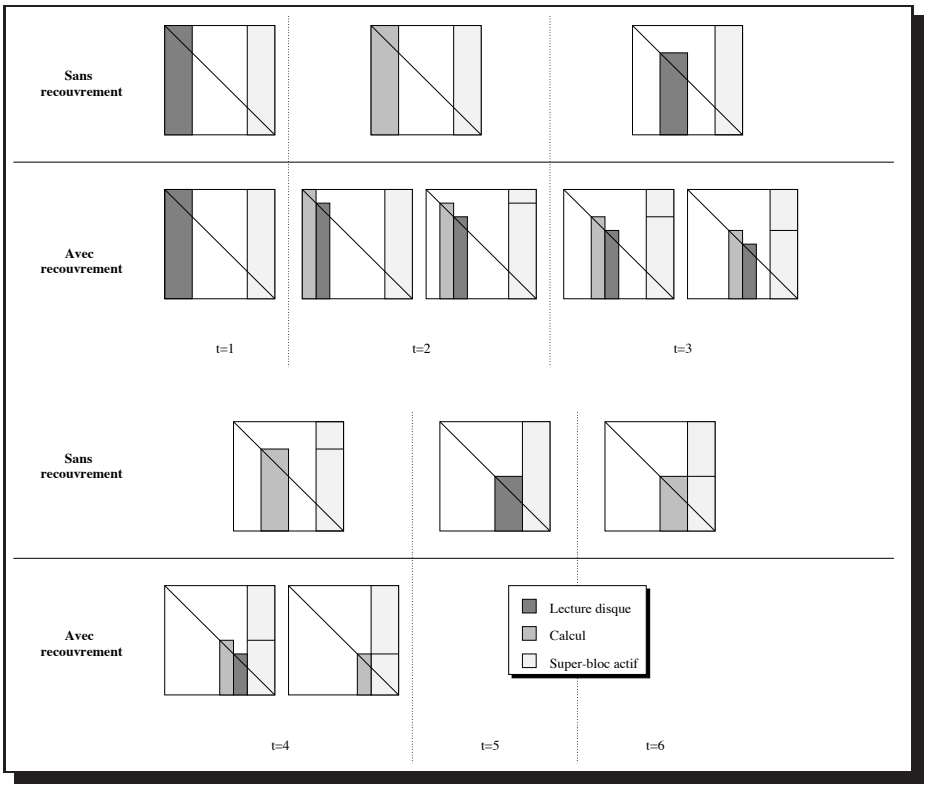

FIG. 5.1 – *Étapes de mise à jour de la factorisation LU sans recouvrement et avec recouvrement.*

Le tableau 5.1 présente ces différentes implémentations sur une matrice de 6Go (d'ordre 27648). Les résultats obtenus sur la version avec recouvrement ne correspondent pas aux résultats espérés. En effet, nous obtenons un temps similaire à la méthode sans recouvrement. Les lectures sont bien effectuées en parallèle du calcul en affichant un temps bien inférieur dans la version avec recouvrement, cependant ce gain de temps (de 48m 35s pour la matrice d'ordre 27648) est compensé par un surcoût d'exécution pour les communications et les calculs. Nous pouvons donc en conclure que l'architecture ne parvient pas à effectuer en parallèle les accès disques et les calculs. Même si un second processus est lancé pour gérer les accès disques, ce dernier requiert de la puissance CPU qui perturbe le reste de l'exécution. L'accès de type `Ultra-DMA`[1] et la lenteur du débit des disques laissaient pourtant présager un recouvrement possible.

Afin de comprendre le comportement de la gestion du contrôleur disque nous nous sommes livrés à l'expérimentation suivante : nous avons créé puis relu séquentiellement un fichier de 400Mo sur un contrôleur IDE et sur un contrôleur SCSI (voir tableau 5.2). Les résultats sont collectés par le système Linux via le fichier `/proc/interrupts`. Nous avons constaté une interruption tous les 500 octets dans le cas de l'IDE et une interruption tous les 4 Ko dans le cas du SCSI, ce qui nous donne 80 fois plus d'interruptions sur l'IDE et par conséquent 80 fois plus d'occupation CPU. Pour la matrice 27648×27648 on obtient plus de 32 minutes de consommation processeur pour le traitement des interruptions ce qui explique l'inefficacité de la version asynchrone.

1. Direct Memory Acces

	Synchrone	Asynchrone
E/S Synchrone	18m 51s	18m 41s
E/S Asynchrone	39m 43s	30s
Total des E/S	58m 34s	19m 11s
Total Communications	13m 33s	18m 06s
Total Calculs	2h 05m	2h 31m
Total Factorisation LU	3h 18m	3h 09m

TAB. 5.1 – *Matrice de taille 27648×27648 avec des super-blocs de taille 1024 sur une grille 8×1. La ligne E/S Asynchrone représente les accès disques qui peuvent être recouverts.*

	Celeron avec disque IDE	Alpha avec disque SCSI
Création d'un fichier de 400Mo	760998	6371
Lecture d'un fichier de 400Mo	889488	13959
Total des interruptions	1650486	20330
octets par interruption	500	41263

TAB. 5.2 – *Nombre d'interruptions générées par la lecture séquentielle puis l'écriture d'un fichier de 400Mo. Les résultats sont collectés par le système Linux via le fichier /proc/interrupts.*

La figure 5.2 développe comment le CPU est mis à contribution par la phase de calcul et par la phase d'accès disques via les interruptions. La phase de calcul occupe le même temps CPU dans le cas synchrone que dans le cas asynchrone. La phase d'accès disques est divisée en deux parties : les interruptions et les accès physiques au disque. Pour les interruptions, ces dernières requièrent la même consommation CPU dans les deux approches. Quant aux accès disques physiques, ils sont réalisés de manière contiguë et bénéficient alors des techniques d'anticipation du contrôleur IDE. Le temps de lecture physique est masqué par les interruptions et peut donc être considéré comme nul. Ce qui explique l'inefficacité du recouvrement qui comptait effectuer son gain sur ces lectures physiques.

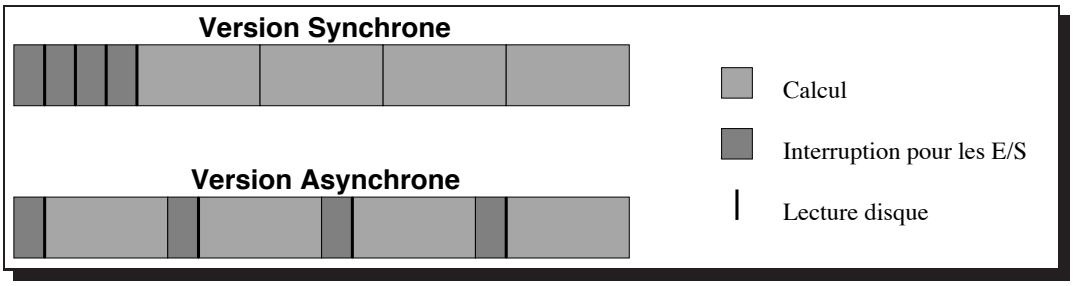

FIG. 5.2 – *Occupation du CPU dans le cas synchrone (sans recouvrement) et dans le cas asynchrone (avec recouvrement).*

Le problème étant identifié, il nous fallait comprendre la raison de ces si nombreuses interruptions et en réduire le nombre.

La technologie Ultra-DMA et Linux

Après la naissance du mode PIO[2] 4 (d'un débit de 16,6 Mo/s de débit synchrone maximum), l'interface EIDE adopte le DMA/33.

Définition 3 *Ultra-DMA : Ultra Direct Memory Access. Technologie de transfert de données entre un périphérique et la mémoire.*

Ce nouveau protocole a permis de multiplier par deux les performances du PIO 4 et d'obtenir ainsi un débit théorique de 33Mo/s en mode rafale (accès contiguë pour la tête de lecture). Son développement est facilité par le constructeur Quantum qui a rendu libre de droit la technologie [44, 64]. Les données peuvent être transmises lors d'un front montant mais également lors d'un front descendant.

En pratique, les PC et les disques durs doivent disposer d'une interface, d'un chipset et d'un BIOS, capables de gérer le mode DMA/33. Dans le cas contraire, le disque DMA/33 se comportera comme un PIO 4. Notons que, l'Ultra-DMA/66 permet théoriquement des transferts de 66 Mo/s.

Le DMA permet d'accélérer les requêtes au disque. La séquence d'instruction réalisée par le processeur pour accéder au disque devient une opération câblée. Le DMA effectue un double accès simultané au contrôleur de périphérique et à la mémoire. Il n'y a plus qu'une interruption en fin d'opération et non plus une interruption par mot transféré.

La gestion par le processeur d'une opération d'Entrée/Sortie en mode DMA s'effectue en deux étapes :

1. programmation du contrôleur de périphérique.
2. programmation du contrôleur de DMA. Ce dernier reçoit alors le numéro du canal, le type d'opération, l'adresse mémoire et la longueur de la zone.

L'intérêt de la norme Ultra-DMA réside dans l'utilisation du mode DMA qui décharge le processeur du contrôle de l'échange des données en permettant au contrôleur d'accéder directement à la mémoire sans avoir recours à ce dernier.

En principe, un noyau linux intégrant l'Ultra-DMA devrait automatiquement permettre le support de l'Ultra-DMA sur les disques et les interfaces adéquats. Or l'arrivée de ce protocole sous Linux date d'août 1998 avec le noyau 2.1.113 [83]. De plus, encore à ce jour, les transferts DMA sont par défaut désactivés pour éviter les conflits avec certains contrôleurs qui ne fonctionnent pas lorsque ce mode est activé.

Une commande sous Linux permet d'activer ou de désactiver les propriétés de la technologie Ultra-DMA. Nous avons découvert une solution système à ce problème via la commande hdparm qui permet d'activer ou de désactiver les propriétés de la technologie Ultra-DMA. La syntaxe pour activer l'Ultra-DMA du disque référencé par /dev/hda est :

```
hdparm -d 1 /dev/hda
```

Résultats avec le mode Ultra-DMA

On réitère alors l'expérience menée précédemment pour la lecture d'un fichier de 400Mo. Les résultats obtenus en activant le mode Ultra-DMA sont présentés dans le tableau 5.3. On obtient une lecture et une écriture 3,6 fois plus rapide.

À partir de ce constat, nous avons effectué une série de factorisations sur un cluster de 8 Celeron présenté tableau 5.4. L'optimisation proposée ne trouve son sens que sur des matrices

2. **P**rogrammed **I**nput/**O**utput

	Celeron avec disque IDE DMA passif	Celeron avec disque IDE DMA actif
Création d'un fichier de 400Mo	792044 *(124s)*	6503 *(35s)*
Lecture d'un fichier de 400Mo	889488 *(128s)*	8509 *(35s)*
Total des interruptions	1681532	15012
octets par interruption	499	55879

TAB. 5.3 – *Nombre d'interruptions générées par la lecture séquentielle puis l'écriture d'un fichier de 400Mo.*

de taille conséquente. On obtient un gain de 8 minutes pour une matrice de 6Go et un gain de 53 minutes pour une matrice de 13Go. Cependant on se heurte aux limites du recouvrement des Entrées/Sorties sur grappes avec architecture `Ultra-DMA`. En effet, si le recouvrement s'effectuait sans aucune consommation CPU le gain pour la matrice de 13Go passerait de 53 minutes à 1h heures et 55 minutes. Le surcoût des interruptions provoque donc une perte d'envioron 50%.

Une expérimentation identique menée sur une grappe d'Alpha possédant une architecture SCSI montre la supériorité, s'il est besoin, de ce type de contrôleur (cf tableau 5.5). La perte constatée n'est plus que de 40%.

5.3 Inversion matricielle parallèle *out-of-core* avec recouvrement des accès disques

L'inversion matricielle bénéficie du recouvrement intégré dans la factorisation LU. Le même mécanisme de recouvrement peut être incorporé dans les deux résolutions triangulaires en suivant la même approche algorithmique que pour la factorisation LU.

Les résultats obtenus présentés tableau 5.6, montrent un gain de 4% sur une matrice de 3,3Go. Il est important de préciser que le gain sera proportionnel à la taille de la matrice. Tout comme pour la factorisation LU, plus le nombre d'accès disques sera important, plus le gain sera important. Notons, que pour les raisons évoquées dans la section 5.2.2 le recouvrement n'est pas complet. Avec une technologie de recouvrement complet le gain serait alors de 16% soit 4 fois meilleur.

5.4 Conclusion

Les expérimentations effectuées ont permis de mettre en évidence une faiblesse de la technologie `Ultra-DMA` très répandue dans les architectures grappes. Le recouvrement des accès disques sur grappes de PC, muni de disques `Ultra-DMA` ne permet pas d'effectuer un recouvrement complet. On constate une consommation CPU ralentissant les calculs. Néanmoins le mécanisme de recouvrement que nous avons mis en place donne des résultats. Rappelons que l'intérêt de l'optimisation proposée ne prend son sens que pour le traitement de matrice de très grande taille. On remarque également, que le gain sur un traitement complet, factorisation ou inversion, est d'autant plus significatif pour l'utilisateur que la puissance processeur est élevée.

M	K		Synchrone dma off	Synchrone dma on	Asynchrone dma on
12288	**3072**	E/S Synchrone	3m 22s	1m59s	1m 51s
		E/S Asynchrone	2m 10s	1m 04s	0s
		Total des E/S	5m 32s	3m 03s	1m 51s
		Total Communication	2m 03s	2m 04s	2m 28s
		Total Calcul	12m 47s	11m 16s	11m 44s
		Total Factorisation LU	20m 22s	16m 23s	16m 03s
27648	**1024**	E/S Synchrone	18m 28s	7m 51s	7m 46s
		E/S Asynchrone	42m 23s	14m 0s	22s
		Total des E/S	1h 00m	21m 51s	8m 08s
		Total Communication	17m 55s	16m 27s	18m 40s
		Total Calcul	2h 05m	1h 55m	1h 58m
		Total Factorisation LU	3h 24m	2h 33m	2h 25m
40960	**1024**	E/S Synchrone	53m 05s	24m 09s	24m 49s
		E/S Asynchrone	4h 52m	1h 57m	18s
		Total des E/S	5h 45m	2h 21m	25m 07s
		Total Communication	1h 17m	56m 11s	1h 30m
		Total Calcul	10h 05m	8h 21m	8h 51m
		Total Factorisation LU	17h 08m	11h 39m	10h 46m

TAB. 5.4 – *Factorisation sur une grille 8×1 sur un cluster de Celeron. La ligne E/S Asynchrone représente les accès disques qui peuvent être recouverts.*

M	K		Version Synchrone	Version Asynchrone
12288	**3072**	E/S Synchrone	10m 12s	6m 18s
		E/S Asynchrone	5m 42s	0s
		Total des E/S	15m 54s	6m 18s
		Total Communication	1m 48s	3m 28s
		Total Calcul	11m 02s	13m 29s
		Total Factorisation LU	28m 44s	23m 15s
21504	**1536**	E/S Synchrone	11m 08s	9m 51s
		E/S Asynchrone	27m 02s	1s
		Total des E/S	38m 10s	9m 52s
		Total Communication	9m 38s	12m 09s
		Total Calcul	41m 29s	47m 20s
		Total Factorisation LU	1h 29m	1h 09m
27648	**1152**	E/S Synchrone	14m 55s	14m 32s
		E/S Asynchrone	49m 33s	7s
		Total des E/S	1h 04m	14m 19s
		Total Communication	26m 10s	34m 19s
		Total Calcul	1h 36m	1h 58m
		Total Factorisation LU	3h 07m	2h 47m

TAB. 5.5 – *Factorisation sur une grille 6×1 sur un cluster d'Alpha. La ligne E/S Asynchrone représente les accès disques qui peuvent être recouverts. Dans la version synchrone ces accès sont réalisés de manière synchrone.*

M	K		Version Synchrone	Version Asynchrone
10240	**2048**	E/S Synchrone TRSM	2m 48s	2m 48s
		E/S Asynchrone TRSM	9m 39s	0s
		Total des E/S	12m 27s	2m 48s
		Total Communication TRSM	9m 30s	11m 25s
		Total Calcul TRSM	1h 02m	48m 21s
		Total Factorisation LU	23m 05s	23m 46s
		Total Inversion	1h 47m	1h 25m
15360	**1536**	E/S Synchrone TRSM	8m 23s	8m 23s
		E/S Asynchrone TRSM	34m 18s	1s
		Total des E/S TRSM	42m 41s	8m 24s
		Total Communication TRSM	21m 43s	27m 08s
		Total Calcul TRSM	3h 33m	3h 27m
		Total Factorisation LU	1h 44m	1h 44m
		Total Inversion	5h 43m	5h 48m
20480	**1024**	E/S Synchrone TRSM	9m 03s	10m 15s
		E/S Asynchrone TRSM	1h 27m	13s
		Total des E/S	1h 36m	10m 28s
		Total Communication TRSM	38m 06s	50m 42s
		Total Calcul TRSM	7h 04m	6h 14m
		Total Factorisation LU	2h 55m	3h 01m
		Total Inversion	10h 43m	10h 17m

TAB. 5.6 – *Inversion matricielle sur une grille 4×1 sur une grappe de Celeron. La ligne E/S Asynchrone des résolutions triangulaires représente les accès disques qui peuvent être recouverts. Dans la version synchrone ces accès sont réalisés de manière synchrone.*

Troisième partie

Application

Chapitre 6

Scilab$^{OOC}_{//}$

Sommaire

6.1 Scilab **82**
6.1.1 L'interfaçage dans Scilab 82
6.2 Scilab$_{//}$ **84**
6.3 Interfaçage des PBLAS dans Scilab: niveau applicatif **85**
6.3.1 Nouveaux types Scilab$_{//}$ 86
6.4 Interfaçage du prototype *out-of-core* dans Scilab$_{//}$: niveau applicatif . **87**
6.4.1 Nouveaux types de Scilab$^{ooc}_{//}$ 87
6.5 Optimisations pour les traitements *out-of-core* **88**
6.5.1 Opérateurs et fonctions purs 88
6.5.2 Persistance des matrices *out-of-core* 88
6.5.3 Promotions, contagions et dégradations des matrices distribuées 90
6.6 Conclusions et travaux futurs **91**

La majorité des utilisateurs de calcul numérique ne sont pas des programmeurs. Le manque d'interface conviviale pour ScaLAPACK restreint le champ de ses utilisateurs. Scilab [51] offre quant à lui une interface utilisateur bien plus évoluée. Scilab est un logiciel libre pour le calcul scientifique, composé de centaines de fonctions mathématiques et d'un langage de programmation. L'interface propose également de nombreux formats d'importation et d'exportation des données. Un ensemble d'outils graphiques vient compléter ce logiciel pour offrir une application très complète pour le calcul scientifique.

L'alliance de ces deux outils permet à ScaLAPACK de bénéficier d'une interface utilisateur et d'un prototypage rapide ; elle permet également la parallélisation de Scilab. L'intégration d'un outil de traitement, d'objets de taille supérieure à la mémoire, trouve aussi sa justification dans sa simplicité de mise en œuvre : l'utilisateur pourra alors depuis sa station de travail manipuler des objets *out-of-core* distribués sur l'une ou l'autre des machines parallèles à sa disposition sans modification de code. Les capacités de calcul offertes par Scilab$_{//}$ étant ainsi étendues au delà de la capacité mémoire disponible.

Basé sur le projet en cours d'interfaçage de ScaLAPACK [38], ce chapitre présente la continuité de ces travaux en proposant d'interfacer le prototype *out-of-core* dans Scilab$_{//}$.

6.1 Scilab

Scilab est un logiciel développé par l'INRIA [1]. Scilab dispose d'un environnement riche et varié pour le calcul numérique. Le logiciel offre de nombreuses fonctions comme :

- la résolution de systèmes linéaires (les systèmes creux sont également pris en considération).
- le calcul de valeurs et vecteurs propres.
- différentes méthodes de résolution d'équations différentielles.
- la résolution d'équations non-linéaires.
- ...

Par ailleurs, Scilab dispose d'un environnement graphique (cf. figure 6.1). Scilab offre une librairie d'instructions graphiques de bas niveau, pour la réalisation de figures géométriques, récupérer les coordonnées du pointeur de la souris, etc. ; et une librairie de plus haut niveau, pour générer des courbes, gérer les surfaces, etc.

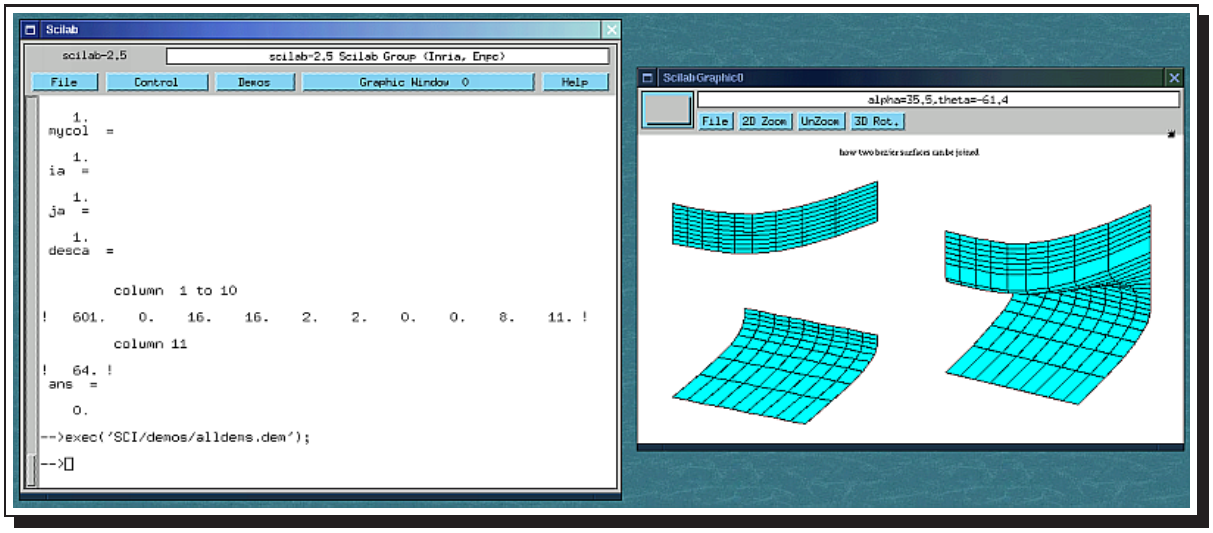

FIG. 6.1 – *Scilab*

Scilab repose sur un langage interprété, mais propose une alternative en offrant différents mécanismes pour intégrer des fonctions externes compilées. Cette fonctionnalité permet également d'accroître les fonctionnalités de Scilab en intégrant de nouvelles fonctions.

6.1.1 L'interfaçage dans Scilab

Une nouvelle fonction est intégrée, appelées primitives pour les distinguer des fonctions interprétées. Scilab propose quatre approches différentes pour créer une interface (cf. figure 6.2).

Interface mex

Ce type d'interfaçage est basé sur les mécanismes mexfile de MATLAB [63] pour éviter aux utilisateurs de cet outil de devoir se familiariser avec les mécanismes d'interface de Scilab.

1. **I**nstitut **N**ational de **R**echerche en **I**nformatique et **A**utomatique

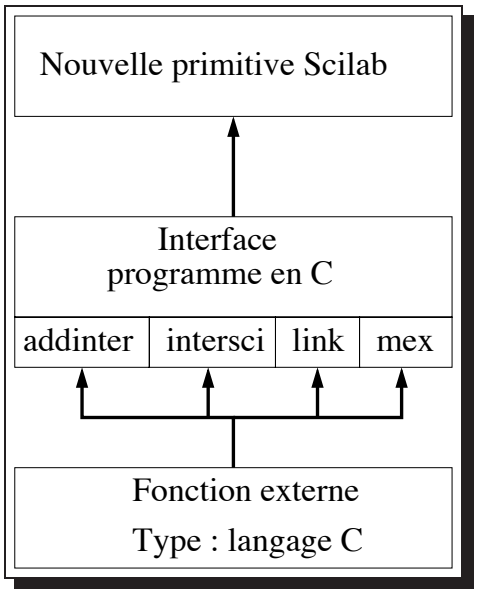

FIG. 6.2 – *Mécanisme d'interface entre une fonction externe (ici écrite en c) et une nouvelle primitive de* `Scilab`.

Interface via `addinter`

Dans le cas de la réalisation de l'interface avec `addinter`, l'utilisateur doit écrire deux fonctions. La fonction externe, effectuant le traitement souhaité et, une fonction permettant de lier l'appel et les données issues de `Scilab` avec la primitive. Pour cela la fonction effectuant l'interface est écrite par l'utilisateur en utilisant les fonctions de la librairie conçue à cet effet, ex : `CheckRhs, CheckLhs, GetRhsVar, CreateVar`. La fonction effectuant l'interface doit dans un premier temps contrôler que le nombre d'arguments en entrées et sorties sont correctes, c'est le rôle des fonctions `CheckLhs` et `CheckRhs`. Dans un second temps, la fonction d'interfaçage vérifie le type et la taille des paramètres. Pour finir, il faut récupérer un pointeur vers les données pour les transmettre à la fonction interfacée, c'est le rôle de la fonction `GetRhsVar`. Avant d'appeler la fonction on doit réserver une zone pour stocker le résultat, c'est le rôle de la fonction `CreateVar`. Enfin le fichier d'interfaçage fait appel à la primitive.

Pour prendre en compte cette interface ont doit alors compiler nos deux programmes, la fonction qui effectue l'interfaçage et la nouvelle primitive, et, utiliser l'utilitaire `addinter` qui va générer un script `Scilab` à lancer pour activer la nouvelle primitive.

Interface via `link`

Cette méthode permet d'interfacer une nouvelle primitive sans avoir à réaliser la fonction effectuant l'interface. La procédure en est simplifiée mais l'utilisation de la nouvelle primitive dans `Scilab` l'est moins. Le principe est le suivant :

1. On compile notre primitive en `.o`.
2. À l'aide de la commande `link` on charge dynamiquement la primitive dans `Scilab`.

```
--> n=link(SCI+'/routines/blas/dgemm.o','dgemm')
```

```
linking files /usr/lib/scilab-2.5/routines/blas/dgemm.o
to create a shared executable.
Linking dgemm (in fact dgemm_)
Link done
n =

  0.
```

3. La commande `call` ou `fort` permet d'appeler la primitive ainsi linkée. Elle est utilisée pour passer les valeurs des variables nécessaires à la primitive, mais également, précise leur type en C ou Fortran, et leur position dans la liste d'appel de la primitive. La syntaxe est de la forme :

```
[paramètres résultats]=fort('dgemm',description des entrées,
                            'out',description des sorties)
```

L'inconvénient de cette méthode est que l'appel via `call` ou `fort` ne correspond plus à un appel standard de `Scilab` et donne une syntaxe plus complexe.

Interface via `intersci`

`Intersci` est un outil permettant de réaliser automatiquement l'interface entre la nouvelle primitive et la fonction externe. Pour cela, l'utilisateur définit un fichier de description de l'interface avec le suffixe `.desc`. La commande `intersci <nom interface>.desc` va générer deux fichiers :

1. un fichier d'interface : `<nom.interface>.f`.
2. un fichier script `Scilab` : `<nom.interface>.sce`.

Ce sont alors ces deux fichiers qui seront utilisés pour interfacer la fonction dans `Scilab`. Nous avons utilisé cette méthode pour le développement de $\texttt{Scilab}^{ooc}_{//}$. L'utilisation d'`intersci` est affiné en prenant comme fichier à interfacer non pas la fonction `ScaLAPACK` mais un fichier intermédiaire en langage `c` effectuant l'appel à la fonction `ScaLAPACK`. Ce fichier intermédiaire permet d'une part un contrôle des paramètres, et d'autre part de passer les données par adresse entre `Scilab` et `ScaLAPACK`. Dans le cas *out-of-core*, il sera indispensable d'éviter une recopie mémoire des données.

6.2 $\texttt{Scilab}_{//}$

$\texttt{Scilab}_{//}$ est basé sur l'interface de `ScaLAPACK` dans `Scilab`. On utilise une programmation de type maître-esclaves avec un processus maître qui possède la fenêtre graphique et des processus esclaves qui sont en attente de commandes Scilab (cf. figure 6.3) . Dans l´interface `ScaLAPACK`, toutes les communications se font à l´aide de la bibliothèque de communication `BLACS` [86].

Dans la version *out-of-core*, la philosophie introduite dans l'interface `ScaLAPACK/Scilab` est respectée. On garde le même modèle de programmation de type maître-esclaves avec un processus maître qui possède la fenêtre graphique et des processus esclaves en attente de commandes $\texttt{Scilab}_{//}$.

Comme le montre la figure 6.4, notre proposition [16] utilise une instance de $\texttt{Scilab}_{//}$ comme console de développement et de prototypage. L'utilisateur de l'environnement lance des traitements en les décrivant sur la console. L'interpréteur distribue alors les traitements sur chaque nœud de la machine parallèle en utilisant l'interface de la bibliothèque de communication à passage de messages `PVM` [5, 46, 59] proposée par $\texttt{Scilab}_{//}$. Sur chacun des processeurs distants, un processus Scilab reçoit les commandes émises à partir de la console et les relaye par des appels à `ScaLAPACK` *out-of-core*.

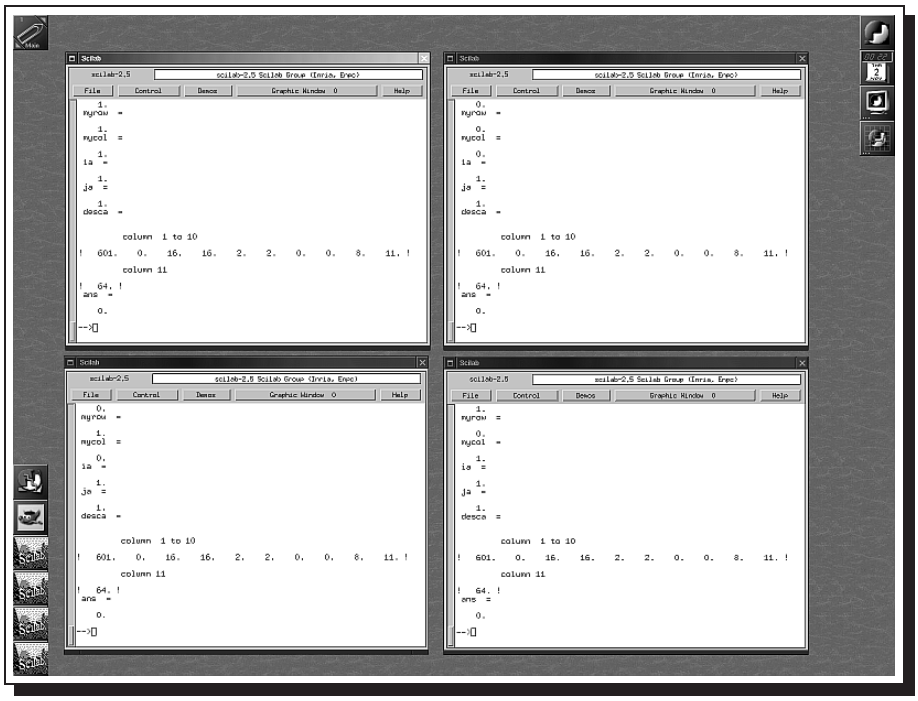

FIG. 6.3 – *Scilab*$_{//}$

Ce modèle, s'il nécessite la présence d'un Scilab relais sur chaque nœud de calcul, est certainement le moins coûteux du point de vue du développement. En effet, pour pouvoir utiliser une bibliothèque tel que le prototype `ScaLAPACK` *out-of-core*, il suffit d'en faire l'interface, afin de simplifier les appels et d'automatiser le démarrage sur tous les nœuds de l'exécution d'une fonction. La réalisation d'une interface est facilitée par des pré-processeurs syntaxiques de `Scilab`. Le coût de développement de notre environnement est faible car l'interface avec `ScaLAPACK` pourra être réutilisée.

Le développement de `Scilab`$_{//}$ est décomposé en trois niveaux d'accès au parallélisme : un niveau expert, un niveau applicatif, et un niveau intermédiaire à ces deux niveaux :

- Le **niveau applicatif** est le niveau offrant le plus de confort à l'utilisateur et nécessitant le moins de connaissances. Il permet entre autre la surcharge d'opérateur `Scilab` ; de plus, la distribution et l'équilibrage sont transparents pour l'utilisateur.
- Le **niveau expert** réclame un investissement et des connaissances plus larges de la part de l'utilisateur. La distribution est ici réalisée selon les directives de l'utilisateur et la gestion des machines est à la charge de l'utilisateur. Ce niveau nécessite l'accès aux structures de données internes (*dans notre cas, il s'agit des structures de* ***ScaLAPACK***). On bénéficie donc d'un contrôle plus important, mais l'utilisation est d'autant plus complexe.

6.3 Interfaçage des `PBLAS` dans `Scilab` : niveau applicatif

L'objectif de `Scilab` est d'offrir à l'utilisateur un outil évolué avec un haut niveau d'abstraction pour les types de données. L'utilisateur doit manipuler ces données en laissant à la charge du logiciel

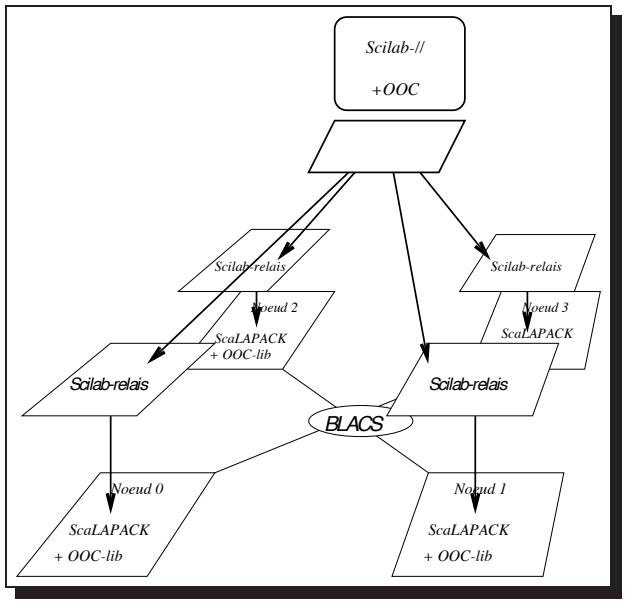

FIG. 6.4 – *Architecture de l'interface* `Scilab`$_{//}$ *out-of-core.*

le choix du type le plus adapté à son traitement : type distribué, type *out-of-core*, type creux.... Le niveau applicatif est en phase de développement, cependant une étude a déjà été menée sur les différents types à ajouter dans `Scilab` pour la mise en place de la version parallèle, brièvement décrite dans la section 6.3.1. Nous avons étendu cette étude menée par Eric Fleury à la version *out-of-core* dans la section 6.4.1.

6.3.1 Nouveaux types `Scilab`$_{//}$

Les types de `ScaLAPACK` utilisés par l'ensemble des fonctions de la librairie, doivent être intégrés dans la version actuelle de `Scilab` afin de mettre en place une corrélation entre ces deux outils. Quatre nouveaux types ont été créés :

1. Le type matrice distribuée qui décrit une matrice distribuée sur l'ensemble des processus Scilab.
2. Le type sous-matrice qui permet de décrire la sous-matrice d'une matrice distribuée.
3. Le type contexte qui décrit la forme de la grille virtuelle.
4. Le type distribution qui décrit la distribution d'une matrice dans un contexte : taille des blocs, type de distribution.

La suite de ce chapitre présente plus en détails ces différents types. Leur implémentation est par ailleurs fournie en annexe.

Type contexte

La forme de la grille virtuelle, ainsi que le contexte de communication `BLACS` qui lui est lié, est un concept important de `ScaLAPACK`. La forme réelle de cette grille, ainsi que la disposition des processus sur les processeurs, n'est gérée que de manière interne dans les `BLACS`, (cf. annexe B.1.1).

Le type contexte de Scilab (`ctxt`) contient les informations sur la forme de la grille, à savoir le nombre de lignes P et le nombre de colonnes Q, ainsi que le contexte `BLACS`. Ce dernier est ajouté automatiquement par l'interface.

Type distribution

À ne pas confondre avec le type *matrice distribuée*, le type distribution (`dist`) décrit une distribution générique d'une matrice pour son utilisation dans une bibliothèque de type `ScaLAPACK`, (cf. annexe B.1.2).

Type sous-matrice

Ce type (`sub`) permet de définir une sous-matrice à partir d'un élément (élément associé au coin supérieur gauche de la sous-matrice) et de la taille de la sous-matrice (cf. B.1.3). Ce type, défini dans le cadre de $\texttt{Scilab}_{//}$, pourra être utilisé dans le cadre *out-of-core*.

Type matrice distribuée

Ce type correspond à la définition même de la distribution d'une matrice particulière (cf B.1.4). Le type matrice distribuée est propre à chaque matrice.

6.4 Interfaçage du prototype *out-of-core* dans $\texttt{Scilab}_{//}$: niveau applicatif

Rappelons que le prototype *out-of-core* fournit un solver permettant d'effectuer trois types de factorisation : Cholesky, QR et LU. Ces résolutions s'effectuent à l'aide des fonctions `ScaLAPACK`, auxquelles on ajoute des fonctions d'accès disques.

6.4.1 Nouveaux types de $\texttt{Scilab}^{ooc}_{//}$

- Le type matrice distribuée *out-of-core* décrit une matrice *out-of-core* distribuée sur l'ensemble des processus Scilab.
- Le type distribution *out-of-core* décrit la distribution d'une matrice dans un contexte : taille des blocs, type de distribution, taille des super-blocs.

Type distribution *out-of-core*

Le type distribution (`dist_ooc`) décrit une distribution générique d'une matrice pour son utilisation dans une bibliothèque de type `ScaLAPACK` *out-of-core*. La lecture de la matrice sur le disque s'effectue par super-bloc, ce qui peut être assimilé à un niveau de distribution supplémentaire, comme nous l'avons illustré dans la section 2.2. La taille des super-blocs est donc ajoutée au type distribution (cf. annexe B.2.1).

Type matrice distribuée *out-of-core*

Tout comme son homologue en version *in-core*, ce type est la définition même de la distribution d'une matrice particulière. Il est propre à chaque matrice. Dans le cadre *out-of-core* trois informations supplémentaires seront à prendre en considération, le nom du fichier, le type du fichier et la taille mémoire locale allouée pour charger une partie de la matrice en mémoire (cf. annexe B.2.2).

6.5 Optimisations pour les traitements *out-of-core*

Néanmoins, une manipulation transparente des objets *out-of-core* telle que nous la proposons amène à considérer la spécificité de la problématique des traitements *out-of-core*, telles que la gestion des temporaires, la persistance des données et la promotion de type entre différents objets. De plus, notre modèle architectural se composant de machines hétérogènes, le problème de promotion de type est étendu au cas de la migration des données d'une machine à l'autre.

L'intérêt de conserver la présence des données en mémoire en cours de traitement est un facteur décisif dans les performances du calcul *out-of-core*. Ce facteur détermine en effet le nombre des accès disques. Nous proposons donc d'optimiser le code exécuté par le regroupement des boucles traitant les super-blocs successifs. Cette optimisation introduit une phase d'analyse de code afin d'offrir ces optimisations favorisant la fusion des boucles de manière automatique. Cette phase d'analyse pourra être délimitée par des séparateurs de début et de fin de phase donnés par l'utilisateur, ou être effectuée au vol, par le biais d'un retard de l'interprétation des instructions.

La figure 6.5 présente un exemple simple des pseudo-codes exécutés par une séquence d'instructions `Scilab`$_{//}$ s'exécutant sur des matrices *out-of-core*. Le code présenté a été écrit à des fins pédagogiques et a pour but d'exposer quelques optimisations possibles. Néanmoins, il existe des codes Scilab comprenant de telles instructions. Dans cet exemple, on part d'un code interprété brut auquel on apporte une analyse tel qu'un compilateur dédié pourrait l'effectuer.

Le regroupement des boucles par cette phase d'analyse, ainsi définie, permet d'obtenir un gain de lecture de 33% (4 lectures par super-bloc au lieu de 6) et un gain en écriture de 25% (3 écritures au lieu de 4). L'exemple illustre aussi le gain de place disque introduit par cette technique qui évite la création de temporaires *out-of-core*.

6.5.1 Opérateurs et fonctions purs

Cette phase d'analyse de code doit aussi être effectuée pour l'évaluation d'expressions faisant interagir plusieurs objets *out-of-core*. Pour les mêmes raisons que précédemment, il est en effet nécessaire de regrouper les traitements quand ceux-ci s'avèrent ne s'effectuer que localement à un super-bloc. On peut ainsi montrer qu'une sous-expression, ne comportant que des opérateurs purs (au sens HPF), ou des appels à des fonctions pures, peut être évaluée localement à un super-bloc. Par exemple, l'expression `A=B+C+D+ones(N,N)` ne générera l'exécution que d'une unique boucle parcourant les *super-blocs*.

L'interpréteur `Scilab`$_{//}$ pourrait être modifié pour prendre en compte cette analyse de code. Néanmoins, une solution alternative pourra éviter cette tâche : il suffira d'introduire un mécanisme de retardement des traitements effectifs, qui pourrait permettre d'analyser les expressions et les suites d'instructions jusqu'à la prochaine dépendance de données. Les traitements ainsi en attente peuvent être regroupés puis exécutés effectivement. On entre alors dans un processus de compilation.

6.5.2 Persistance des matrices *out-of-core*

La persistance des données allouées en mémoire est un problème crucial pour les environnements interactifs : entre deux instructions, les données référencées dans l'environnement doivent en effet rester accessibles. Dans le cas d'objets out-of-core, cette persistance doit s'appliquer non seulement sur la mémoire distribuée mais aussi sur les disques distribués. La gestion de la persistance en mémoire distribuée est optimisée dans le cas d'appel aux fonctions du prototype *out-of-core*. Ces techniques d'optimisation s'inspirent de celles développées pour l'optimisation de la gestion des caches [88]. Grâce à la phase d'analyse des expressions et instructions pour déterminer les

```
B=ooc(M,N,seed) // génération de 3 matrices
C=ooc(M,N,seed) // out-of-core de M×N
G=ooc(M,N,seed) // aléatoire
A=B+C
D=A+EYE()
T=A*G
A=G+3
D=G+D
```

(a) *Code Scilab.*

```
Pour chaque super-bloc I faire
  MEM(1)=LireSuperBloc(B[I])
  MEM(2)=LireSuperBloc(C[I])
  MEM(3)=AdditionInCore(MEM(1),MEM(2))
  EcrireSuperBloc(A[I],MEM(3))
FinPour
Pour chaque super-bloc I faire
  MEM(1)=LireSuperBloc(A[I])
  MEM(2)=[zeros((I-1)*nK,nK);eye(mK-(I-1)*nK,nK)]
  MEM(3)=AdditionInCore(MEM(1),MEM(2))
  EcrireSuperBloc(D[I],MEM(3))
FinPour
MultiplicationOutOfCore (A,G,T)
Pour chaque super-bloc I faire
  MEM(1)=LireSuperBloc(G[I])
  MEM(2)=AdditionInCore(MEM(1),3)
  EcrireSuperBloc(A[I],MEM(2))
FinPour
Pour chaque super-bloc I faire
  MEM(1)=LireSuperBloc(G[I])
  MEM(2)=LireSuperBloc(D[I])
  MEM(3)=AdditionInCore(MEM(1),MEM(2))
  EcrireSuperBloc(D[I],MEM(3))
FinPour
```

(b) *Pseudo-code d'interprétation.*

```
Pour chaque super-bloc I faire
  MEM(1)=LireSuperBloc(B[I])
  MEM(2)=LireSuperBloc(C[I])
  MEM(3)=LireSuperBloc(G[I])
  MEM(1)=AdditionInCore(MEM(1),MEM(2))
  MEM(2)=[zeros((I-1)*nK,nK);eye(mK-(I-1)*nK,nK)]
  MEM(4)=AdditionInCore(MEM(1),MEM(2))
  MEM(4)=AdditionInCore(MEM(3),MEM(4))
  EcrireSuperBloc(A[I],MEM(1))
  EcrireSuperBloc(D[I],MEM(4))
FinPour
MultiplicationOutOfCore (A,G,T)
Pour chaque super-bloc I faire
  MEM(1)=LireSuperBloc(G[I])
  MEM(1)=AdditionInCore(MEM(1),3)
  EcrireSuperBloc(A[I],MEM(1))
FinPour
```

(c) *Pseudo-code après analyse.*

FIG. 6.5 – *Analyse de code out-of-core dans* $\mathtt{Scilab}_{//}$.

dépendances de données, on peut détecter s'il est possible de trouver un ordonnancement, tel que le dernier super-bloc de données, chargé en mémoire lors d'une opération puisse être utilisé par l'opération suivante.

De plus, il faut gérer la persistance des données *out-of-core* sur disques distribués. L'espace de travail sur disque n'étant pas infini, il est nécessaire de disposer d'une politique de gestion de ce support. Cependant, dans le cadre d'un environnement interactif, on ne peut décider à la place de l'utilisateur si des données *out-of-core* doivent demeurer sur disque en vue d'une prochaine utilisation ou si elles peuvent être écrasées. Ce type de décision restera de la responsabilité de l'utilisateur, l'environnement ne fournissant alors que des primitives de création et de destruction d'objets *out-of-core* persistants.

6.5.3 Promotions, contagions et dégradations des matrices distribuées

L'environnement `Scilab`$_{//}$ manipulera différents types de matrices et prendra en charge les éventuelles promotions d'un objet vers un type différent. La figure 6.6 illustre les conversions de types qui devront être pris en compte. En effet, lorsque l'on effectue une opération entre deux opérandes de types différents, il faut déterminer d'une part si l'une, l'autre ou les deux opérandes doivent changer de type et d'autre part quel sera le type du résultat. On peut pour cela estimer que si une matrice est de type out-of-core, c'est qu'elle ne peut être stockée en mémoire et donc qu'elle ne peut changer de type. Ce sera donc la seconde matrice, selon les paramètres de la fonction appelée, qui sera promue vers le type *out-of-core*. Quant au résultat, il est nécessaire de disposer d'informations sur la fonction pour connaître sa façon d'affecter la taille des données (i.e. expansion, réduction...) et ainsi déterminer le type du résultat.

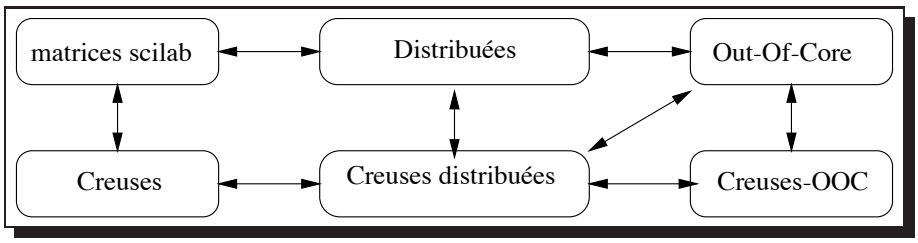

FIG. 6.6 – *Promotion des diffrents types de* `Scilab`

De plus, si l'utilisateur souhaite migrer ses données vers une autre des machines parallèles à sa disposition, il se peut que les caractéristiques des machines et en particulier leurs capacités mémoire soient différentes. Ceci engendrera une promotion ou une dégradation du type de la matrice. Pour gérer ce type de situation, il suffit de disposer d'une modélisation de l'architecture ciblée, et de comparer les informations extraites de ce modèle avec celles contenues dans l'objet *type (out-of-core ou distribué)* de la matrice pour savoir si ce type doit être modifié. Si ces informations sont disponibles la promotion pourra s'effectuer de manière automatique ; dans le cas contraire, la propagation devra être explicite et à la charge de l'utilisateur.

Ce polymorphisme des données est possible grâce à l'interactivité et à l'interprétation du code. Dans le cas d'applications compilées, la dynamicité des données des applications creuses interdit ce mécanisme, ou le rend très coûteux.

6.6 Conclusions et travaux futurs

L'utilisation d'un langage interprété comme `Scilab` est inhabituel pour les applications numériques intensives, car ces langages introduisent généralement un temps d'interprétation non négligeable par rapport au temps de calcul. Toutefois, le traitement interactif de données *out-of-core* n'est pas pénalisant, le temps d'interprétation devenant négligeable par rapport au temps de calcul résultant.

L'intégration d'une version *out-of-core* nécessite un interfaçage avec $\texttt{Scilab}_{//}$ peu coûteux. Sa construction basée sur `ScaLAPACK` qui est en phase d'intégration dans `Scilab` rend la tâche aisée. De plus, les fonctions de bases, ex : l'addition, la soustraction, la multiplication etc., restent à implémenter pour fournir un niveau applicatif de qualité et ainsi offrir le plus grand nombre d'opérations sur les matrices *out-of-core*. Cependant, deux problématiques majeures seront à prendre en considération : les performances et la promotion des types de données.

La première problématique concerne les performances du code interprétées. Dans le cadre du calcul *out-of-core* chaque calcul peut prendre plusieurs heures. Nous avons montré qu'il était possible de mettre en place une analyse de code. Le côté interprété, *d'un intérêt médiocre en out-of-core,* s'oriente alors vers un code compilé au profit de meilleures performances. Une perspective d'extension de $\texttt{Scilab}^{ooc}_{//}$ pour accroître les performances, serait de proposer un compilateur de haut niveau *nettement supérieure à l'exemple proposé.*

La seconde problématique est liée à la promotion d'un type de données vers un autre. Ce chapitre soulève les problèmes intrinsèques à la gestion des types : La promotion peut-elle être automatique ou doit- elle être explicite, et, dans quel cas ? Quels sont les directives spécifiant à `Scilab` qu'une promotion doit ou peut avoir lieu ? Une étude plus approfondie permettrait de définir les stratégies de promotion à mettre en place. De plus les modélisations, de la factorisation LU et de l'inversion matricielle présentées dans les chapitres 3 et 4, permettraient d'évaluer en fonction d'une situation architecturale donnée la distribution *out-of-core* la plus performante.

Conclusion

La première approche, afin de comprendre les mécanismes du calcul *out-of-core*, fut d'étudier le système de pagination. Nos évaluations, nous ont permis d'établir le constat suivant : le système de pagination n'est pas adapté au calcul numérique. Les raisons de l'inefficacité de la pagination dans ce cadre restaient à déterminer. C'est ainsi, que nous avons isolé ce problème sous le nom de l'effet trou. Les facteurs déclenchant ce phénomène sont multiples : accès linéaire en mémoire, multi-flots d'exécution ou encore nombreuses communications. Une fois ce problème isolé, nous avons présenté des solutions basées sur le réordonnancement des données afin de réduire les conséquences de l'effet trou. Nous avons également proposé un autre type de solution, via un outil de gestion de la mémoire virtuelle au niveau utilisateur. La librairie MMUMSEL nous a permis de proposer une optimisation de la factorisation LU *out-of-core*, en manipulant en cours d'exécution la politique d'évincement des pages dans le système de pagination.

La seconde approche était de nature algorithmique. Dans cette partie, nous avons mené une étude sur la librairie *out-of-core* de `ScaLAPACK`. Et plus spécifiquement sur la factorisation LU. Nous avons étudié et évalué les mécanismes de traitement *out-of-core* mis en place dans l'algorithme. Puis dans un second temps, et en vue de mettre en évidence une optimisation possible, nous avons modélisé l'algorithme de la factorisation LU. Ceci nous a permis de déterminer la meilleure topologie de grille à adopter ainsi que de déterminer la faisabilité et l'intérêt de concevoir un mécanisme de recouvrement des entrées/sorties. Les optimisations ainsi obtenues montrent qu'il est possible d'obtenir une exécution *out-of-core* similaire à une exécution en mémoire.

Ensuite, à partir de ces résultats nous avons étendu le prototype *out-of-core* de `ScaLAPACK` en proposant une inversion matricielle parallèle *out-of-core*. Et de la même façon que pour la factorisation LU, nous avons mené une évaluation et une modélisation de l'inversion matricielle. Conformément à la factorisation cela nous a permis de définir la topologie la plus efficace ainsi que l'intérêt d'utiliser le mécanisme d'entrées/sorties défini pour la factorisation LU. Et de la même manière, on obtient des performances proches d'une version en mémoire.

Le recouvrement des entrées/sorties, nécessaire à l'optimisation de la factorisation LU et de l'inversion est alors présenté. Dans ce chapitre, le lecteur aura pu comprendre la mise en œuvre du recouvrement ainsi que les modifications algorithmiques qui en découlent. Pour finir, les performances des versions améliorées sont également présentées.

Pour clore ce rapport, nous présentons nos travaux concernant l'intégration de `ScaLAPACK` au sein de `Scilab`. L'utilisateur bénéficie d'une interface conviviale pour gérer ces problèmes *out-of-core*. La puissance de `Scilab` au service de `ScaLAPACK` et la puissance de `ScaLAPACK` au service de

Scilab. Les travaux déjà entrepris pour l'intégration de ScaLAPACK permettent de traiter le cas du prototype *out-of-core* pour un faible coût de mise en œuvre.

Perspectives

Deux types de factorisation *out-of-core* sont disponibles dans le prototype de ScaLAPACK. Forts de nos résultats sur la factorisation LU et l'inversion, il est envisageable d'espérer obtenir des optimisations pour la factorisation de Cholesky *out-of-core* et la factorisation QR *out-of-core*.

Par ailleurs, le taux de recouvrement des accès disques n'est pas total, le processeur reste sollicité lors des accès disques. Une étude plus complète sur les solutions de recouvrement de ces accès disques permettrait de définir des stratégies de recouvrement plus efficaces. Dans cet esprit, nous pourrons également étudier le comportement et les performances de la technologie fibre channel [34].

Enfin, La version de $\text{Scilab}^{ooc}_{//}$ est en plein développement. De nombreuses fonctions *out-of-core*, addition, multiplication, transposée, extraction, calcul des valeurs propres, etc., restent encore à développer et à intégrer. Par ailleurs, le développement de l'interface *out-of-core* de Scilab au niveau expert nécessitera une étude complète combinant à la fois analyse de code et analyse du prototypage.

À plus long terme, on pourra envisager d'associer les travaux menés dans le cadre de la compilation *out-of-core* [9, 24] avec la librairie de gestion de mémoire virtuelle au niveau utilisateur MMUM. L'objectif étant de proposer un système de pagination capable de s'adapter au mieux aux problèmes qui lui sont soumis, qu'ils soient *in-core* ou *out-of-core*.

Il reste un point que nous n'avons pas développé : les systèmes de fichiers parallèles. De nombreuses approches existent, citons PVFS [12], MPI-IO [79], BPFS [76], Galley [68], PIOUS [66], Vesta [28] ou encore PASSION [24]. Après étude, il peut être intéressant de voir comment nous pourrions utiliser ces systèmes de fichiers parallèles pour améliorer encore les performances de nos applications *out-of-core*.

Pour terminer, de nombreux domaines autres que le calcul numérique hautes performances nécessitent des accès intensifs aux entrées/sorties. Le *data-mining* [82, 67] ou toutes les applications utilisant des systèmes de vidéo à la demande [4, 52, 53] en sont deux exemples concrets. Il serait intéressant d'analyser les contraintes en entrées/sorties de ces applications et de voir si des solutions communes existent.

Quatrième partie

Annexe

Annexe A

Commandes ScaLAPACK

Sommaire

A.1 Commandes BLACS 97
A.2 Commandes PBLAS 98
A.3 Commandes du prototype *out-of-core* 99
A.4 Contribution au prototype *out-of-core* 102

Cette annexe n'est pas une liste exhaustive des commandes de ScaLAPACK, mais présente les principales fonctions utilisées dans nos développements. De plus, on peut y trouver le prototypage des fonctions *out-of-core* que nous avons réalisées (cf. section A.4).

A.1 Commandes BLACS

Connaître son identité

```
CALL BLACS_PINFO( IAM, NPROCS )
```

Cette fonction retourne dans l'entier IAM le numéro du processeur lorsque le programme est exécuté sur NPROCS processeurs.

Contexte de communications BLACS

```
CALL BLACS_GET( -1, 0, ICTXT )
```

Le contexte permet de créer un groupe de processeurs, c'est l'équivalent d'un communicateur sous MPI. Il est possible de créer plusieurs contextes. Ces derniers peuvent être disjoints et donc indépendants dans les communications. On peut également réaliser un recouvrement de processeurs.

Création d'une grille de processeurs

```
CALL BLACS_GRIDINIT( ICTXT, 'Row-major', NPROW, NPCOL )
```

Cette fonction va créer une grille de NPROW×NPCOL processeurs. La numérotation des processeurs s'effectue en ligne dans le cas d'un appel avec le paramètre *row-major* et en colonne dans le cas d'un appel avec le paramètre *column-major* (figure A.1).

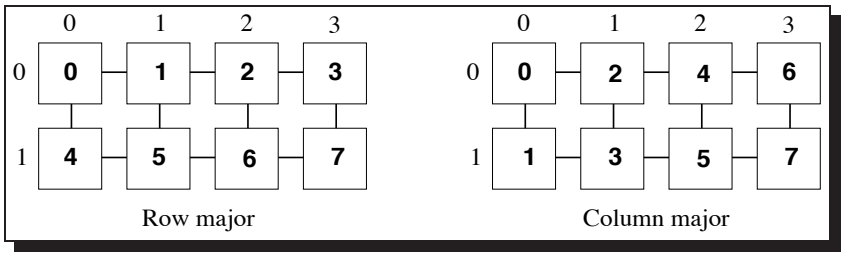

FIG. A.1 – *Huit processeurs virtualisés sur une grille 2x4*

Localisation sur la grille de processeurs

```
CALL BLACS_GRIDINFO( ICTXT, NPROW, NPCOL, MYPROW, MYPCOL )
```

Cette fonction permet de récupérer les coordonnées du processeur sur la grille `NPROW`×`NPCOL` dans les entiers `MYPROW` et `MYPCOL`.

Barrière de synchronisation

```
CALL BLACS_BARRIER(ICTXT, 'All')
```

À l'appel de cette fonction l'exécution est suspendue jusqu'à ce que tous les processeurs soient parvenus à ce point.

Fermeture des contextes

```
CALL BLACS_EXIT(0)
```

Ferme tous les contextes de communication.

A.2 Commandes PBLAS

Création d'un descripteur

Pour effectuer le traitement d'une matrice, `ScaLAPACK` a besoin de diverses informations concernant la matrice. Toutes ces informations sont stockées dans **le descripteur de la matrice.**

in-core

```
CALL DESCINIT( DESCA, M, N, MB, NB, RSRC, CSRC, CTXT, LDA, INFO )
```

Les paramètres utilisés pour la création du descripteur `DESCA` sont décrits dans le tableau A.1. La variable `INFO` permet de récupérer le code d'erreur en cas de problème.

out-of-core

```
CALL PFDESCINIT(  DESCA, M, N, MB, NB, RSRC, CSRC, CTXT,
                  IODEV, FILETYPE, MMB, NNB, ASIZE, FILENAME, INFO)
```

Pour la version *out-of-core*, des paramètres supplémentaires au tableau A.1 sont nécessaires. Ce sont les informations utiles à la création du périphérique (*device*). Le numéro du périphérique, qui va être créé, ouvrira un flux d'accès au fichier avec les informations suivantes :

- `FILETYPE` correspond au type du fichier (D pour Distribué, S pour partagé).
- `FILENAME` est le nom du fichier sur le disque.
- `MMB` est le nombre de lignes du super-bloc.
- `NNB` est le nombre de colonnes du super-bloc.

DESC()	Nom symbolique	Portée	Définition	in-core	ooc
1	DTYPE_A	(global)	Type du descripteur :	1	601
2	CTXT_A	(global)	Identificateur du contexte indiquant la grille de processeurs sur laquelle la matrice globale A est distribuée.	√	√
3	M_A	(global)	Nombre de lignes de la matrice globale A.	√	√
4	N_A	(global)	Nombre de colonnes de la matrice globale A.	√	√
5	MB_A	(global)	Nombre de lignes du bloc pour effectuer la décomposition.	√	√
6	NB_A	(global)	Nombre de colonnes du bloc pour effectuer la décomposition.	√	√
7	RSRC_A	(global)	Numéro de ligne du processeur où est mappé la première ligne de la matrice globale A.	√	√
8	CSRC_A	(global)	Numéro de colonne du processeur où est mappé la première colonne de la matrice globale A.	√	√
9	LLD_A	(local)	*Leading dimension* du tableau local.	√	√
10	IODEV_A	(global)	Numéro de device associé à la matrice *out-of-core* A.		√
11	ASIZE	(local)	Quantité de mémoire locale disponible pour la factorisation de A.		√

TAB. A.1 – *Descripteur de matrice in-core et out-of-core (ooc). Dans le cas du in-core le descripteur est un tableau à 9 entrées, dans le cadre du out-of-core le tableau est à 11 entrées.*

A.3 Commandes du prototype *out-of-core*

Générer une matrice pseudo-aléatoire

Lorsque le descripteur est créé, on peut générer une matrice distribuée pseudo-aléatoire.

in-core

```
CALL PDMATGEN(  CTXT, AFORM, DIAG, M, N, MB, NB, A, LDA,
                IAROW, IACOL, ISEED, IROFF, IRNUM, ICOFF, ICNUM,
                MYROW, MYCOL, NPROW,NPCOL)
```

- AFORM définit le type de la matrice A.
  a. AFORM='S' : Génération d'une matrice symétrique.
  b. AFORM='H' : Génération d'une matrice hermitian.
  c. AFORM='T' : Génération de la matrice A et retourne sa transposée.
  d. AFORM='C' : Génération de la matrice A et retourne son conjugué.
  e. AFORM dans les autres cas on obtient la génération d'une matrice aléatoire[1].
- DIAG='D' : la matrice est à dominante diagonale. Ce n'est pas le cas pour toute autre valeur de DIAG.
- IAROW : Numéro de ligne du processeur sur lequel le premier bloc de A est distribué.
- IACOL : Numéro de colonne du processeur sur lequel le premier bloc de A est distribué.
- ISEED : Graine de génération aléatoire des valeurs.
- IROFF : Le nombre de lignes locales de A qui ont déjà été générées. (Dans le cas du traitement de la matrice complète ce paramètre est à zéro). IROFF doit être un multiple de MB.
- IRNUM : Nombre de lignes localement générées.
- ICOFF : Le nombre de colonnes locales de A qui ont déjà été générées. (Dans le cas du traitement de la matrice complète ce paramètre est à zéro). ICOFF doit être un multiple de NB.
- ICNUM : Nombre de colonnes localement générées.
- MYROW : Coordonnée ligne du processeur appelant (variable locale).
- MYCOL : Coordonnée colonne du processeur appelant (variable locale).
- NPROW : Nombre de processeurs sur la ligne de la grille.
- NPCOL : Nombre de processeurs sur la colonne de la grille.

out-of-core

```
CALL PFDMATGEN(  IODEV, CTXT, AFORM, DIAG, M, N, MB, NB, A, ASIZE,
                 IAROW, IACOL, ISEED, MYROW, MYCOL, NPROW,NPCOL,INFO)
```

On retrouve les mêmes informations pour la version *out-of-core* auxquelles viennent s'ajouter le périphérique (IODEV) et la mémoire *in-core* localement disponible (ASIZE).

Afficher une matrice

in-core

```
CALL PDLAPRNT( M, N, A, IA, JA, DESCA, IRPRNT, ICPRNT, CMATNM, NOUT, WORK)
```

1. Fonctionnalité annoncée mais non implémentée à ce jour

- IA : Première ligne de la matrice globale A.
- JA : Première colonne de la matrice globale A.
- IRPRNT : Coordonnée ligne du processeur réalisant l'affichage.
- ICPRNT : Coordonnée colonne du processeur réalisant l'affichage.
- CMATNM : Texte de description de la matrice à afficher avant le contenu de la matrice.
- NOUT : Unité de sortie fortran (6 pour l'écran, 0 pour la sortie erreur).
- WORK : Tableau local de travail nécessaire à l'affichage. Ce tableau est de taille suffisante pour contenir au moins un bloc.

out-of-core
Bien que l'affichage d'une matrice *out-of-core* à l'écran puisse paraître hors de propos. Cette fonction à titre de vérification et correction de code peut s'avérer fort utile :

```
CALL PFDLAPRNT( M, N, A, IA, JA, DESCA, IRPRNT, ICPRNT, CMATNM, NOUT, WORK )
```

Les paramètres ne diffèrent pas de la version *in-core*.

Factorisation de Cholesky

in-core

Pour effectuer une factorisation de Cholesky sur une matrice carrée d'ordre $M \times M$ on utilisera la fonction suivante :

```
CALL PDPOTRF( UPLO, M, A, IA, JA, DESCA, INFO )
```

- UPLO :spécifie le type de matrice triangulaire.
  1. UPLO='U' : Le résultat de la factorisation est stocké dans la matrice triangulaire supérieure ($A = U' * U$).
  2. UPLO='L' : Le résultat de la factorisation est stocké dans la matrice triangulaire inférieure ($A = L * L'$).

Attention, la matrice à traiter doit être obligatoirement carrée.
out-of-core

```
CALL PFDPOTRF( UPLO, N, A, IA, JA, DESCA, INFO )
```

Les paramètres ne diffèrent pas de la version *in-core*. Pour plus d'informations sur les mécanismes algorithmiques de cette factorisation le lecteur pourra se référer à [47, 33].

Factorisation LU.

in-core

```
CALL PDGETRF( M, N, A, IA, JA, DESCA, IPIV, INFO )
```

– IPIV: Vecteur pivot distribué. Le résultat de la factorisation LU est du type LU=PA où P est le vecteur d'échange des lignes.

out-of-core

```
CALL PFDGETRF( M, N, A, IA, JA, DESCA, IPIV, INFO)
```

Les paramètres ne diffèrent pas de la version in-core. Pour plus d'informations sur les mécanismes algorithmiques de cette factorisation le lecteur pourra consulter le chapitre 3.

Fermer le périphérique associé à une matrice

out-of-core

```
CALL LACLOSE( IODEV, 'Keep', MYID, NPROCS, INFO )
```

Ferme le périphérique associé à une matrice *out-of-core*. Le second paramètre indique si le fichier doit être gardé sur disque après fermeture ('Keep') ou non ('NoKeep').

A.4 Contribution au prototype *out-of-core*

Factorisation LU *out-of-core* avec recouvrement des accès disques

```
CALL PFDGETRF_PALADIN( M, N, A, IA, JA, DESCA, IPIV, INFO)
```

Fonction similaire au PFDGETRF mais qui offre un mécanisme permettant de recouvrir les accès disques.

Résolution triangulaire *out-of-core* avec recouvrement des accès disques

```
CALL PFDTRSM_PALADIN(  SIDE, UPLO, TRANS, DIAG, M, N, ALPHA, A, IA,
                       JA, DESCA, B, IB, JB, DESCB, INFO)
```

– SIDE [entrée] spécifie l'ordre de prise en compte de A par rapport à X (à droite ou à gauche de X).
  a. SIDE='L': la résolution triangulaire effectuée sera A*X=B.
  b. SIDE='R': la résolution triangulaire effectuée sera X*A=B.
– UPLO [entrée] spécifie le type de la matrice triangulaire A.
  a. UPLO='U': A est une matrice triangulaire inférieure.
  b. UPLO='L': A est une matrice triangulaire supérieure.
– TRANSA [entrée] spécifie le type d'opération appliqué sur A.
  a. TRANSA='N': Réalise le calcul sur A.
  b. TRANSA='T': Réalise le calcul sur la transposée de A.
  c. TRANSA='C': Réalise le calcul sur la transposée de A.
– DIAG [entrée] spécifie si la matrice A est triangulaire unitaire (prise en compte de la diagonale ou non).
  a. DIAG='U': A est triangulaire unitaire.

b. DIAG='N' : A n'est pas triangulaire.

- M [entrée] est le nombre de lignes de B.
- N [entrée] est le nombre de colonnes de B.
- ALPHA [entrée] spécifie le scalaire α appliqué sur la résolution AX=αB.
- A [entrée] est une matrice out-of-core.
- IA [entrée] : Première ligne de la matrice globale A.
- JA [entrée] : Première colonne de la matrice globale A.
- DESCA [entrée] : Descripteur de la matrice A.
- B [entrée/sortie] matrice *in-core*. A la sortie, la matrice B est écrasée par la solution X.
- IB [entrée] : Première ligne de la matrice B.
- JB [entrée] : Première colonne de la matrice B.
- DESCB [entrée] : Descripteur de la matrice *in-core* B.

Générer une matrice identité

Afin de réaliser l'inversion matricielle (chapitre 4), nous avons implémenté une fonction *out-of-core* permettant de générer une matrice identité distribuée *out-of-core*.

```
CALL PFDMATGENID(  IODEV, ICTXT, M, N, MB, NB, A, ASIZE, IAROW, IACOL,
                   MYPROW, MYPCOL,NPROW, NPCOL, INFO )
```

On retrouve les paramètres utilisés dans le cadre d'une création de matrice pseudo-aléatoire.

Comparaison de matrice *out-of-core*

```
PFDCOMP( M, N, A, B, IA, JA, DESCA, DESCB)
```

- M est le nombre de lignes de la matrice A et de la matrice B.
- N est le nombre de colonnes de la matrice A et de la matrice B.
- A est une matrice *out-of-core*.
- B est une matrice *out-of-core*.
- IA : Première ligne des matrices A et B.
- JA : Première colonne des matrices A et B.
- DESCA : descripteur de la matrice A.
- DESCB : descripteur de la matrice B.

Cette fonction effectue la comparaison de deux matrices *out-of-core* en effectuant le calcul de la norme des deux matrices :

Définition 4 *Deux matrices A et B de taille $M \times N$ sont identiques si la distance entre les deux matrices est inférieure à ε. On considère que deux matrices sont strictement identiques pour un ε de l'ordre de 10^{-9}. Ce qui nous donne la relation :*

$$\sqrt{\sum_{i=0}^{i=M}\sum_{j=0}^{j=N}(A(i,j)-B(i,j))^2} \leq \varepsilon$$

.

Cette fonction fut développée afin de nous permettre de vérifier la validité numérique de nos résultats.

Conversion *in-core/out-of-core*

Cette fonction convertit une matrice de type *in-core* en matrice de type *out-of-core*. La fonction inscrit les données de la matrice A sur disques et les associe à la matrice *out-of-core* B.

```
CALL PFDDIST ( A, DESCA, B, DESCB, INFO)
```

- `A` est une matrice *in-core*.
- `B` est une matrice *out-of-core*.
- `DESCA` : descripteur de la matrice A.
- `DESCB` : descripteur de la matrice B.

Cette fonction fut développée dans le cadre du développement de $\texttt{Scilab}^{ooc}_{//}$ (chapitre 6).

Annexe B

Prototypage de Scilab

Sommaire

B.1 Prototypage de Scilab$_{//}$ 105
B.1.1 Type contexte 105
B.1.2 Type distribution 106
B.1.3 Type sous-matrice 106
B.1.4 Type matrice distribuée 106
B.2 Prototypage de Scilab$^{ooc}_{//}$ 107
B.2.1 Type distribution *out-of-core* 107
B.2.2 Type matrice distribuée *out-of-core* 108

Cette annexe présente une proposition de définition des types nécessaires à l'intégration de ScaLAPACK et de son prototype *out-of-core* dans Scilab.

B.1 Prototypage de Scilab$_{//}$

B.1.1 Type contexte

Élément	Signification
P	Nombre de ligne(s)
Q	Nombre de colonne(s)
ID	Tableau de taille $P \times Q$ des id des processeurs participant au contexte
CTXT	Tableau de taille $P \times Q$ des contextes BLACS de chaque processeur participant au contexte

Mise en œuvre : `mlist(["ctxt", "P","Q", "ID", "CTXT"], P, Q, ProcId, BlacsCTXT);`
Création : `ctxt r = scip_create_ctxt();`
Défaut : `mlist(["ctxt", "P","Q","ID", "CTXT"], -1, -1, [], []);`
Initialisation : `ctxt r = scip_init_ctxt(int p, int q, int[p,q] ids, ctxt[p,q] ctxt);`
Fonctions : `==, scip_is_ctxt`

B.1.2 Type distribution

Élément	Signification
MB	Taille de bloc (ligne)
NB	Taille de bloc (colonne)
Type	Type de distribution: CC cyclique-cyclique, BB bloc-bloc, CB cyclique-bloc, BC bloc-cyclique
RSRC	Indice ligne du processeur possédant le 1er élément de la matrice
CSRC	Indice colonne du processeur possédant le 1er élément de la matrice
Contexte	Objet contexte (défini précédemment)

Mise en œuvre : `mlist(["dist", "MB","NB","TYPE","RSRC","CSRC", "CTXT"], MB, NB, "TYPE", RSRC, CSRC, CTXT);`

Création : `dist r = scip_create_dist();`

Défaut : `mlist(["dist", "MB","NB","TYPE","RSRC","CSRC", "CTXT"], -1, -1, "UNKNOW", -1, -1, scip_create_ctxt());`

Initialisation : `dist r=scip_init_dist(["CC" | "BB" | "CB" | "BC"] dist, int rsrc, int csrc, ctxt ctxt, int mb, int nb);`

Fonctions : `==, ~=, scip_is_dist`

Divers : Pour une distribution BB, la taille des blocs MB et NB n'est pas utile et sera fixée à -1.

B.1.3 Type sous-matrice

Élément	Signification
M	Nombre de lignes de la sous-matrice
N	Nombre de colonnes de la sous-matrice
IA	Indice ligne du premier élément
JA	Indice colonne du premier élément

Mise en œuvre : `mlist(["sub", "M", "N", "IA", "JA"], M, N, IA, JA);`

Création : `sub r=scip_create_sub();`

Défaut : `mlist(["sub", "M", "N", "IA", "JA"], -1, -1, -1, -1);`

Initialisation : `sub r=scip_init_sub(int m, int n, int ia, int ja);`

Fonctions : `==, scip_is_sub(obj), scip_sub_isnull(obj);`

B.1.4 Type matrice distribuée

Élément	Signification
M	Nombre de lignes de la matrice
N	Nombre de colonnes de la matrice
DIST	Objet distribution
Data	Données
LDA	Dimension principale (locale)
SUB	Description d'un sous bloc de la matrice. Par défaut la sous-matrice est nulle, *i.e.*, égale à sa valeur par défaut.

Mise en œuvre : `mlist(["ds", "M","N","DIST","DATA", "LDA", "SUB"], M, N, DIST, DATA, LDA, SUB);`

Création : `ds r=scip_create_ds();`

Défaut : `mlist(["ds", "M","N","DIST","DATA", "LDA", "SUB"], -1, -1, scip_create_dist(), [], -1, scip_create_sub());`

Initialisation : `ds r=scip_init_ds(["MASTER"|"SLAVE"] conf, int m, int n, int lda, dist dist,[string|scalar] data);`

L'initialisation d'une variable distribuée doit se faire depuis le maitre (`MASTER` du paramètre `conf`). En mode `SLAVE`, chaque processeur crée la portion de la variable distribuée qui lui incombe en fonction du contexte. Pour calculer la taille de la zone mémoire nécessaire pour stocker localement la matrice distribuée, on utilise la fonction `sca_numroc`.

Fonctions : `==, () extraction, () insertion, scip_is_ds(obj),size, scip_ds_onmaster();`

Divers :

Si la sous-matrice n'est pas nulle *i.e.* que le champ `SUB` n'est pas égal à sa valeur par défaut, alors la taille d'une matrice distribuée est égale à la taille de la sous-matrice.

B.2 Prototypage de Scilab$^{ooc}_{//}$

B.2.1 Type distribution *out-of-core*

Élément	Signification
`MB`	Taille de bloc (ligne)
`NB`	Taille de bloc (colonne)
`Type`	Type de distribution: CC cyclique-cyclique, BB bloc- bloc, CB cyclique-bloc et BC bloc-cyclique
`MMB`	Taille de super-bloc (ligne)
`NNB`	Taille de super-bloc (colonne)
`RSRC`	Indice ligne du processeur possédant le 1er élément de la matrice
`CSRC`	Indice colonne du processeur possédant le 1er élément de la matrice
`Contexte`	Objet contexte tel que défini précédemment

Mise en œuvre : `mlist(["dist_ooc", "MB", "NB", "TYPE", "MMB", "NNB", "RSRC", "CSRC", "CTXT"], MB, NB, "TYPE", MMB, NNB, RSRC, CSRC, CTXT);`

Création : `dist r = scip_create_dist_ooc();`

Défaut : `mlist(["dist_ooc", "MB", "NB", "TYPE", "MMB", "NNB", "RSRC", "CSRC", "CTXT"], -1, -1, "UNKNOW", -1, -1, -1, -1, scip_create_ctxt();`

Initialisation : `dist_ooc r=scip_init_dist_ooc(["CC" | "BC" | "BB" | "BC" ] dist, int rsrc, int csrc, ctxt ctxt, int mmb, int nnb, int mb, int nb;`

Fonctions : `==, =, scip_is_dist_ooc`

B.2.2 Type matrice distribuée *out-of-core*

Élément	Signification
M	Nombre de lignes de la matrice
N	Nombre de colonnes de la matrice
DIST	Objet de distribution
Data	Données
LDA	Dimension principale (locale)
SUB	Description d'un sous-bloc de la matrice. Par défaut la sous-matrice est nulle, i.e. égale à sa valeur par défaut
FILENAME	Nom du fichier sur le disque
FILETYPE	Type du fichier
ASIZE	Taille mémoire allouée pour la matrice

La dimension principale ainsi que la taille mémoire utilisée sont locales au processeur qui possède les données.

Mise en œuvre : `mlist(["ds", "M", "N", "DIST", "DATA", "LDA", "SUB", "FILENAME", "FILETYPE", "ASIZE"], M, N, DIST, DATA, LDA, SUB, "FILENAME", "FILETYPE", ASIZE);`

Création : `ds_ooc r=scip_create_ds_ooc();`

Défaut : `mlist(["ds_ooc", "M", "N", "DIST", "DATA", "LDA", "SUB", "FILENAME", "FILETYPE", "ASIZE"], -1, -1, scip_create_dist_ooc(), [], -1, scip_create_sub(), "UNKNOW", "UNKNOW", -1);`

Initialisation : `ds_ooc r=scip_init_ds_ooc(["MASTER","SLAVE"] conf, int m, int n, int lda, dist_ooc dist_ooc, [string|scalar] data, string filename, char filetype, int asize);`

Fonctions : `==, () extraction, () insertion, scip_is_ds_ooc(obj), size, scip_ds_ooc_onmster()`

Table des figures

1.1 Représentation de la hiérarchie mémoire. 12
1.2 Modèle de gestion mémoire 13
1.3 Front d'accès mémoire : Découpage vertical 15
1.4 Front d'accès mémoire : Découpage horizontal 16
1.5 Pagination et flux d'exécution 17
1.6 Accès linéaire en mémoire. 17
1.7 Principe de l'*effet trou*. 18
1.8 Factorisation LU 18
1.9 Factorisation LU et effet trou 19
1.10 Application nécessitant l'accès à un fichier projeté dans une région mémoire. 21
1.11 Fonctions associées à la nouvelle zone mémoire 22
1.12 Factorisation LU avec les directives de pagination. 23
1.13 Nombre de pagination en fonction du nombre de threads : découpage vertical. 24
1.14 Découpage *data-driven* 25
1.15 Pseudocode de la convolution 27
1.16 Produit de convolution avec une décomposition du calcul. 28
1.17 Convolution version décomposée. 29

2.1 La hiérarchie de `ScaLAPACK`. 34
2.2 Distribution blocs cycliques 37
2.3 Analogie hiérarchie mémoire et décomposition `ScaLAPACK`. 38
2.4 Décomposition d'une matrice *out-of-core* sur 4 nœuds. 39

3.1 Un appel récursif à l'algorithme *right-looking* 43
3.2 Algorithme out-of-core left-looking avec super-blocs 44
3.3 Distribution des données pour la décomposition de la factorisation LU *out-of-core* . 46
3.4 Blocs impliqués dans les 3 fonctions principales de la factorisation LU 46
3.5 Comparaison expérimentation vs théorie : Factorisation LU (PC Celeron) 50
3.6 Comparaison expérimentation vs théorie : Factorisation LU (Alpha) 51
3.7 Performances théoriques de la factorisation LU 52
3.8 Rapports des surcoûts de communication et des E/S. 53

4.1 Substitutions triangulaires 58
4.2 Décomposition par blocs pour la résolution triangulaire. 59
4.3 Principe de la résolution triangulaire par bloc 60
4.4 Un niveau de la résolution triangulaire par bloc *(substitution avant)*. 61
4.5 Pseudo-code de l'inversion matricielle *out-of-core* 62
4.6 Résolution triangulaire sur un super-bloc 63

4.7 Comparaison expérimentation vs théorie : Inversion matricielle 66
4.8 Performances théoriques de l'inversion matricielle : Comparaison *in-core* vs *out-of-core* 67
4.9 Rapports des surcoûts pour les communications et les accès disques. 68
4.10 Performances théoriques de l'inversion matricielle : variation de la taille du super-bloc 69

5.1 Étapes de mise à jour de la factorisation LU sans recouvrement et avec recouvrement. 73
5.2 Occupation du CPU synchrone vs asynchrone 74

6.1 `Scilab` 82
6.2 Mécanisme d'interface 83
6.3 $\texttt{Scilab}_{//}$ 85
6.4 Architecture de l'interface $\texttt{Scilab}_{//}$ *out-of-core*. 86
6.5 Analyse de code *out-of-core* dans $\texttt{Scilab}_{//}$. 89
6.6 Promotion des diffrents types de `Scilab` 90

A.1 Huit processeurs virtualisés sur une grille 2x4 98

Bibliographie

[1] Vadim Abrossimov, Marc Rozier, et March Shapiro. « Generic virtual memory management for operating system kernels. ». Dans *Proc. of th 12th ACM Symposium on Operating System Principles*, décembre 1989.

[2] G. Alleon, S. Amram, N. Durante, P. Homsi, D. Pogarieloff, et C. Farhat. « Massively parallel processing boosts the solution of industrial electromagnetic problems : High Performance out-of-core solution of complex dense systems ». Dans *8th conference SIAM on Parallel Processing for Scientific Computing*. Aerospatiale Centre de Recherche, 1996.

[3] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, et D. Sorensen. « LAPACK: A Portable Linear Algebra Library for High-Performance Computers ». Dans IEEE, éditeur, *Proceedings, Supercomputing '90: November 12–16, 1990, New York Hilton at Rockefeller Center, New York, New York*, pages 2–11, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1990. IEEE Computer Society Press.

[4] ATM Forum. « Audiovisual Multimedia Services: Video on Demand Specification 1.1 », mars 1997.

[5] Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, et Vaidy Sunderam. « A Users' Guide to PVM Parallel Virtual Machine ». Rapport Technique CS-91-136, University of Tennessee, juillet 1991.

[6] L. A. Belady. « A study of replacement algorithms for virtual storage computers ». *IBM Systems Journal*, 5:78–101, 1966.

[7] Michel Blanc et Christian Clavier. « Les bronzés font du ski », 1979. Réalisation Patrice Leconte.

[8] Rajesh Bordawekar, Alok Choudhary, Ken Kennedy, Charles Koelbel, et Michael Paleczny. « A Model and Compilation Strategy for Out-of-core Data Parallel Programs ». Dans *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–10, Santa Barbara, CA, juillet 1995. ACM Press. Also available as the following technical reports: NPAC Technical Report SCCS-0696, CRPC Technical Report CRPC-TR94507-S, SIO Technical Report CACR SIO-104.

[9] Peter Brezany, Thomas A. Mueck, et Erich Schikuta. « Language, Compiler and Parallel Database Support for I/O Intensive Applications ». Dans *Proceedings of the International Conference on High Performance Computing and Networking*, volume 919 de *Lecture Notes in Computer Science*, pages 14–20, Milan, Italy, mai 1995. Springer-Verlag. also available as Technical Report of the Inst. f. Software Technology and Parallel Systems, University of Vienna, TR95-8, 1995.

[10] Philippe Candinot, Neilze Dorta, et Bertil Folliot. « MAÎS : un système de pagination en mémoire distante dans un environnement réseau à haut débit. ». Dans *RenPar'9*. EPF Lausanne, 20-23mai 1997.

[11] Rémy Card, Eric Dumas, et Franck Mével. *Programmation linux 2.0 - API système et fonctionnement du noyau*. Eyrolles, 1996.

[12] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, et Rajeev Thakur. « PVFS: A Parallel File System for Linux Clusters ». Dans *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, octobre 2000. USENIX Association.

[13] Eddy Caron, Olivier Cozette, Dominique Lazure, et Gil Utard. « Virtual Memory Management in Data Parallel Applications ». Dans *Proc. of HPCN'99 (High Performance Computing and Networking)*. Springer, avril 1999.

[14] Eddy Caron, Olivier Cozette, Dominique Lazure, et Gil Utard. « Mémoire virtuelle et parallélisme de données ». Dans *11èmes Rencontres francophones du parallélisme des architectures et des systèmes. Rennes. France*, pages 49–54, 8-11 Juin 1999.

[15] Eddy Caron, Frédéric Desprez, Eric Fleury, Dominique Lazure, Frédéric Suter, et Gil Utard. « ScalInt Interface Scilab/ScaLAPACK out-of-core Rev 0.1 ». Ouragan white paper, LaRIA / ReMaP / Résédas, Février 1999.

[16] Eddy Caron, Dominique Lazure, et Frédérique Suter. « Manipulation de données de grande taille dans Scilab// ». Dans *RenPar 2000. 12èmes Rencontres francophones du parallélisme. Besançon. France*, 19-22 Juin 2000.

[17] Eddy Caron, Dominique Lazure, et Gil Utard. « Génération de code pour le traitement de grandes données (Présentation sur affiche) ». Dans *10èmes Rencontres Francophones du Parallélisme. Strasbourg. France*, 9-12 Juin 1998.

[18] Eddy Caron, Dominique Lazure, et Gil Utard. « Inversion matricielle out-of-core par factorisation LU ». Dans *ParDi'99. Oujda. Maroc*, 27-30octobre 1999.

[19] Eddy Caron, Dominique Lazure, et Gil Utard. « Inversion of Huge Matrix on Cluster ». Dans *Cluster 2000. IEEE International Conference on Cluster Computing. Technische Universität Chemnitz, Saxony, Germany. (Poster)*, 28novembre-2décembre 2000.

[20] Eddy Caron, Dominique Lazure, et Gil Utard. « Modélisation et analyse des performances de la factorisation parallel LU out-of-core ». Dans *RenPar 2000. 12èmes Rencontres francophones du parallélisme. Besançon. France*, 19-22 Juin 2000.

[21] Eddy Caron, Dominique Lazure, et Gil Utard. « Performance Modeling and Analysis of Parallel Out-of-Core Matrix Factorization ». Dans *HiPC'2000. 7th International Conference on High Performance Computing. Bangalore, India*, December 17-20 2000.

[22] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, et R. C. Whaley. « LAPACK Working Note: ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performaces. ». Rapport Technique UT-CS-95, Department of Computer Science, University of Tennessee, 1995.

[23] Jaeyoung Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, et R. C. Whaley. « A proposal for a set of parallel basic linear algebra subprograms ». Dans J. J. Dongarra, Kaj Madsen, et Jerzy Wasniewski, éditeurs, *Applied parallel computing: computations in physics, chemistry, and engineering science: second international workshop, PARA '95, Lyngby, Denmark, August 21–24, 1995: proceedings*, volume 1041 de *Lecture Notes in Computer Science*, pages 107–114, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1996. Springer-Verlag.

[24] Alok Choudhary, Rajesh Bordawekar, Michael Harry, Rakesh Krishnaiyer, Ravi Ponnusamy, Tarvinder Singh, et Rajeev Thakur. « PASSION: Parallel And Scalable Software for Input-Output ». Rapport Technique SCCS-636, ECE Dept., NPAC and CASE Center, Syracuse University, septembre 1994.

[25] Almadena Chtchelkanova, John Gunnels, Greg Morrow, James Overfelt, et Robert Van de Geijn. « Parallel Implementation of BLAS: General Techniques for Level 3 BLAS ». Technical Report CS-TR-95-40, University of Texas, Austin, octobre 1995.

[26] J. Clinckemaillie, B. Elsner, G. Lonsdale, S. Meliciani, S. Vlachoutsis, F. de Bruyne, et M. Holzner. « Performance Issues of the Parallel PAM-CRASH Code ». *The International Journal of Supercomputer Applications and High Performance Computing*, 11(1):3–11, Spring 1997.

[27] Jean-François Collard et Gil Utard. « Automatic Data Layout and Code Restructuring for Out-of-Core Programs ». Dans *Proc. of the Workshop on Out-of-Core Computation and Adaptative Compilation (COCA'98)*, Cap-Hornu, Baie de Somme, France, septembre 1998. IEEE Yuforic and GDR ARP, LaRIA and PRiSM.

[28] Peter F. Corbett et Dror G. Feitelson. « Design and Implementation of the Vesta Parallel File System ». Dans *Proceedings of the Scalable High-Performance Computing Conference*, pages 63–70, 1994.

[29] Thomas H. Cormen et David M. Nicol. « Performing Out-of-Core FFTs on Parallel Disk Systems ». Technical report PCS-TR96-264, Dartmouth College Department of Computer Science, août 1996.

[30] Toni Cortes. « *Cooperative Caching and Prefetching in Parallel/Distributed File Systems* ». PhD thesis, UPC: Universitat Politècnica de Catalunya, Barcelona, Spain, 1997.

[31] Toni Cortes, Sergi Girona, et Jesús Labarta. « PACA: A Cooperative File System Cache for Parallel Machines ». Dans *Proceedings of the 2nd International Euro-Par Conference*, pages I:477–486, août 1996.

[32] Olivier Cozette. « Gestion de la pagination pour le calcul intensif ». Mémoire de D.E.A., LaRIA, Université de Picardie Jules Verne, juillet 1998.

[33] Ed F. d'Azevedo et Jack J. Dongarra. « The design and implementation of the parallel out-of-core Scalapack LU, QR and Cholesky factorization routines ». Rapport Technique UT-CS-97-347, Department of Computer Science, University of Tennessee, janvier 1997.

[34] S. DeFoster, D. Karst, M. Peterson, P. Sendelbach, et K. Kottschade. « Manufacturing Test of Fiber Channel Communications Cards and Optical Subassemblies ». Dans *ITC, International Test Conference*, pages 127–134, Washington - Brussels - Tokyo, octobre 1996. IEEE.

[35] J. W. Demmel, J. R. Gilbert, et X. S. Li. « *SuperLU "User's Guide"* », 1995.

[36] James W. Demmel, John R. Gilbert, et Xiaoye S. Li. « SuperLU ». http://www.nersc.gov/ xiaoye/SuperLU/, 1999.

[37] F. Desprez, S. Domas, et B. Tourancheau. « Optimization of the ScaLAPACK LU Factorization Routine Using Communication/Computation Overlap ». Dans *Europar'96 Parallel Processing*, volume 1124 de *Lecture Notes in Computer Science*, pages 3–10. Springer Verlag, août 1996.

[38] Frédéric Desprez et Eric Fleury. « ScalInt Interface Scilab/ScaLAPACK Rev 0.1 ». Ouragan white paper #2, ReMaP / Résédas, Décembre 1999.

[39] J. Dongarra, R. van de Geijn, et D. Walker. « A look at scalable dense linear algebra libraries ». Dans *Proceedings. Scalable High Performance Computing Conference SHPCC-92*, 1992. IEEE catalog number 92TH0432-5.

[40] J. J. Dongarra, C. B. Moler, J. R. Bunch, et G. W. Stewart. *LINPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1979.

[41] J. J. Dongarra, R. A. van de Geijn, et R. C. Whaley. « A Users' Guide to the BLACS ». Manuscript. Department of Computer Science, University of Tennessee, Knoxville, TN 37996., 1993.

[42] Jack Dongarra et E. F. D'Azevedo. « The Design and Implementation of the Parallel Out-of-core ScaLAPACK LU, QR, and Cholesky Factorization Routines ». Rapport Technique UT-CS-97-347, Department of Computer Science, University of Tennessee, janvier 1997. Tue, 1 Dec 98 21:54:53 GMT.

[43] Jack J. Dongarra, Sven Hammarling, et David W. Walker. « Key Concepts for Parallel Out-Of-Core LU Factorization ». Rapport Technique UT-CS-96-324, Department of Computer Science, University of Tennessee, avril 1996. Thu, 18 Feb 99 19:30:06 GMT.

[44] Mark Evans. « Ultra DMA – A Proposal for a New Protocol in ATA/ATAPI-4 ». Rapport Technique X3T13/D96153, Quantum Corporation, 1997.

[45] GDB/RBD. « MPI Primer/Developing with LAM ». Rapport Technique, Ohio Supercomputer Center, novembre 1996.

[46] A Geist, A Beguelin, J Dongarra, W Jiang, R Manchek, et V Sunderam. *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing.* Scientific and Engineering Computation. MIT Pres, 94.

[47] Sylvain Gérard. « *Algorithme de Cholesky out-of-core* ». Mémoire de D.E.A., Université de Picardie Jules Vernes Amiens, juillet 1999.

[48] William Gropp. « Tutorial on MPI: The Message-Passing Interface ». Rapport Technique, Mathematics and Computer Science Division, Argonne National Laboratory, 1995.

[49] William Gropp et Ewing Lusk. « User's guide for mpich, a Portable Implementation of MPI ». Rapport Technique ANL/MCS-TM-ANL-96/6, Mathematic and Computer Science Division, Argonne National Laboratory. University of Chicago, juin 1996.

[50] William Gropp, Ewing Lusk, et Anthony Skjellum. *USING MPI: Portable Parallel Programming with the Message-Passing Interface.* Scientific and Engineering Computation Series. The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts 02142, 1994.

[51] Scilab Group. « Introduction to Scilab: User's guide ». Rapport Technique, INRIA - Unité de Recherche de Rocquencourt - Projet Méta2, http://www-rocq.inria.fr/scilab, 1997.

[52] Divyesh Jadav, Chutimet Srinilta, Alok Choudhary, et P. Bruce Berra. « Techniques for Scheduling I/O in a High Performance Multimedia-On-Demand Server ». *Journal of Parallel and Distributed Computing*, pages 190–203, novembre 1996.

[53] Edgar T. Kalns et Yarsun Hsu. Video on Demand Using the Vesta Parallel File System. Dans Ravi Jain, John Werth, et James C. Browne, éditeurs, *Input/Output in Parallel and Distributed Computer Systems*, volume 362 de *The Kluwer International Series in Engineering and Computer Science*, Chapitre 8, pages 187–204. Kluwer Academic Publishers, 1996.

[54] Mahmut Kandemir, Rajesh Bordawekar, et Alok Choudhary. « I/O Optimizations for Compiling Out-of-Core programs on Distributed-Memory Machines ». Dans *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing.* Society for Industrial and Applied Mathematics, mars 1997. To appear. Extended Abstract.

[55] Nancy P. Kronenberg, Henry M. Levy, William D. Strecker, et Richard J. Merewood. « The VAXcluster Concept: An Overview of a Distributed System ». *Digital Technical Journal of Digital Equipment Corporation*, 1(5):7–21, septembre 1987.

[56] Keith Krueger, David Loftesness, Amin Vadhat, et Thomas Anderson. « Tools for the Development of Applications-Specific Virtual Memory Management ». Rapport Technique, University of California, Berkeley, avril 1993.

[57] Dominique LAZURE. « *Programmation géométrique à parallélisme de données : modèle, langage et compilation* ». Thèse, Université des Sciences et Technologies de Lille, Janvier 1995.

[58] M. MAKKI, J.J. LERALLUT, et I. IDY-PERETTI. « Vers une segmentation optimale d'images cardiaques acquises par RNM pour la reconstruction tridimensionnelle du coeur. ». Dans *Journée du Collège d'Etudes Doctorales en Sciences et Santé.* Université de Picardie Jules Verne, Novembre 1997.

[59] Robert J. MANCHEK. « Design and Implementation of PVM Version 3 ». Rapport Technique CS-94-232, University of Tennessee, mai 1994.

[60] Dan C. MARINESCU et Kuei Yu WANG. « An Analysis of the Paging Activity of Parallel Programs, Part I: Correlation of the Paging Activity of Individual Node Programs in the SPMD Execution Mode ». Rapport Technique CSD-TR-94-042, Purdue University, juin 1994.

[61] Dan C. MARINESCU et Kuei Yu WANG. « Characterization of the Paging Activity of NAS Benchmark Programs on the Intel Paragon ». Rapport Technique CSD-TR-95-015, Purdue University, mars 1995.

[62] Dan C. MARINESCU et Kuei Yu WANG. « Gang scheduling and Demand paging ». Dans *Proc. of the Int. Conf. on High Performance Computing*, pages 180 – 188, New Delhi, India, décembre 1995.

[63] MATHWORKS. « MATLAB ». http://www.mathworks.com.

[64] Peter T. MCLEAN et AL.. « Information technology - AT attachement with Packet Interface Extension (ATA/ATAPI-4) ». Working draft X3T13/1153D, Maxtor Corporation and Seagate Technology, février 1997.

[65] Todd C. MOWRY, Angela K. DEMKE, et Orran KRIEGER. « Automatic compiler-inserted I/O prefetching for out-of-core applications ». Dans *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pages 3–17. USENIX Association, octobre 1996.

[66] Steven A. MOYER et V. S. SUNDERAM. « PIOUS: A Scalable Parallel I/O System for Distributed Computing Environments ». Dans *Proceedings of the Scalable High-Performance Computing Conference*, pages 71–78, 1994.

[67] M. K. NG, Z. HUANG, et M. HEGLAND. « Data-Mining Massive Time Series Astronomical Data Sets — A Case Study ». *Lecture Notes in Computer Science*, 1394:401–402, 1998.

[68] Nils NIEUWEJAAR et David KOTZ. « The Galley Parallel File System ». Rapport Technique PCS-TR96-286, Dartmouth College, Computer Science, Hanover, NH, mai 1996.

[69] J.K. OUSTERHOUT. « Scheduling techniques for concurrent systems ». Dans *Proc. of the 3rd Int. Conf on Distributed Computing System*, pages 22–30, octobre 1982.

[70] Peter S. PACHECO. *Parallel Programming with MPI.* Morgan Kaufmann Publishers, Inc., 1997.

[71] Yoonho PARK, Ridgway SCOTT, et Stuart SECHREST. « Virtual Memory Versus File Interfaces for Large, Memory-intensive Scientific Applications ». Dans *Proceedings of Supercomputing '96.* ACM Press and IEEE Computer Society Press, novembre 1996. Also available as UH Department of Computer Science Research Report UH-CH-96-7.

[72] Wesley C. REILEY et Robert A. van de GEIJN. « POOCLAPACK : Parallel Out-of-Core Linear Algebra Package ». Rapport Technique Draft 27, Department of Computer Sciences, The University of Texas, Austin, octobre 1999.

[73] Howard ROBINSON et James DEMMEL. « Spectral Divide and Conquer (SDC) using the Matrix Sign Function ». http://http.cs.berkeley.edu/ hbr/sdc/, 1997.

[74] J.M. Del ROSARIO et A. CHOUDHARY. « High performance I/O for massively parallel computers: Problems and Prospects ». *IEEE Computer*, 27(3):59–68, 1994.

[75] David A Rusling. « *The Linux Kernel* ». Department of Computer Science and Engineering, University of Washington Seattle WA 98195, version 0.1-13(19) édition, juillet 1997.

[76] Robert D. Russell. « The Architecture of BPFS: a Basic Parallel File System Version 1.0 ». Technical Report RR-3460, Inria, Institut National de Recherche en Informatique et en Automatique, juillet 1998.

[77] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, et C. B. Moller. « Matrix Eigensystem Routines-EISPACK Guide ». *LectureNotes in ConrpurerScience*, 6, 1976.

[78] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, et Jack Dongarra. *MPI: the complete reference*. MIT Press, Cambridge, MA, USA, 1996.

[79] Hakan Taki et Gil Utard. « MPI-IO on a Parallel File System for Cluster of Workstations ». Dans *Proc. of IWCC'99 (1th International Workshop on Cluster Computing)*, Melbourne, Australia, décembre 1999. IEEE Press.

[80] Rajeev Thakur, Ewing Lusk, et William Gropp. « I/O Characterization of a Portable Astrophysics Application on the IBM SP and Intel Paragon ». Rapport Technique MCS-P534-0895, Argonne National Laboratory, août 1995. Revised October 1995.

[81] Sivan Toledo et Fred G. Gustavson. « The Design and Implementation of SOLAR, a Portable Library for Scalable Out-of-Core Linear Algebra Computations ». Dans *Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 28–40, Philadelphia, mai 1996. ACM Press.

[82] J. D. Ullman. « Some Advances in Data-Mining Techniques ». *Lecture Notes in Computer Science*, 1649:1–??, 1999.

[83] Brion Vibber. « *The Linux Ultra-DMA Mini-Howto* », v3.0 édition, novembre 1999.

[84] Jeffrey Scott Vitter et Elizabeth A. M. Shriver. « Algorithms for Parallel Memory I: Two-level Memories ». *Algorithmica*, 12(2/3):110–147, août and septembre 1994.

[85] Tom Wagner et Don Towsley. « Getting started with POSIX Threads ». Rapport Technique, Department of Computer Science, University of Massachusetts at Amherst, juillet 1995.

[86] R. Clint Whaley. « Installing and testing the BLACSv1.1 ». Rapport Technique, Department of Computer Science, University of Tennessee, mai 1997.

[87] R. Clint Whaley et Jack J. Dongarra. « Automatically Tuned Linear Algebra Software ». Rapport Technique UT-CS-97-366, Department of Computer Science, University of Tennessee, décembre 1997.

[88] P.E. Wolf et M.S. Lam. « A Data Locality Optimizing Algorithm ». Dans *Proc. ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, juin 1991.