

# Mémoire

présenté par

**Frédéric Desprez**

en vue de l'obtention du diplôme

**d'habilitation à diriger des recherches  
de l'Université Claude Bernard de LYON**  
(Numéro d'ordre 48-2001)

Spécialité : **Informatique**

## **CONTRIBUTION A L'ALGORITHMIQUE PARALLELE**

**Calcul numérique : des bibliothèques aux environnements de  
metacomputing**

Date de soutenance : 20 juillet 2001

Composition du jury :

Rapporteurs	Pierre	MANNEBACK
	Thierry	PRIOL
	Jean	ROMAN
Examineurs	Michel	COSNARD
	Jack	DONGARRA
	Thomas	LUDWIG
	Yves	ROBERT

**pour les travaux effectués au Computer Science Department de l'Université du Tennessee à Knoxville, au Laboratoire Bordelais de Recherche en Informatique et au Laboratoire de l'Informatique du Parallélisme de l'École normale supérieure de Lyon.**

Laboratoire de l'Informatique du Parallélisme  
UMR 5668 CNRS – ENS Lyon – INRIA Rhône-Alpes.



CONTRIBUTION A L'ALGORITHMIQUE PARALLELE  
Calcul numérique : des bibliothèques aux environnements de  
metacomputing

Frédéric Desprez

20 juillet 2001

Java and Jini sont des marques déposées de Sun Microsystems, Inc.

Myrinet est une marque déposée de Myricom Inc.

# Remerciements

Mes premiers remerciements iront tout naturellement à mes *papas* scientifiques qui, en me prenant en stage/thèse/postdoc ou en me recrutant, m'ont ouvert la voie vers des recherches passionnantes et ce beau métier qu'est celui de chercheur INRIA (mais si papa, c'est un métier!) : Bernard Tourancheau (maintenant « grand » professeur à Lyon I, euh non, CEO Myricom-France. Oups, directeur de recherche chez Sun Labs), Yves Robert (mon relooker et conseiller en management que j'ai le plaisir maintenant de diriger dans le projet ReMaP! En passant, tu penseras à me donner ta liste de publiés et ton rapport d'activité! ;-)), Michel Cosnard (toujours en mouvement, de ministères en URs INRIA...), Jack Dongarra (dont les GHz et le cv m'impressionneront toujours) et enfin Jean Roman (qui parle si bien et très très très très très longtemps de la scalabilité de la méthode Crout avec distribution 2D et agrégation partielle devant une Leffe à des heures où les braves gens et les CR INRIA raisonnables dorment!).

Je voudrais aussi rendre hommage à tous les(ex-)étudiants d'horizon divers et variés qui m'ont aidé à réaliser tous ces travaux et à qui, j'espère, j'ai un peu donné le goût de faire ce beau métier. En particulier, galanterie oblige, Frédérique Chaussumier (qui a illuminé le visage Clooney-sque de notre bien aimé directeur le jour de son arrivée au labo), Stéphane Domas (pourfandeur de manants et féministe endurci, pense à m'apprendre la nage médiévale), Cyril Randriamaro (désolé pour la semaine Knox-villienne où j'ai mis à mal ta récupération et ton sommeil!), Pierre Ramet (dont la capacité à écraser les champignons n'a d'égale que sa facilité dans la programmation des machines IBM de tous poils), Emmanuel Jeannot, Julien Zory (Good luck in Boston!), Fred Suter (avec qui j'aime à me rappeler le délicat plic-plic de la pluie Picarde), Martin *NWS-guru* Quinson, Georges-André Silber (qui a fait crever de jalousie la moitié des normaliens boutonneux), Fred *Corba rules!* Lombard, Eddy *Fast 25m* Caron, Laurent Bobelin, Jacques-Alexandre Gerber, Nathalie Viollet et Seb Cabaniols. Mais aussi des collègues de travail, Eric Fleury (qui subit sans crier mes délires sur la surcharge des opérateurs Scilab, mais bon, il va les présenter à Hawaii. Merci pour le stylo!), Stéphane Ubéda (« grand » professeur à l'INSA! Pense à me passer la doc PVM-Scilab avant 2013!), Claude Gomez (qui a accepté sans hurler de laisser paralléliser son beau Scilab), Jean-Michel Muller (mon nouveau partenaire de CP et futur bien-aimé directeur dont l'acidité des remarques matinales n'a d'égal que sa capacité à citer le regretté Cordic), Arnaud Tisserand (*sono de la casa! Quoi? 31 Frs pour une pizza?*), Jean-François Méhaut (les industries de la tong et du short de plage le remercient chaleureusement de son choix de carrière et de la hausse astronomique de leurs actions qui en suivi), Raymond Namyst (qui a eu le bon goût d'acheter une moto qui va moins vite que la mienne), Alain Darté et son vélo pliant (cher du kilo!), Luc Bougé (ex-fidèle et auguste adjoint!), Jérôme Gensel (ahhh, Kouzin! Encore merci pour avoir pu serrer la main d'Angry Anderson pendant que mes tympanes rendaient l'âme!), Serge Chaumette, Franck Rubi, Jean-Michel Lepine, Marie-Christine Counilh (qui j'espère s'est remise de mon passage éclair dans son bureau et de l'agitation qui s'en suivit), François Pellegrini, David Goudin, Jean-Christophe Mignot (pense à te faire écouter le dernier Slipknot, ça va te remonter!), Loic *Combien-Cà-Coûte* Prylli (dont la capacité à lire l'avenir dans les noyaux m'impressionne), Gil Utard, Jean-Marc Nicod, Thierry Priol, Thomas Brandes, Franck Cappello et en général tous les collègues rencontrés au gré des conférences ou écoles de part le monde (dur métier!).

Je tiens aussi à remercier l'ensemble du projet ReMaP que j'ai l'honneur à présent de « diriger ». Comme aime à me le rappeler Yves à chaque fois que je lui pose une question (certes probablement naïve), *tu as carte blanche!* ce qui signifie parfois (si je traduis bien) *démerde toi!*. Ca n'est pas toujours facile et j'espère que je ne vous décevrais pas! Un rapide coucou à Isabelle Guérin-Lassous qui illumine de sa délicate présence ce joyeux ensemble d'hommes des bois rassemblés dans le projet et qui a le bon goût de sourire à nos vanes de salle de garde (promis, on va te le trouver, ton poster des Chippendales).

Et d'autres, Xavier Vigouroux (range ton bureau, j'arrive!), Henri Casanova (entre autre pour Ministry et leur *Burning Inside* qui rythme mes entraînements matinaux. Prépare ta chambre d'amis à San Diego), Antoine Petitet (entre autre pour le prêt de la décapotable top-frime à Knoxville), Jean-Christophe Mison (qui assez gentil pour rester 10" derrière moi au 10 Kils), Michel Loi (avec qui je partageais ma solitude et un café au LIP le matin à l'heure où blanchit la campagne).

Et mes (parfois ex-)assistantes préférées (pour reprendre les termes chers à notre bien aimée INRIA) Marie (révise bien l'œuvre de Gaston Chaput avant de commander un déambulateur), Mme Sylvie, Anne-Pascale (entre autre pour son joli teint qui nous rappelle que Gerland n'est pas le top pour le bronzage), Corinne et Christelle.

Je n'oublierais pas bien sûr ceux qui ont eu l'obligeance de se pencher sur ce travail et de participer à sa présentation « finale », à savoir le jury. Pierre Manneback, Thierry Priol (qui a pu trouver le temps de relire tout ça entre 23 conférences et 54 réunions INRIA), Jean Roman (qui j'espère n'a pas usé trop son stylo rouge sur mes documents), Michel Cosnard, Jack 20 GHz Dongarra, Thomas Ludwig et Yves Robert, déjà remercié mais je le remercie doublement pour m'avoir permis d'obtenir ce poste en créant REMAP et en avoir ensuite fait cadeau (euh, cadeau, ça dépend des jours!).

Enfin, je dédie ce dernier diplôme à ma « petite » famille qui arrive parfois à comprendre que je peux me lever à 5h00 pour pédaler ou courir avant d'aller chercher je ne sais quoi à Gerland.

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Évolution des architectures . . . . .	3
1.2	Évolution des logiciels . . . . .	4
1.3	Plates-formes de metacomputing . . . . .	5
1.4	Bibliothèques numériques . . . . .	6
1.5	Contexte de recherche . . . . .	9
1.6	Organisation du document . . . . .	10
<b>2</b>	<b>LOCCS/OPIUM</b>	<b>13</b>
2.1	Introduction . . . . .	15
2.2	Recouvrements calculs/communications et tolérance à la latence . . . . .	15
2.3	Problématique et applications des macro-pipelines . . . . .	17
2.4	Gains prévus . . . . .	22
2.5	Enchainements de BLAS . . . . .	26
2.6	Calcul de la taille optimale de paquets . . . . .	28
2.7	Implantation des schémas macro-pipelines – Bibliothèque LOCCS . . . . .	32
2.8	Tiling . . . . .	34
2.9	Conclusion et perspectives . . . . .	35
<b>3</b>	<b>TransTool/ALASca</b>	<b>37</b>
3.1	Introduction . . . . .	39
3.2	Optimisation des redistributions . . . . .	40
3.3	Distribution automatique . . . . .	50
3.4	TransTool . . . . .	53
3.5	Conclusions et perspectives . . . . .	59
<b>4</b>	<b>Scilab//–DIET</b>	<b>63</b>
4.1	Introduction . . . . .	65
4.2	SCILAB// . . . . .	66
4.3	DIET, Distributed Interactive Engineering Toolbox . . . . .	74
4.4	Algorithmique parallèle mixte . . . . .	78
4.5	Conclusion et perspectives . . . . .	79
<b>5</b>	<b>Conclusions et perspectives</b>	<b>81</b>
5.1	Algorithmique hétérogène . . . . .	83
5.2	Serveurs de calculs . . . . .	84
5.3	Outils de haut niveau . . . . .	85
5.4	Conclusion personnelle . . . . .	85





# Chapitre **1**

## Introduction

Alors, elenaihoustabilisation ? (1732<sup>e</sup> fois)

Yves Robert, LIP ENS Lyon.



Ce document présente les activités de recherche que j'ai menées depuis l'obtention de mon doctorat en 1994. Ces activités se sont déroulées dans le **Computer Science Department de l'Université du Tennessee** où j'ai effectué mon postdoctorat, au **Laboratoire Bordelais de Recherche en Informatique** où j'ai été Maître de Conférence et enfin au **Laboratoire de l'Informatique du Parallélisme de l'École normale supérieure de Lyon** où j'ai été recruté comme Chargé de Recherche INRIA dans le projet ReMaP<sup>1</sup>.

Le parallélisme est maintenant solidement entré dans le monde industriel. Certes on n'a pas assisté à l'explosion promise au début des années 80 mais on retrouve des architectures parallèles dans tous les domaines, au niveau des processeurs jusqu'aux machines gigantesques du programme nucléaire américain ASCI [18], en passant par les réseaux de PCs. Le Téraflops a été obtenu en 96 [255] et on parle maintenant de Pétaflops [110].

Le domaine du calcul numérique, bien qu'il ne représente certainement qu'une infime portion du marché potentiel du parallélisme, est celui qui a reçu jusqu'à maintenant le plus d'attention. Les applications sont nombreuses et des livres entiers y sont consacrés [80, 113, 128, 133, 177, 235, 258]. La simulation, qu'elle soit utilisée dans le cadre de la construction mécanique, de la mécanique des fluides, de la simulation d'explosions nucléaires, de la météo, ou dans bien d'autres domaines, utilise le parallélisme de manière intensive depuis plusieurs années. Les besoins en puissance de calcul et en taille mémoire sont cependant toujours plus importants et la technologie n'avance pas assez vite pour accéder à cette demande.

Depuis mon DEA, je me suis intéressé à l'algorithmique numérique parallèle, au départ sur des machines aux performances et aux fonctionnalités sommaires (Telmat Tnode, Intel iPSC/860, Archipel Volvox) puis sur des machines plus évoluées (Intel Paragon, IBM SP2, Cray T3E, SGI Origin 2000). J'ai vécu la création des bibliothèques numériques parallèles et l'essor considérable d'un environnement de programmation par passage de messages comme PVM [131]. Récemment, nous sommes revenus à des machines relativement sommaires, construites à partir de cartes de PC connectées par des réseaux rapides comme Myrinet [204].

Durant ma thèse, j'ai travaillé sur l'optimisation de bibliothèques numériques de calcul sur machines à mémoire distribuée en étudiant l'optimisation de leurs communications (optimisation des schémas globaux, recouvrements calculs/communication, routines de communication pour architecture reconfigurable) et sur leur utilisation dans des codes numériques. Ces travaux avaient pour cible à l'époque des machines parfaitement homogènes, tant au niveau des processeurs, que des réseaux (machines à base de Transputers, Intel iPSC/860, Intel Paragon, réseaux de stations de travail).

## 1.1 Évolution des architectures

Les caractéristiques des processeurs évoluent chaque année et l'on atteint aujourd'hui pour des stations de travail de taille moyenne des performances dignes de supercalculateurs des années quatre-vingt. Les processeurs actuels possèdent des unités fonctionnelles multiples et donc un parallélisme interne au processeur, des pipelines superscalaires, des hiérarchies mémoire importantes et des tailles de caches internes dignes des mémoires des stations de travail du passé.

En ce qui concerne les architectures parallèles, sans faire un historique complet des évolutions, on peut dégager quelques grandes classes d'architectures. Après les supercalculateurs de style CRAY (mono-processeurs très puissants) et quelques architectures à mémoire partagée de taille modeste, on a vu le développement de nombreuses machines à mémoire distribuée dont on a trop vite dit qu'elles enterreront les machines à mémoire partagée. Il y a eu l'époque du massivement parallèle à la portée de tous avec les machines SIMD dont on a rapidement vu les limitations à des classes d'applications restreintes.

---

<sup>1</sup>Projet CNRS-ENS Lyon-INRIA.

On constate actuellement une évolution franche vers les grappes de machines SMP. Un bon exemple est la nouvelle machine SP3 d'IBM qui est constituée de nœuds puissants à 8 processeurs partageant une même mémoire reliée par un réseau rapide. Par ailleurs, on a également vu la disparition totale des processeurs spécialisés développés par les constructeurs au profit de processeurs « on the shelf » utilisés plutôt dans les PCs puissants ou les stations de travail que dans les machines parallèles. Les dernières versions de ces cartes peuvent abriter jusqu'à 8 processeurs pour donner des puissances jusqu'au Gigaflops.

Du côté des réseaux également, on a vu la prolifération de réseaux à bas prix et aux caractéristiques tout à fait impressionnantes comme les réseaux Myrinet, SCI, Fast Ethernet ou GigaEthernet. L'agglomération de ces processeurs et ces réseaux bon marché nous a conduit à construire des grappes de PCs comme on joue au Lego. Ces prix compétitifs ont permis la mise en place de plusieurs machines de petites tailles (jusqu'à 200 processeurs toutefois pour la grappe de l'UR Rhône-Alpes !) et ainsi d'effectuer les tests de diverses configurations matérielles et logicielles.

## 1.2 Évolution des logiciels

Les logiciels, comme dans tous les domaines de l'informatique mais encore plus sur les machines parallèles, ont eu une évolution plus lente comparée à celle du matériel.

La vitesse des processeurs progresse mais la complexité et la puissance des compilateurs également. Les recherches sur la compilation pour les architectures superscalaires ont donné de très bons résultats et les compilateurs actuels peuvent obtenir des performances proches des performances en crête sur des benchmarks qui donnaient auparavant de bien piètres résultats. Les compilateurs d'aujourd'hui restructurent le code de manière importante et savent générer des opérations duales pour tirer parti au mieux des processeurs à plusieurs unités d'exécution [20]. Si l'on regarde plus spécifiquement le domaine du calcul numérique parallèle, on peut dégager trois tendances.

Tout d'abord, et comme nous le verrons dans les sections suivantes, les bibliothèques numériques rassemblant des noyaux de base ont eu un grand impact sur le développement des applications [129]. Ces bibliothèques permettent de remplacer certains noyaux numériques par des versions optimisées et de pouvoir tester simplement de nouveaux algorithmes (comme par exemple dans le cas des solveurs itératifs). Je reviendrai plus précisément sur les bibliothèques numériques dans la section 1.4.

La complexité des architectures parallèles a vite ralenti le développement de compilateurs parallélisateurs efficaces. Il y a eu la définition d'un Fortran à haute performance<sup>2</sup> [175] dont on avait dit qu'il résoudreait tous les problèmes de parallélisation d'applications numériques grâce à la collaboration efficace du programmeur de l'application (grâce aux directives de placement) et aux compilateurs (grâce aux progrès de la parallélisation automatique). Malheureusement pour HPF, la plupart des « vraies » applications travaillent sur des structures de données très irrégulières et la compilation d'algorithmes pour ces structures, même avec l'aide de l'utilisateur, reste un problème d'une complexité extrême [76, 152]. Un autre effort de standardisation d'un Fortran à haute performance, mais cette fois pour les machines à mémoire partagée est apparu : OpenMP [215]. Il est certes moins difficile à compiler mais il restreint (pour l'instant) les libertés laissées à l'utilisateur d'aider le compilateur.

Le passage de messages est resté le paradigme de programmation privilégié pour le développement d'applications sur machines parallèles à mémoire distribuée mais aussi sur machines à mémoire partagée. La complexité de sa programmation et de sa recherche d'erreur le rend souvent relativement inaccessible au bétien. Il reste toutefois le moyen le plus « facile » d'obtenir les meilleures performances et ceci pour un grand nombre d'applications. Le

---

<sup>2</sup>High Performance Fortran.

développement récent d'une interface standard comme MPI<sup>3</sup> [124, 246] l'a propulsé au premier rang.

Il y a bien sûr d'autres moyens et outils et langages de parallélisation d'applications comme les threads, des langages data-parallèles basés sur C, Fortran, des outils graphiques (Hence, Grade), des mémoires partagées virtuelles, la programmation par composants logiciels et même Java.

Nous reviendrons sur divers environnements dans la suite du document.

## 1.3 Plates-formes de metacomputing

Après avoir tenté de mettre au point des machines de tailles toujours plus importantes (jusqu'à 9000 Pentiums pour la machine du programme américain ASCI), on agrège maintenant la puissance de nombreuses machines de tailles moins importantes pour former des grappes. Une nouvelle tendance consiste même à tenter d'utiliser des machines disponibles dans le monde entier, et ceci avec une relative transparence. Les plates-formes ainsi mises en place ont un potentiel immense car la plupart des machines utilisées interactivement sont le plus souvent sous-employées, et parce que l'on peut déporter les calculs vers la machine la plus adaptée pour les résoudre. Malheureusement, de telles solutions sont extrêmement difficiles à mettre en place de manière efficace. Le principe général est donc de tenter d'agréger un certain nombre de ressources disponibles comme un *méta-ordinateur*. À mi-chemin entre le concept architectural (on agrège des machines accessibles sur la toile) et le logiciel (on utilise la puissance des machines sans les connaître comme on utiliserait l'électricité), le metacomputing ou *grid computing* constituera certainement l'une des évolutions majeures de l'informatique de demain.

Au niveau de l'architecture matérielle, le metacomputing peut prendre plusieurs formes. Une caractéristique commune est cependant son importante hétérogénéité et sa hiérarchisation. Dans les rêves les plus fous des chercheurs (voire plutôt des commerciaux), le metacomputing est l'agrégation de milliers de machines accessibles sur l'Internet pour résoudre des applications à grande échelle. Il s'agit déjà d'une réalité puisque des projets comme SETI@home permettent d'effectuer des calculs sur des PCs de par le monde. Il faut bien avouer que vue la bande passante actuelle du réseau, il s'agit plutôt d'applications « *embarrassingly parallel* » qui ne communiquent qu'épisodiquement avec une machine maître. Un autre exemple est certainement la grille GUSTO<sup>4</sup> qui interconnecte 17 sites et 330 supercalculateurs (plus de 3600 processeurs) pour fournir une puissance globale cumulée de plus de 2 TeraFlops par seconde en crête ! Toujours à grande échelle mais avec des performances en communications plus importantes, on peut citer le projet VTHD<sup>5</sup> du RNRT qui relie les centres de recherche de France Telecom, les unités de recherche INRIA et d'autres laboratoires par un réseau à 2.5 Gb/s. Sur une telle plate-forme, le réseau inter-centres est même plus rapide que les réseaux internes des grappes qu'il relie ! Plusieurs projets de ce type existent à travers le monde qui connectent généralement un petit ensemble de machines par un réseau rapide. Enfin, au niveau d'un laboratoire, on peut monter une plate-forme hétérogène en connectant plusieurs grappes entre elles par des réseaux plus ou moins rapides.

Le problème commun à toutes ces architectures n'est en fait pas le matériel (ces machines sont rarement isolées du monde) mais plutôt le logiciel (ce qui va naturellement du système à l'algorithme). En effet, il y a peu de chance que ces machines soient totalement homogènes. Au mieux, on agrège des machines SMP de même type par un réseau rapide et au pire, on prend la puissance de calcul disponible sur la toile et, des réseaux aux processeurs en passant par les systèmes d'exploitation, l'ensemble de l'architecture est hétérogène. Depuis cinq ans, nous constatons une explosion dans le monde entier des recherches autour du *grid computing*. La plupart des laboratoires qui travaillaient auparavant sur le parallélisme travaillent maintenant sur le metacomputing. Plus étonnant encore, on constate que de nombreux projets dans le monde

---

<sup>3</sup>Message Passing Interface.

<sup>4</sup>Globus Ubiquitous Supercomputing Testbed.

<sup>5</sup>Réseau à Vraiment Très Haut Débit.

sont tirés par des chercheurs venant du monde des applications (physique, chimie, imagerie, etc.).

Les machines, les réseaux et même les systèmes d'exploitation étant hétérogènes, une partie de cette hétérogénéité (comme le transfert des données) devra être cachée par l'environnement de ces machines. Si l'on sait aujourd'hui parfaitement communiquer des données entre processeurs hétérogènes, par exemple en utilisant des formats standard de représentation des données (XDR), il n'en est pas de même au niveau des protocoles de communication. Jusqu'à aujourd'hui, l'approche retenue était de s'appuyer sur un unique protocole de communication (TCP/IP) pour interconnecter les ordinateurs de la planète. Cependant, les recherches montrent les limites de ce protocole pour le calcul à haute performance. Dans le contexte du metacomputing, pour éviter de s'appuyer sur ce protocole unique, l'un des challenges sera de réussir à faire cohabiter dans les environnements d'exécution (MPI, CORBA) les protocoles spécifiques et performants de communication (e.g. BIP, VIA) que l'on trouve sur des architectures parallèles ou des grappes, avec le protocole de communication à grande échelle (TCP/IP). On obtiendra des couches de communications multi-protocoles [19]. Les aspects d'administration des machines (gestion de comptes, quotas, accounting, etc.) et les aspects sécurité doivent être étudiés de plus près.

Il y a deux challenges principaux pour le développement des plates-formes de metacomputing et pour une utilisation plus sérieuse que la recherche de vie extra-terrestre dans l'univers : le développement d'environnements rendant « transparente » l'utilisation de la grille et des résultats sur l'algorithmique des applications utilisant de telles plates-formes. Dans « environnement », on peut placer bien sûr le système d'exploitation mais aussi les langages, les bibliothèques et le middleware [24, 119, 125]. Actuellement, la plupart des environnements existants reprennent les approches choisies pour les machines parallèles « classiques » en tentant de les adapter à la grille. Les résultats sont bien entendu inégaux. Les recherches autour de l'ordonnancement des tâches en milieux hétérogènes sont également de première importance.

Je reviendrai par la suite sur le metacomputing en parlant des approches *Network Enabled Servers* pour le calcul distribué dans le chapitre 4.

## 1.4 Bibliothèques numériques

Le but des bibliothèques numériques est triple. Premièrement, elles servent à offrir à l'utilisateur une interface simple pour résoudre des problèmes communs à diverses applications tout en laissant à leur développeur une certaine liberté quant au choix de l'algorithme permettant de les résoudre. Ensuite, elles permettent de cacher à l'utilisateur les caractéristiques de sa machine cible et de laisser au développeur de bibliothèques la lourde tâche d'utiliser au mieux les interactions entre le matériel (niveaux de caches, pipelines superscalaires, unités parallèles, etc.) et le logiciel (utilisation de bibliothèques de plus bas niveau, options de compilation, choix du langage d'implémentation, etc.). Enfin, et comme nous le verrons par la suite, ces bibliothèques peuvent être utilisées pour développer des environnements de plus haut niveau (autres bibliothèques, logiciels mathématiques, support d'exécution pour des langages, serveurs de calculs, etc.).

Cependant, la tâche des programmeurs de bibliothèques mathématiques est loin d'être aisée. Différents problèmes doivent être examinés (et résolus !) afin d'obtenir un logiciel utilisable par la communauté scientifique :

- choix de l'interface de programmation (API) : plus elle est compliquée, moins elle sera utilisée par le plus grand nombre. Les approches orientées objet ont clairement leur rôle à jouer dans ce domaine [108],
- choix des algorithmes pour la résolution des problèmes : des problèmes de stabilité numérique [142] peuvent intervenir et des tests intensifs doivent assurer que la bibliothèque donnera des résultats justes dans tous les cas (y compris sur toutes les machines). Par ailleurs, les meilleurs algorithmes séquentiels ne donnent pas forcément les meilleurs algorithmes parallèles (et vice-versa) [221],

- problèmes de dissémination et de maintenance du logiciel. La dissémination a été résolue grâce à Internet avec des bases de données comme netlib [50] ou GAMS [40]. Le problème de la maintenance reste d'actualité puisque de nombreuses bibliothèques de qualité ont été développées par des étudiants en thèse et que leur pérennité est plutôt incertaine.

Un excellent historique des bibliothèques numériques des années 70 à aujourd'hui est donné dans l'article de R. Boisvert [39].

### 1.4.1 Optimisation des bibliothèques séquentielles

Dans de nombreux algorithmes numériques pour lesquels le nombre d'opérations flottantes est proportionnel au cube de la taille des données en entrée, l'obtention de bonnes performances en parallèle passe déjà par l'optimisation des noyaux numériques séquentiels.

L'utilisation de bibliothèques numériques est maintenant rentrée dans les mœurs et personne de sensé n'écrit une multiplication de matrices avec trois boucles<sup>6</sup>. Les noyaux numériques sont variés, qu'ils traitent de matrices denses comme les BLAS [111, 112, 182] ou LAPACK [114], ou de matrices creuses comme SPARSEKIT [238], les NIST Sparse BLAS [233] et d'autres [116, 129].

Les optimisations de ces bibliothèques sont essentiellement tournées autour de l'utilisation d'algorithmes par blocs permettant une meilleure réutilisation des hiérarchies mémoire. On constate que les performances des routines matrice-matrice (BLAS de niveau 3) sont bien meilleures que les performances des niveaux inférieurs (opérations scalaire-vecteur ou vecteur-matrice) [107]. Une approche intéressante consiste à avoir des générateurs de bibliothèques. Ces outils permettent d'adapter un code source (déjà optimisé) à la fois à l'architecture d'une machine cible ainsi qu'à son compilateur. ATLAS [267] de l'Université du Tennessee à Knoxville et PhiPac [34] de l'Université de Berkeley sont de tels outils. L'idée est relativement simple mais très efficace : plutôt que de développer des noyaux numériques en assembleur afin de tirer parti au mieux d'une architecture ou de laisser faire le compilateur en espérant qu'il saura trouver les optimisations qui permettront d'atteindre les meilleures performances, on développe un outil qui analyse itérativement l'architecture cible (niveaux et tailles des caches, opérations duales) et son compilateur (effet des options de compilation) et l'on génère le source de la bibliothèque. Il suffit ensuite de compiler la bibliothèque résultante pour obtenir une librairie à lier avec le reste de l'application. Ces outils permettent d'obtenir des performances proches des performances de crêtes et surtout équivalentes à celles des bibliothèques classiques en assembleur. Si cela est relativement aisé à faire pour des noyaux numériques denses, cela s'avère excessivement compliqué pour des noyaux numériques creux. Certains travaux existent cependant pour la génération de codes creux efficaces [157, 170, 171] mais à ma connaissance seuls les travaux autour du service « Sparse Subroutine on Demand » [33] de l'Université de Leiden au Pays-Bas a donné lieu au développement d'un outil similaire à ATLAS ou PhiPAC. Une autre approche existe autour des templates pour la génération de codes denses ou creux à partir de C++ [242].

Si l'on prend le produit de matrice, qui est utilisé dans de nombreux noyaux numériques [162], des algorithmes existent comme ceux de Strassen [146, 250] ou de Winograd [123] qui réduisent le nombre de multiplications et ainsi le temps total d'exécution mais leur utilisation comme noyaux numériques s'avère assez difficile parce que des problèmes de stabilité numérique peuvent apparaître [81].

En ce qui concerne l'utilisation du langage Java pour le calcul à haute performance, il apparaît que les seules solutions actuelles consistent surtout à interfacier des bibliothèques classiques [38, 41] ou à utiliser des optimisations classiques des compilateurs en donnant des régions où l'interpréteur n'effectuera pas de vérification de dépassement mémoire [199].

---

<sup>6</sup>Sauf à titre d'exercice !

### 1.4.2 Bibliothèques numériques parallèles

La difficulté de programmation des machines parallèles s'avérant souvent quasiment insurmontable pour de nombreux programmeurs d'applications, des bibliothèques numériques ont également été conçues pour ces plates-formes et notamment pour les machines parallèles à mémoire distribuée. La plus célèbre est certainement ScaLAPACK [37]. Celle-ci est la version parallèle d'un sous-ensemble de LAPACK. Elle est bâtie sur les BLAS, sur les BLACS<sup>7</sup> [109] et sur la version parallèle des BLAS, les PBLAS [72]. La figure 1.1 donne l'architecture logicielle de cette bibliothèque.

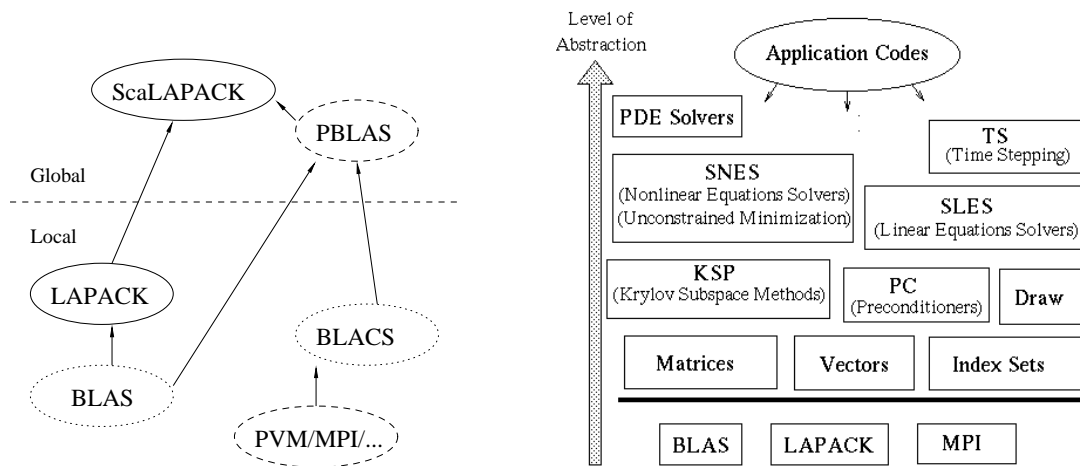


FIG. 1.1 – Architecture logicielle de ScaLAPACK.

FIG. 1.2 – Architecture logicielle de PETSc.

Le concept novateur de ScaLAPACK consiste à conserver une vision globale de la matrice au plus haut niveau de l'algorithme et ainsi pouvoir transformer facilement les algorithmes séquentiels de LAPACK en algorithmes de ScaLAPACK. En effet, la transformation se résume à passer en paramètres aux routines BLAS (devenues des PBLAS) les descripteurs des matrices. Ces descripteurs spécifient la distribution des matrices sur une grille virtuelle de processeurs. Tous les appels à des routines de communication sont cachés dans les PBLAS. On constate également que, comme pour les bibliothèques séquentielles, l'utilisation d'algorithmes par blocs, liée à une distribution des données de manière cyclique par blocs, permet de réduire les communications et de mieux équilibrer la charge entre les processeurs de la grille. Je reviendrai sur ce type de distributions et sur les redistributions nécessaires à l'équilibrage des charges dans une application utilisant ScaLAPACK dans le chapitre 3.

Pour le calcul creux, les difficultés sont encore plus importantes puisque la distribution des données est dépendante de la répartition des éléments non-nuls dans les matrices et il est souvent bien difficile de trouver une distribution statique efficace [6, 138, 144]. Il existe cependant quelques bibliothèques parallèles pour le creux comme PASTIX [144], PETSc [27], PS-PASES [138], MUMPS [6], etc. Les problèmes algorithmiques (et logiciels) sont nombreux et les techniques employées sont souvent issues de l'algorithmique pour les graphes [132]. De plus, si l'on prend les méthodes itératives, il y a pratiquement autant d'algorithmes que de types de matrices [239] et bien des problèmes restent ouverts.

<sup>7</sup>Basic Linear Algebra Communication Subroutines.



### 1.4.3 Algorithmique numérique séquentielle et parallèle

Le développement de bibliothèques de haut niveau a donné lieu à de nombreuses recherches autour de l'algorithmique elle-même. En effet, si une solution mathématique existe pour un problème, elle n'est pas forcément implantable sur une machine réelle. On l'a constaté plus d'une fois et notamment avec le développement de méthodes réduisant la complexité des opérations mathématiques comme la factorisation ou le produit de matrices. Le développement des architectures et leur évolution ont eu un impact important sur l'algorithmique elle-même [142]. Même si l'on travaille sur une abstraction du matériel en supposant l'existence d'une machine virtuelle, on se doit de tenir compte de l'architecture cible pour espérer obtenir des performances. Un bon exemple est le développement des algorithmes par blocs dans LAPACK [9].

Les algorithmes parallèles n'échappent pas à ce principe. La distribution des données doit être optimisée pour équilibrer au mieux la charge et réduire leurs transferts. Les données utilisées dans les applications numériques étant essentiellement des matrices et les tâches effectuées étant généralement similaires, le paradigme de programmation choisi est souvent le parallélisme de données. De nombreuses recherches ont également pris comme base le parallélisme de tâches en cherchant des heuristiques pour leur placement sur les processeurs. On peut aussi combiner parallélisme de tâches et parallélisme de données pour obtenir un parallélisme mixte (chapitre 4, section 4.4 et chapitre H du document annexe).

## 1.5 Contexte de recherche

Mes travaux ont été bien sûr très influencés par les chercheurs que j'ai cotoyés depuis ma thèse et par l'évolution de notre domaine. Dans cette section, je positionne mes travaux dans cet environnement de travail.

### 1.5.1 Contexte scientifique et technologique

Mon domaine de recherche s'est essentiellement concentré autour de la parallélisation d'applications numériques sur des machines parallèles diverses et variées et sur l'étude des recouvrements calculs/communications dans les algorithmes pipelines.

La parallélisation automatique a montré ses limitations comme l'explique notamment A. Darté dans son document d'habilitation [79] et les bibliothèques de calcul représentent une solution intéressante pour atteindre de bonnes performances avec un coût de développement relativement réduit. Par contre, leur utilisation est loin d'être transparente pour le programmeur d'application et il est nécessaire de développer des outils qui lui simplifient la tâche. Ces outils peuvent prendre diverses formes, de la génération automatique de directives pour des langages data-parallèles aux outils mathématiques comme Matlab ou Scilab.

Mon deuxième sujet de recherche consiste à masquer la latence des communications en les recouvrant par des calculs, soit en utilisant des communications asynchrones, soit en modifiant les algorithmes pour obtenir des algorithmes pipelines. Mes recherches dans ce domaine ont suivi toute la chaîne de développement, des aspects architecturaux et de bas niveau dans les architectures elles-mêmes et dans les bibliothèques de communication jusqu'aux langages et algorithmes.

### 1.5.2 Contexte environnemental

Les années 90 ont vu l'explosion de la recherche autour du parallélisme. C'est dans ce contexte et en participant à cette excitante aventure que j'ai développé mes propres recherches.

Premièrement, ma mobilité m'a permis de travailler dans divers laboratoires et j'ai eu la chance de connaître de l'intérieur certaines des équipes françaises et étrangères les plus pointues dans

le domaine et de participer à l'émergence de nouveaux thèmes de recherche dans ces équipes. Durant ma thèse puis en temps qu'ATER, j'ai travaillé dans l'équipe ANOD<sup>8</sup> du LIP dirigée à l'époque par B. Tourancheau. Le LIP à l'époque regroupait un ensemble très important de thématiques autour du parallélisme (parallélisation automatique, graphes et communications, PRAM, imagerie, réseaux de neurones, algorithmique numérique, monitoring, complexité, etc.) et ce vivier m'a permis de discuter et de collaborer avec plusieurs chercheurs d'autres thèmes (P. Fraigniaud sur les communications dans les hypercubes [94], A. Ferreira sur les algorithmes de communication pour réseaux optiques [91] et C. Biondello sur les algorithmes numériques pour les réseaux reconfigurables [42]). Nos collaborations externes étaient nombreuses et parfois avec des chercheurs d'autres disciplines comme par exemple M. Garbey avec lequel j'ai parallélisé une application de front de flammes [96].

Ensuite, durant mon postdoctorat à l'Université du Tennessee à Knoxville (et grâce aux projets PICS, CNRS-NSF et INRIA-NSF de B. Tourancheau et J. Dongarra), j'ai pu participer au développement du projet ScaLAPACK et voir de l'intérieur les côtés positifs et parfois négatifs du travail en collaboration sur un si gros projet.

À la suite de mon postdoctorat, j'ai été recruté au LaBRI dans l'équipe de J. Roman<sup>9</sup>. J'ai eu la liberté de continuer mes travaux sur les recouvrements calculs/communications [99] et j'ai pu démarrer le projet TRANSTOOL autour de la parallélisation de programmes Fortran (décrit dans le chapitre 3).

Enfin, j'ai été recruté comme chargé de recherche à l'INRIA en 95 et j'ai participé au démarrage du projet ReMaP dirigé à l'époque par Y. Robert. J'ai continué mes recherches sur les recouvrements calculs-communications et poursuivi le projet TRANSTOOL<sup>10</sup> en collaboration avec A. Darté. En 1999, j'ai démarré le projet SCILAB<sub>//</sub><sup>11</sup> (décrit dans le chapitre 4).

Durant toutes ces années, j'ai eu aussi la chance de participer au PRC-GDR C3 (groupes de travail CAPA et Rumeur), au PRC-GDR PRS<sup>12</sup> (projet inter-PRC Stratagème) et enfin au GDR ARP<sup>13</sup> (groupes de travail Grappes et iHPerf) ainsi qu'à de nombreuses écoles d'été et d'hiver. Je dois dire que ces regroupements de chercheurs français sur des domaines évoluant aussi rapidement ont beaucoup fait pour le dynamisme de ces équipes de recherche et pour le développement de nouveaux projets et collaborations.

## 1.6 Organisation du document

Ce document a été découpé en utilisant les divers projets de recherche auxquels je me suis intéressé et contient donc trois chapitres principaux.

Dans le premier chapitre, je présente l'utilisation des recouvrements calculs/communication et des pipelines de calcul dans les algorithmes numériques et les applications parallèles en général. Après avoir proposé dans ma thèse une bibliothèque permettant d'effectuer simplement de tels recouvrements (les LOCCS), j'ai poursuivi ces travaux dans diverses directions (évaluation des capacités architecturales et logicielles à recouvrir les communications, analyse des gains, utilisation dans un compilateur paralléliseur, calcul du grain optimal, utilisation dans diverses applications). Ce domaine de recherche a ensuite été récurrent. Au départ centré vers les applications régulières, je me suis peu à peu tourné vers des applications irrégulières et la compilation de programmes data-parallèles.

Dans le second chapitre, je présente mes travaux autour du développement d'un outil de transformation de codes Fortran 77 vers High Performance Fortran, TRANSTOOL. Ce projet, initié durant mon année au LaBRI, avait pour but de construire une plate-forme d'expérimentation d'ou-

<sup>8</sup>Algorithmique Numérique et Outils de développement

<sup>9</sup>ALiENor.

<sup>10</sup>Financé par le projet EuroTOPS.

<sup>11</sup>Financé par l'ARC INRIA OURAGAN.

<sup>12</sup>Parallélisme, Réseaux et Systèmes.

<sup>13</sup>Architecture, Réseaux, Systèmes et Parallélisme.

tils de transformation source-à-source de programmes Fortran. En effet nous avons développé, au LaBRI comme au LIP, des prototypes permettant la transformation de codes. Par contre, la validation et la présentation de tels outils sans une infrastructure générale n'était pas une tâche aisée.

Enfin, dans le troisième chapitre et avant quelques conclusions et perspectives, je présente mes travaux autour du projet OURAGAN dont le but est d'apporter les hautes performances au logiciel SCILAB et à développer un environnement de conception d'applications de type ASP (*Application Service Provider*). Ces derniers travaux représentent également mon projet de recherche pour les années à venir.



# Chapitre 2

## LOCCS/OPIUM ou Recouvrements et pipelines à tous les étages

Dans ce chapitre, je présente mes travaux dans le domaine de l'optimisation d'algorithmes numériques grâce aux recouvrements calculs/communications et aux pipelines de calcul.

Ces travaux s'échelonnent depuis ma thèse jusqu'à aujourd'hui.

*Travaux effectués en collaboration avec F. Chaussumier, S. Domas, L. Prylli, P. Ramet et J. Roman.*



## 2.1 Introduction

L'utilisation de machines parallèles à mémoire distribuée induit un surcoût dû aux communications, sauf si le programme est *embarrassingly parallel*. Ce coût doit être réduit à son minimum si l'on veut garantir l'obtention de bonnes performances et surtout si l'on souhaite conserver une bonne extensibilité pour une application lorsque le nombre de processeurs augmente. Il va de soi que la première étape pour la réduction des communications lors de la parallélisation d'une application sur une machine à mémoire distribuée est le choix de bonnes distributions pour les données (cf. chapitre 3). Ces distributions sont « bonnes » si elles réduisent les communications tout en conservant un bon parallélisme. Il faut ensuite être en mesure de recouvrir un maximum de communications par des calculs.

J'ai commencé à m'intéresser aux recouvrements calculs–communications et recouvrement pipelines durant ma thèse. Dans un premier temps, j'ai étudié comment recouvrir les communications dans des machines à base de Transputers. Ces machines possédaient 4 liens alimentés par 4 DMA. L'extrême simplicité du système alliée à l'efficacité de l'architecture permettait d'obtenir un recouvrement total [95, 97]. Cependant, je me suis vite aperçu que les recouvrements simples n'étaient pas toujours possibles du fait des dépendances à l'intérieur du code. Je me suis donc attaqué à l'implémentation de schémas macro-pipelines dans des programmes réguliers [54, 55, 103].

Il faut donc faire la distinction entre **recouvrements calculs-communications** qui ont lieu à l'intérieur d'un processeur lorsqu'une unité de communication est capable d'envoyer des données sur un lien tandis que le processeur travaille et **schémas macro-pipelines** ou **pipelines de calculs** dans lesquels on brise la séquentialité d'un algorithme en envoyant au plus tôt des données calculées afin que le processeur voisin commence son travail dès que possible. On a alors un recouvrement des calculs entre eux sur les différents processeurs. On peut bien sûr combiner les deux optimisations et recouvrir les communications dans un schéma macro-pipeline, ce qui permet d'obtenir les meilleures performances possibles.

Je vais détailler dans ce chapitre ces deux types de recouvrements et comment ils peuvent être combinés pour masquer le plus possible la latence du réseau.

## 2.2 Recouvrements calculs/communications et tolérance à la latence

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

Au point de vue architectural, si les recouvrements sont complets, c'est qu'il existe des dispositifs permettant d'envoyer les données sur les canaux de communication sans interrompre le processeur de calcul. Ceci est réalisé généralement grâce à des DMA<sup>1</sup> ou à des co-processeurs de communication [77]. Il faut aussi que les accès aux données pour les calculs ou les communications puissent se faire de manière concurrente, sans partage de bus mémoire par exemple. Dans les machines à mémoire partagée, on peut également avoir des contrôleurs qui effectueront les accès aux bancs distants de manière asynchrone [32, 77]. Le livre de Culler et al. [77] présente de manière précise les aspects architecturaux des recouvrements calculs/communications, et ceci pour les machines à mémoire distribuée et pour les machines à mémoire partagée.

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

---

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

Avec ce type d'appel, le programmeur reprend la main immédiatement après l'appel de la routine. Pour peu qu'il n'ait pas à utiliser le tampon de communication, il peut effectuer d'autres tâches pendant que la communication s'exécute en arrière-plan. Lorsque le programmeur voudra (ré-)utiliser le tampon de communication, il ne pourra le faire qu'après l'appel d'une routine d'attente/test (`MPI_Wait`/`MPI_Test` en MPI). Dans une architecture à mémoire partagée (ou virtuellement partagée), on pourra effectuer des opérations de *prefetch* asynchrones grâce auxquelles on pourra récupérer les données nécessaires à des calculs futurs pendant l'exécution d'autres opérations [187, 245]. Enfin, pour ces deux types de machines, on peut mettre les communications dans des processus légers (*threads*) qui permettront de minimiser les latences de communication en s'exécutant de manière asynchrone avec les calculs [32, 49, 70, 207, 252]. En effet, un changement de contexte sera effectué dès qu'une communication (ou une entrée/sortie en général) sera en attente.

En ce qui concerne les bibliothèques de communication, de nombreux articles étudient les capacités de recouvrements de diverses interfaces de communication et de diverses architectures. Dans [248], les auteurs évaluent les capacités de deux machines (EM-X et IBM SP-2) à recouvrir les calculs et les communications. Ces résultats sont étendus dans [247], avec l'étude des capacités de la SGI PowerCHALLENGE et de l'IBM SP-2. Leurs conclusions sont que le SP-2 permet de meilleurs recouvrements des communications que la SGI grâce à l'utilisation d'un co-processeur de communication. Certaines bibliothèques de communication ont été conçues pour tirer parti des recouvrements calculs-communications. C'est le cas de MPI, qui est présenté par la suite, et de la bibliothèque décrite dans [26]. Il s'agit de découper chaque appel de communication en trois composantes : initialisation, exécution et complétion. L'exécution peut alors se faire de manière totalement asynchrone. L'intérêt de cette bibliothèque est que ce principe a également été appliqué aux communications collectives, ce qui permet par exemple d'obtenir un échange total non-bloquant.

L'interface MPI fournit la possibilité d'effectuer des émissions ou des réceptions non-bloquantes. Par contre, elle ne requiert pas que ces communications soient effectuées de manière asynchrone. Le réseau Myrinet est l'un des réseaux les plus rapides actuellement pour connecter des grappes de PC. Nous avons étudié les possibilités de recouvrements de ce type de réseau avec l'interface BIP ou MPI [67]. Nous avons montré que, même si Myrinet possède une interface réseau (le LANAI) sur laquelle on peut exécuter un code de gestion des communications, les implémentations de MPI classiques (LAM [201], MPICH [202]) n'utilisent pas cette fonctionnalité. Par contre, BIP [35] et sa version MPI l'utilisent de manière efficace.

Dans [156], les auteurs constatent que dans certaines implémentations de MPI (IBM SP-2 et SGI Origin2K), les recouvrements sont quasiment inexistant malgré le fait que les communications soient non-bloquantes. Ces résultats sont en contradiction avec ceux présentés auparavant [247]. Sur le CRAY T3E, les résultats sont surprenants puisque les tests avec recouvrements ont des performances supérieures aux prédictions. La SP-2 d'IBM est encore étudiée dans [62] pour l'implémentation sur cette machine des messages actifs [120]. Ceux-ci permettent de réduire la latence des communications grâce à des appels proches des RPC. Cette bibliothèque possède également des routines asynchrones (`am_store_async`) permettant les recouvrements. Des routines non-bloquantes sont utilisées pour éviter les blocages.

Il est également possible de mixer *threads* et communications pipelines pour améliorer les performances des applications. C'est ce qui est décrit par exemple dans [252] où les auteurs optimisent un code de gradient conjugué sur une grappe de SMP. Il faut par contre que la bibliothèque de communication soit *thread-safe*. Pour ce type d'architecture, un autre problème réside dans le fait que les communications doivent être hybrides (par mémoire partagée à l'intérieur d'une grappe et par passage de messages « classique » entre les grappes).

Les recouvrements calculs/communications simples peuvent être trouvés relativement facilement dans de nombreuses applications du calcul numérique. On peut citer par exemple le gradient conjugué [195], la factorisation *LU* [61, 90, 241], le produit de matrices [3, 61, 97, 241], la simulation de front de flamme [95], l'algorithme du simplexe [1] et même certaines applications irrégulières [181], Gauss-Seidel (avec un algorithme Red-Black) [21], etc. Un grand nombre



d'applications ont également été testées dans [188] pour montrer les performances des *prefetch* dans les machines à mémoire partagée. On peut citer par exemple la factorisation de Cholesky, le gradient conjugué, la FFT, l'élimination de Gauss, le SOR, les tris et bien d'autres. Les optimisations effectuées ne sont pas décrites et il est difficile de savoir s'il s'agit de recouvrements simples ou pipelines. Ces applications n'obtiennent pas toutes des gains importants. Le calcul de la distribution des données peut tenir compte de la présence ou pas de recouvrements calculs/communications. Par exemple, dans [122], dans le cas d'une grappe de machines SMP, les auteurs optimisent la distribution irrégulière des données pour un algorithme de différences finies en introduisant les recouvrements calculs/communications dans le modèle d'évaluation. Dans [214], une telle optimisation est réalisée dans le cas de la parallélisation d'un code de factorisation *LU*. Par ailleurs, toutes les applications des macro-pipelines présentées dans la section suivante sont également des applications potentielles des recouvrements puisqu'il est possible (et même conseillé) d'utiliser des communications asynchrones dans un schéma macro-pipeline. Dans le chapitre B du document annexe, je présente nos résultats sur les recouvrements calculs/communications avec la factorisation *LU*.

## 2.3 Problématique et applications des macro-pipelines

La technique des macro-pipelines est en fait assez proche de celle des pipelines de communication, utilisée au départ dans les machines parallèles à mémoire distribuée utilisant le protocole *store-and-forward*. En effet, dans de telles architectures, un message passait de mémoire en mémoire jusqu'à atteindre la mémoire du processeur destination. Afin de masquer la distance, le message était découpé en messages de plus petites tailles qui étaient envoyés les uns à la suite des autres. Ces techniques sont toujours utilisées pour masquer les latences dans les réseaux rapides comme par exemple Myrinet [224, 265].

Du fait de certaines dépendances dans le code, il n'est pas toujours facile (voire possible) de paralléliser un code. Un bon exemple est la compilation des boucles *DOACROSS* dans les langages data-parallèles. Les dépendances entre les instructions ou à l'intérieur d'une même instruction peuvent empêcher l'exécution en parallèle d'un code. Prenons l'exemple de la figure 2.1. La dépendance entre les itérations au sein de l'instruction  $S_1$  donnera une exécution séquentielle de la boucle.

```

do i=0, time_step
  do j=1, n-2
    S1   a(j) = 0.5*(a(j-1) + a(j+1))
  end do
end do

```

FIG. 2.1 – Boucle *DOACROSS* séquentielle.

```

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

```

FIG. 2.2 – Boucle *DOACROSS* parallèle avec distribution par blocs.

Si la distribution choisie pour le vecteur *a* est *BLOCK*, la boucle sera parfaitement séquentielle (figure 2.2). Une solution consiste à choisir une distribution de type *CYCLIC(K)* ou *K* est choisi en fonction de la machine cible. Par contre, celle-ci entraîne la génération de nombreuses communications, ce qui peut conduire à une augmentation du temps total d'exécution. Parfois, il se peut que la distribution choisie pour une matrice dépende de calculs précédents et ne puisse être changée. On peut alors faire appel à des optimisations de type macro-pipeline pour briser cette séquentialité. L'utilisation d'un macro-pipeline consiste à réduire le grain de calcul afin de communiquer plus tôt les données nécessaires au démarrage des calculs sur le processeur suivant. On peut bien sûr envoyer chaque donnée dès qu'elle a été calculée ; ce qui a pour effet de faire démarrer le processeur suivant au plus tôt (et ainsi de suite). Par contre, le coût de

communication des machines actuelles induit un temps d'initialisation d'une communication qui est indépendant du nombre de données envoyées. Ce coût est généralement très élevé et il faut donc réduire le nombre de communications en envoyant plusieurs données à la fois (ce qui retarde le démarrage du processeur suivant!). Le grain de calcul (et de communication) est donc un paramètre très important qu'il convient de calculer avec soin. Nous présentons diverses méthodes de calcul pour des cas de figures différents dans la section 2.6. L'exécution d'un macro-pipeline donnera un diagramme de Gantt comme celui présenté dans la Figure 2.3. On notera sur ce schéma qu'il y a en fait deux types de recouvrements : un **recouvrement des communications sur un processeur**, et un **recouvrement des calculs entre les processeurs**. Si l'on utilise un tel schéma sur plusieurs processeurs, on aura un gain supérieur puisque les calculs seront recouverts entre tous les processeurs (figure 2.4). Le calcul de tels gains est présenté dans la section 2.4.

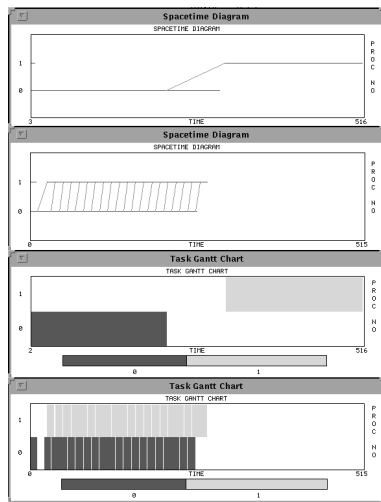


FIG. 2.3 – Diagramme de Gantt d'un programme pipeline entre deux processeurs.

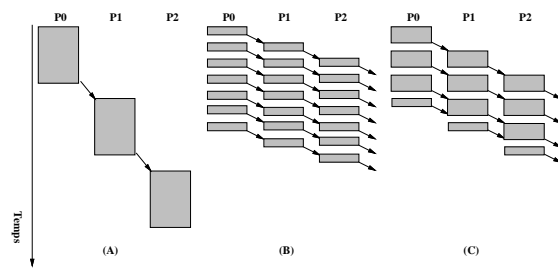


FIG. 2.4 – Programme pipeline entre plusieurs processeurs.

La problématique des macro-pipelines est donc **algorithmique**. En effet, aucun support d'exécution, aussi évolué soit-il, est capable de décider s'il faut ou pas envoyer des données à partir d'un certain nombre, afin d'équilibrer au mieux calculs et communications dans un algorithme parallèle.

### 2.3.1 Exemples applicatifs de recouvrements pipelines

Nous allons donner maintenant quelques exemples d'applications qui peuvent bénéficier de recouvrements pipelines.

### 2.3.2 Alternative Direction Implicit (ADI)

Une application de telles optimisations est illustrée dans le code ADI (Alternative Direction Implicit) de la figure 2.5. Cette application est citée dans de nombreux articles comme application de référence [48, 189, 217, 218, 260, 261]. Ce problème, divisé en deux phases de calculs, nécessite une modification du code si l'on veut réduire le nombre des communications. Les solutions consistent soit à redistribuer les données après la première phase de calcul et ainsi éliminer les communications pour la seconde phase, ou à pipeliner les communications dans la seconde

phase. La version pipeline de l'ADI est donnée dans la figure 2.6. Ici, la redistribution est une simple transposition.

```

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

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

```

FIG. 2.5 – ADI séquentiel.

```

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

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

```

FIG. 2.6 – ADI pipeline.

Une implémentation de cet algorithme optimisée avec la bibliothèque LOCCS et un compilateur HPF est présentée dans la section 2.7.2.

### 2.3.2.1 Successive overrelaxation (SOR)

La surrelaxation successive (*Successive overrelaxation* ou *SOR*) [222] est une méthode itérative utilisée pour la résolution de l'équation de Laplace, l'équation différentielle partielle

$$\frac{\delta^2 U(x,y)}{\delta^2 x} + \frac{\delta^2 U(x,y)}{\delta^2 y} = 0 \quad (2.1)$$

sur un domaine carré avec des valeurs frontières connues. Le domaine de taille  $x \times n$  est représenté par une matrice qui est initialisée avec une approximation de la solution. L'algorithme séquentiel est donné dans la figure 2.7.

Le SOR a été étudié à de nombreuses reprises avec des recouvrements calculs/communications et des pipelines [188, 211, 218, 241, 257]. La principale difficulté vient du fait que l'on a des communications « montantes » (voir figure 2.8). Celles-ci sont ajoutées au pipeline et on doit en tenir compte dans l'évaluation du grain de calcul. Les résultats de nos expériences sur le SOR avec les bibliothèques LOCCS et OPIUM sont donnés dans la section 2.6.3.

### 2.3.2.2 Factorisation LU

La factorisation *LU* est un noyau de calcul important dans de nombreuses applications numériques. Elle a également été utilisée de manière intensive pour l'évaluation de machines parallèles diverses et variées (cf benchmark LINPACK). Cet algorithme a été plusieurs fois étudié au niveau des recouvrements calculs-communications et recouvrements pipelines [1, 99, 106, 188]. Si l'on choisit une distribution cyclique, cet algorithme possède déjà un pipeline « naturel » dû au passage des mises à jour sur une sous-matrice de taille toujours moins importante. Ce pipeline permet déjà d'obtenir un bon recouvrement entre les calculs et

```

iter = 0
error = 1.0
while (error < ζ & iter < maxiter)
  do i=1, n
    do j=1, n
      prev = b(i,j)
      b(i,j) = 0.493*(b(i,j-1) + b(i-1,j)
        + b(i,j+1) + b(i+1,j) + -0.972*prev)
      norm = norm + prev2
      error = error + (b(i,j) - prev)2
    end do
  end do
  error = error/norm
  iter = iter + 1
end while

```

FIG. 2.7 – Algorithme séquentiel pour le SOR.

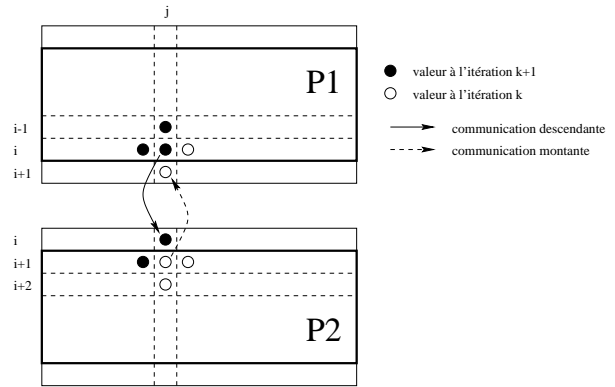


FIG. 2.8 – Détail de l'exécution du SOR sur une frontière.

les communications mais il peut encore être amélioré, que ce soit avec une distribution 1D ([99] et chapitre C du document annexe) que 2D ([106] et chapitre B du document annexe).

### 2.3.2.3 Transformée de Fourier rapide

La Transformée de Fourier Rapide est un algorithme qui se prête bien à une parallélisation à l'aide de macro-pipelines et ceci malgré un schéma de communication relativement compliqué (papillons).

Dans [247], un algorithme de FFT 1D est parallélisé avec un macro-pipeline en découpant les données disposées sur chaque processeur en segments et en calculant sur un segment pendant la réception d'un autre segment. Les recouvrements obtenus sont de 80% sur une IBM SP-2 et de 20% sur une SGI PowerCHALLENGE. La FFT est également présentée dans [178] pour valider la modélisation de pipelines de communications dans un hypercube  $n$ -ports. Dans [1], la FFT est utilisée pour montrer l'intérêt de techniques de *prefetch* dans les machines COMA et dans [70], la FFT 2D est utilisée pour valider des développements d'une implémentation de MPI sur une grappe de SMP.

Dans [55], nous avons présenté avec C. Calvin une distribution des données pour la FFT bi-dimensionnelle qui permet un recouvrement total des communications par les calculs et une utilisation optimale de la bande passante de l'architecture sur un hypercube d'Intel.

### 2.3.2.4 Shear-Warp

Avec F. Chaussumier, nous avons parallélisé un algorithme de rendu volumique pour l'imagerie médicale, le *shear-warp* [179]. La parallélisation complète de cet algorithme est donnée dans le chapitre D du document annexe mais je vais en donner ici les grandes lignes afin de comprendre l'utilisation de pipelines de calculs dans une application irrégulière. Notre but était d'obtenir une exécution en temps réel lorsque le praticien change le point de vue avec un angle quelconque. Seule une version parallèle est capable de donner de telles performances et nous avons choisi de paralléliser cet algorithme sur une grappe de PC connectés par un réseau Myrinet.

Cet algorithme s'appuie sur la factorisation de la transformation liée au point de vue lui permettant de parcourir simplement le volume de données tout en permettant d'utiliser des optimisations séquentielles. Il se décompose en deux étapes : la première mettant en œuvre la transformation du volume en trois dimensions en une image intermédiaire en deux dimensions (*shear*) et la seconde passant de cette image intermédiaire à l'image finale (*warp*). L'essentiel des calculs se trouve dans l'étape de *shear* et nous nous sommes donc concentrés sur celle-ci.

Le *shear-warp* encode l'objet avec une structure creuse en *run length encoding* (RLE). Ce codage est utilisé pour encoder des objets présentant une cohérence de données, notamment des images pour lesquelles on obtient un taux de compression intéressant. Ceci est le cas pour les images issues de modalités médicales telles que le scanner ou la résonance magnétique que nous avons considérées. Par contre, il donne une structure de données irrégulière qu'il faut prendre en compte lors de l'utilisation d'un pipeline de calculs.

Nous avons choisi un partitionnement selon l'image intermédiaire car il permet de conserver l'optimisation de la détection anticipée de rayon en fonction des opacités. Par contre, celui-ci est plus coûteux en nombre et en volume de communications dans une machine à mémoire distribuée. La taille des données manipulées ne permet pas d'avoir beaucoup de réplication.

Notre premier travail a consisté à effectuer un équilibrage des charges en fonction des variations dans l'angle de point de vue. La figure 2.9 montre les variations de charge en fonction des variations d'angle pour 1, 10 et 20 degrés. Les travaux précédents effectuaient soit du vol de tâches [180] (sur machine à mémoire partagée) soit ils ne considéraient que des variations d'angles très petites [7]. Dans notre cas (machine à mémoire distribuée et angles quelconques), ces équilibrages n'étaient pas satisfaisants, nous avons donc utilisé l'algorithme élastique de Miguet et Robert [196] adapté à notre algorithme.

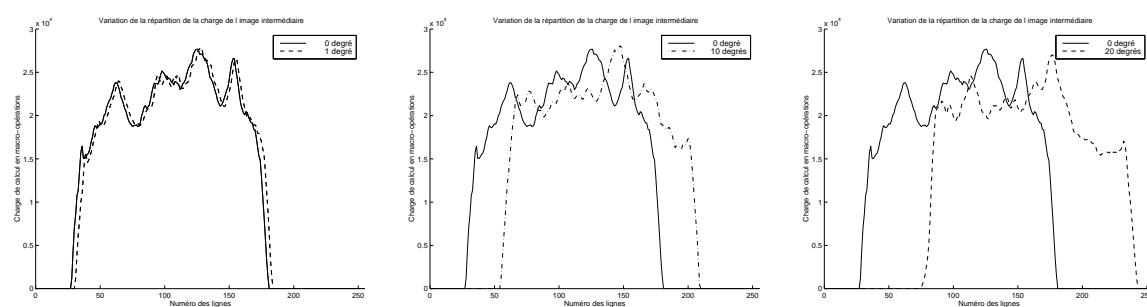


FIG. 2.9 – Répartition de la charge de l'image intermédiaire en fonction de l'angle de rotation.

Ensuite, nous avons optimisé l'une des étapes de communications les plus coûteuses de l'application, la redistribution des données lors des changements de points de vue avec une multidistribution. Nous avons pu utiliser à la fois des recouvrements calculs-communication simples lorsque le changement d'angle est faible et des recouvrements pipelines lorsque l'angle augmente. La difficulté vient de l'irrégularité des structures de données et du choix de stratégie en fonction de l'angle choisi. De plus, il est quasiment impossible de modéliser précisément les différents coûts de calculs et de communication et le grain doit donc être choisi dynamiquement.

L'implantation parallèle de l'algorithme de rendu volumique en *shear-warp* sur une grappe de PC que nous avons proposée a permis d'augmenter son interactivité, c'est-à-dire qu'elle permet à l'utilisateur d'obtenir une image en temps réel pour un point de vue quelconque pour un encombrement mémoire minimal. En effet, pour rendre une image correspondant à  $512^3$  voxels, il faut 1.5 secondes sur 4 processeurs.

### 2.3.2.5 Sweep 3D

Le Sweep3D est un solveur d'une équation de transport de particules en trois dimensions et indépendant du temps [22, 149, 174]. Cette application calcule le flux de particules traversant une région donnée de l'espace, flux qui, pour chaque région, est dépendant du flux venant des cellules voisines. L'espace 3D est discrétisé en cellules 3D. Le calcul s'exécute par vagues à partir des huit octants de l'espace 3D, chaque octant contenant six angles de direction de

flux indépendants (un pour chacune des faces des cellules cubiques). C'est l'un des benchmarks du programme américain ASCI [18] et le code est récupérable sur le réseau. Le Sweep3D décompose l'espace 3D en deux dimensions sur une grille en deux dimensions de processeurs dans les directions  $I$  et  $J$ . Dans cette configuration, un processeur calcule un flux traversant une colonne de cellules et envoie ensuite les frontières nécessaires à ses deux processeurs voisins. Pour améliorer les performances, les données suivant la dimension  $K$  et les angles de direction sont divisés en blocs, permettant ainsi à un processeur de ne calculer qu'une partie des valeurs selon la dimension  $K$  et qu'une partie des angles avant de les envoyer aux processeurs voisins. Le Sweep3D exploite plusieurs types de parallélisme :

- du parallélisme inhérent à cette application sur les angles : chaque balayage pour un angle donné est indépendant ;
- du parallélisme par vague : décomposition spatiale en 2D sur une grille 2D de processeurs ;
- de plus, des blocs de calcul en la dimension  $K$  et en nombre d'angles de direction sont pipelinés sur la grille de processeurs ;
- enfin, le démarrage du calcul d'un octant peut démarrer avant la fin du calcul de l'octant précédent, ceci se limite aux calculs de deux octants simultanément.

Nous nous sommes intéressés au pipeline des calculs sur la dimension  $K$ . En effet, le découpage en paquets des blocs de données sur la dimension  $K$  induit un pipeline à deux dimensions sur les deux autres dimensions. Cette étude nous a permis de valider les résultats présentés dans la section 2.4 [68]. Les résultats complets seront donnés dans la thèse de F. Chaussumier.

### 2.3.2.6 Autres applications

D'autres applications numériques ont été également étudiées comme par exemple la simulation Airshed [189], le code Hydro [189, 218, 257] adapté du noyau 23 du benchmark Livermore, la résolution de systèmes triangulaires [218], la factorisation de Cholesky [188, 229], le gradient conjugué [188, 252], Gauss-Seidel [236], le produit de matrices [21, 165, 173], le benchmark NAS-MG [21], etc.

## 2.4 Gains prévus

Une optimisation dans un programme parallèle n'a de sens que si elle apporte un gain en performances ou en utilisation de la mémoire. Suivant les paramètres de la machine cible, les caractéristiques et les dépendances de l'algorithme, il ne sera pas forcément intéressant de vouloir à tout prix utiliser les pipelines de calculs dans une application. Dans cette section, nous allons donc étudier de manière théorique les gains apportés par les recouvrements calculs/communications en général et les macros-pipelines en particuliers.

J'ai étudié ce problème dans un premier temps avec Julien Zory de manière théorique (section 2.4.2) puis avec Frédérique Chaussumier en améliorant les résultats de Julien et en les appliquant à une application, le Sweep3D (section 2.3.2.5).

### 2.4.1 Travaux précédents

Une étude de Quinn et Hatcher [225] donne une évaluation des gains des recouvrements calculs/communications. Grâce à une modélisation du comportement de la machine cible en terme de communication, de bufferisation et de calculs, Quinn et Hatcher calculent le gain que l'on peut espérer obtenir sur un processeur en utilisant diverses techniques de recouvrements. Les résultats prouvent qu'un gain de 2 au maximum peut être obtenu dans le meilleur des cas et que bien souvent ce gain est inférieur. Les résultats que nous présentons dans cette section sont des extensions de l'article de Quinn et Hatcher. Dans [104] nous prouvons que des gains supérieurs à deux peuvent être obtenus par des techniques de macro-pipelines si l'on tient compte de l'ensemble de la machine et non pas uniquement des gains sur chaque processeur. En effet, l'intérêt

de telles techniques réside surtout dans le fait que l'on brise la séquentialité du programme en pipelinant les messages (et donc les calculs). Sur  $P$  processeurs, on peut ainsi obtenir un gain de  $P - 1$  grâce à la fois aux recouvrements des calculs et des communications, mais aussi grâce aux recouvrements des calculs entre eux sur différents processeurs. Je présente ces résultats dans la section suivante.

Par ailleurs des tests de techniques macro-pipelines à une dimension ont été effectués dans le cadre de l'optimisation des communications dans les compilateurs paralléliseurs [28, 147, 257, 139, 47, 218], d'outils de parallélisation [121], etc. On trouve aussi dans [188] une modélisation précise de l'accélération qui peut être atteinte en utilisant à la fois des threads et des *prefetch* dans une machine à mémoire partagée. Il s'agit toutefois de programmes avec des accès aux données simples.

## 2.4.2 Calcul des gains pour différentes formes de pipelines

Nous avons repris l'article de Quinn et Hatcher pour montrer que si les recouvrements calculs/communications ne pouvaient donner qu'un gain de 2 sur un seul processeur, on pouvait obtenir des recouvrements plus intéressants avec des macros-pipelines sur un plus grand nombre de processeurs.

### 2.4.2.1 Approche Quinn et Hatcher

Quinn et Hatcher [225] utilisent une modélisation à trois composants pour les recouvrements :

- le temps passé par le processeur pour préparer le message (ou pour le récupérer) appelé aussi latence. Cette latence est modélisée par  $L(n) = \lambda + \beta n$  où  $\lambda$  est un terme constant représentant le temps nécessaire pour gérer un appel d'envoi ou de réception,  $\beta$  est inversement proportionnel à la vitesse à laquelle le système peut copier le message et  $n$  est la longueur du message. Cette latence d'émission ou de réception ne peut pas être recouverte ;
- le temps de transmission proprement dit du message (temps de déplacement du message sur le réseau). Il est modélisé par  $T(n)$  et est également linéaire par rapport à la taille du message ( $\beta_T + \tau_T n$ ). Celui-ci peut être recouvert.
- Le temps de calcul est séparé en un temps  $C_o(n)$  qui peut être recouvert par la transmission et un temps  $C_r(n)$  qui ne peut être recouvert. Ce sont ici des considérations essentiellement algorithmiques.

Ils utilisent également un code générique qui permet de caractériser les divers types de recouvrements (figure 2.10). Dans ce code, on peut effectuer une première optimisation en déplaçant la communication  $\langle Com_2 \rangle$  avant le segment de code  $\langle CS_2 \rangle$  ce qui permet d'effectuer une communication asynchrone. Le code résultant est donné dans la figure 2.11.

$\langle Com_1 \rangle$	Communication 1	$\langle Com_1 \rangle$	
$\langle CS_1 \rangle$	Seg. de code 1 (utilise une variable M)	$\langle CS_1 \rangle$	
$\langle CS_2 \rangle$	Seg. de code 2 (n'utilise pas de variable M)	$\langle Com_2 \rangle$	
$\langle Com_2 \rangle$	Communication 2 (avec M)	$\langle CS_2 \rangle$	} $C_o$ } $C_r$
$\langle CS_3 \rangle$	Seg. de code 3 (n'utilise pas de variable M)	$\langle CS_3 \rangle$	
$\langle CS_4 \rangle$	Seg. de code 4 (utilise une variable M)	$\langle CS_4 \rangle$	
$\langle Com_3 \rangle$	Communication 3	$\langle Com_3 \rangle$	

FIG. 2.10 – Algorithme de référence pour les différentes méthodes de recouvrement.

FIG. 2.11 – Algorithme de référence après la première optimisation.

Quinn et Hatcher distinguent alors quatre cas possibles suivant que les segments de codes  $C_o$  et  $C_r$  sont vides ou pas. Leurs noms génériques sont donnés dans le tableau 2.1. Les recouvrements systolique et général sont assez similaires vu qu'ils utilisent tous deux des communications

asynchrones simples comme nous l'avons vu dans la section 2.2. Ils diffèrent uniquement dans le gain obtenu. Dans le cas où les dépendances de données empêchent le recouvrement ( $C_r$  contient du code et  $C_o$  vide), on peut effectuer un pipeline en restructurant le code. Quinn et Hatcher parlent alors de « recouvrement pipeliné ».

		Segment $C_o$	
		Contient du code	Vide
Segment $C_r$	Contient du code	Recouvrement général	Les dépendances de données empêchent le recouvrement
	Vide	Recouvrement systolique	Pas de calcul à recouvrir avec les communications

TAB. 2.1 – Quatre cas pour le problème du recouvrement.

Dans leur article, Quinn et Hatcher évaluent ensuite le gain des recouvrements en comparant asymptotiquement les valeurs des latences, des calculs et des communications. Le tableau 2.2 donne les équations des gains pour les différents types de recouvrements. Le gain d'un recouvrement est calculé en divisant le temps d'exécution **sans** optimisation avec le temps **avec** optimisation. Dans le cas du recouvrement pipeline,  $n_0$  représente la taille d'un paquet de données envoyées. Celui-ci peut être calculé avec les méthodes données dans la section 2.6.

Type de recouvrement	Gain
Recouvrement systolique	$S(n) = \frac{\min(C_o(n), T(n))}{L(n) + \max(C_o(n), T(n))}$
Recouvrement général	$G(n) = 1 + \frac{\min(C_o(n), T(n))}{L(n) + C_r(n) + \max(C_o(n), T(n))}$
Recouvrement pipeliné	$P(n) = \frac{L(n) + T(n) + C_r(n)}{(n/n_0)L(n_0) + ((n/n_0) - 1) \max(T(n_0), C_r(n_0)) + T(n_0) + C_r(n_0)}$

TAB. 2.2 – Complexités des différents recouvrements.

Type de recouvrement	Gain asymptotique	Corollaire
Recouvrement systolique et général	$S(n) \leq 2$	1
	si $\lim_{n \rightarrow +\infty} T(n)/C_o(n) = 0$ et $\lim_{n \rightarrow +\infty} L(n)/C_o(n) = 0$ alors $\lim_{n \rightarrow +\infty} SG(n) = 1$	2
Recouvrement pipeline	$\lambda < \min(T(n_0), C_r(n_0))$	3
	Si $\lim_{n \rightarrow +\infty} \frac{T(n)}{C_r(n)} = 0$ et $\lim_{n \rightarrow +\infty} \frac{L(n)}{C_r(n)} = 0$ alors $\lim_{n \rightarrow +\infty} S(n) \leq 1$	4

TAB. 2.3 – Gains asymptotiques des différents recouvrements.

Suivant le rapport entre le coût de communication (latence et transmission) et le coût de calcul, on peut évaluer asymptotiquement les gains qui peuvent être obtenus avec ce type d'optimisations. Le tableau 2.3 résume les résultats donnés dans [225].

La différence entre le recouvrement systolique et le recouvrement général réside uniquement dans la présence ou pas du terme  $C_r(n)$  au dénominateur. Le premier corollaire dit que le recouvrement ne peut dépasser deux. Il est facile de comprendre que l'on a  $(A + B)/\max(A, B) \leq 2$ . Ce gain est encore plus dur à atteindre si les calculs sont très grands devant les communications (corollaire 2). En effet, la présence du terme  $C_r(n)$  au dénominateur de  $G(n)$  rend encore plus difficile d'atteindre la borne de 2.

En ce qui concerne le recouvrement pipeline qui nous intéresse ici, un gain sera obtenu si  $\lambda < \min(T(n_0), C_r(n_0))$  (corollaire 3), c'est-à-dire si la latence n'est pas trop importante. Intuitivement, si la latence est grande, on aura alors intérêt à envoyer des gros messages et ainsi on se rapprochera du cas non-pipeliné. Au contraire, si elle est négligeable, on se rapprochera



du recouvrement systolique. La latence intervient donc bien sûr dans le calcul du grain optimal. Le corollaire 4 montre que si les communications sont négligeables, alors le gain à pipeliner **sur un processeur** est nul. Les conclusions générales de Quinn et Hatcher sont donc que les recouvrements calculs/communications ne sont pas intéressants en terme de gain de performances par rapport aux développements supplémentaires qu'ils nécessitent.

### 2.4.2.2 Pipeline multi-dimensionnel

A la lecture de cet article, nous pouvions donc nous poser la question de l'intérêt de poursuivre des recherches dans ce domaine! Heureusement, si le pipeline n'offre certes pas beaucoup d'intérêt lorsqu'on l'étudie au niveau d'un processeur, par contre, il prend tout son intérêt lorsqu'on étudie les gains qui peuvent être obtenus sur plusieurs processeurs. Nous avons donc étudié les gains des pipelines dans le cas où l'on a plus de deux processeurs et nous avons également étendu ce type d'optimisation pour des pipelines multi-dimensionnels.

Les paramètres qui peuvent varier sont : la dimension des matrices ( $N_m$ ), le nombre de dimensions distribuées ( $N_d$ ) et les dépendances (vecteurs de la forme  $(0, \dots, 0, d_i > 0, 0, \dots, 0)$  pour  $0 \leq i \leq N_m - 1$ ). Nous supposons qu'il existe au moins un vecteur de dépendances pour chacune des dimensions distribuées  $d_i$ , sans quoi plusieurs macro-pipelines de dimensions inférieures seront lancés en parallèle.

Dans cette section, je résume nos principaux résultats. Les preuves (et les équations complètes du cas à deux dimensions) peuvent être trouvées dans [68, 274].

**Matrice bi-dimensionnelle distribuée selon une seule dimension** C'est le cas classique de l'ADI lorsqu'on distribue une seule dimension de la matrice de départ. Le découpage est effectué sur la dimension qui n'est pas distribuée (avec une taille de message de  $n_0$ ). Nous avons montré que le gain qui pouvait être obtenu dans ce cas était égal à

$$P(n) = \frac{PC_r(n) + (P-1)[2L(n) + T(n)]}{PC_r(n_0) + (P-1)[2L(n_0) + T(n_0)] + \left(\frac{n}{n_0} - 1\right) [\max(L(n_0) + C_r(n_0), T(n_0)) + L(n_0)]} \quad (2.2)$$

*Théorème*

Si  $\lim_{n \rightarrow +\infty} \frac{T(n)}{C_r(n)} = 0$  et  $\lim_{n \rightarrow +\infty} \frac{L(n)}{C_r(n)} = 0$  alors le gain maximum qui peut être obtenu est toujours inférieur à  $P$ , le nombre de processeurs.

On constate donc que si les latences sont négligeables et le nombre de processeurs assez important, il est intéressant de pipeliner les calculs.

**Matrice bi-dimensionnelle distribuée selon deux dimensions** On peut également pipeliner les calculs si la matrice est distribuée par blocs dans les deux dimensions sur une grille de  $\sqrt{P} \times \sqrt{P}$  processeurs. Nos résultats peuvent être facilement étendus à des grilles quelconques et le choix de la taille et de la forme de la grille dépendent des matrices en entrée et des paramètres de la machine.

Dans ce cas, on a déjà un pipeline issu de la distribution des données en 2D (voir figure 2.12). Si nous avons des dépendances de type  $(d_1, 0)$  et  $(0, d_2)$  avec  $d_1, d_2 > 0$ , on peut effectuer un pipeline dans l'une des deux dimensions. La figure 2.12 montre un pipeline mono-dimensionnel dans la première dimension d'une grille 2D. La partie grisée montre le chemin critique d'exécution.

*Théorème*

Si  $\lim_{n \rightarrow +\infty} \frac{T(n)}{C_r(n)} = 0$  et  $\lim_{n \rightarrow +\infty} \frac{L(n)}{C_r(n)} = 0$  alors le gain maximum qui peut être obtenu est toujours inférieur à 2 (la démonstration est donnée dans [274]).

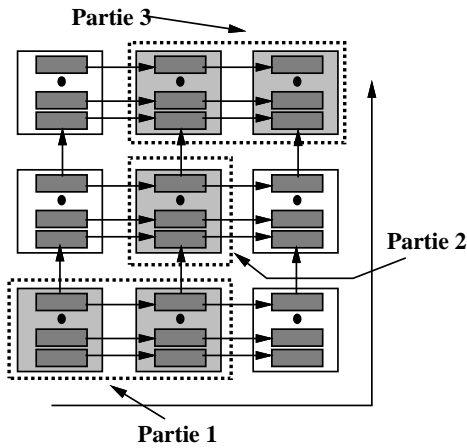


FIG. 2.12 – Pipeline mono-dimensionnel avec une matrice 2D distribuée sur une grille de processeurs.

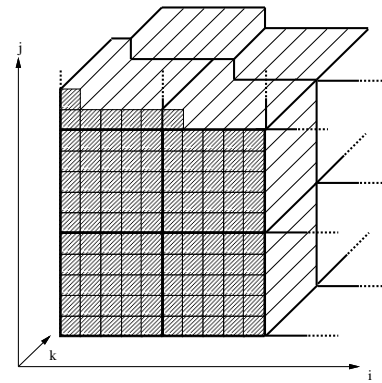


FIG. 2.13 – Pipeline bi-dimensionnel avec une matrice 3D distribuée sur un tore 3D de processeurs.

Dans ce cas, le gain n'est pas très important comparé à la difficulté de programmation ou à la difficulté de génération de code pour le compilateur. C'est pour cette raison que la plupart des exemples de pipelines de calcul de la littérature présentent des pipelines mono-dimensionnels sur des matrices 2D.

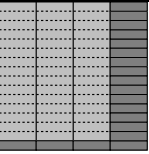
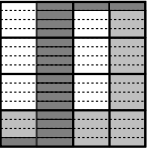
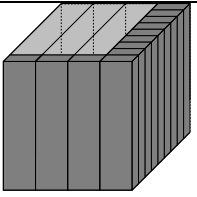
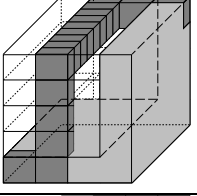
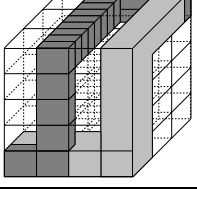
**Extension à des matrices de dimension quelconque** Le tableau 2.4 résume les gains asymptotiques qui peuvent être obtenus en pipelinant une dimension pour des matrices 2D et 3D distribuées selon une, deux ou trois dimensions. On constate que les gains sont variables selon le nombre de dimensions des données en entrées et le nombre de dimensions distribuées. On vérifie aisément que si le nombre de dimensions distribuées est important alors il n'est pas très intéressant de pipeliner les calculs.

**Augmentation du nombre de pipelines** Lorsque l'on travaille sur des matrices 3D, on peut pipeliner plus d'une dimension en même temps. Il y a bien sûr un surcoût dû à l'augmentation du nombre de messages et un compromis doit être trouvé. Avec des pipelines multiples, on peut accélérer le démarrage des processeurs voisins. Par exemple, on voit dans la figure 2.13 que les processeurs dans la dimension  $k$  démarreront plus rapidement dans la dimension  $i$ .

## 2.5 Enchaînements de BLAS

Avec S. Domas, nous nous sommes intéressés à l'enchaînement d'appels à des routines BLAS et ceci de manière pipeline [85, 105]. Notre but a été de mixer l'optimisation avec des pipelines et le découpage d'algorithmes d'algèbre linéaire par blocs dans le cadre d'un programme parallèle utilisant des BLAS de niveau 3. Chaque BLAS 3 est considéré comme une tâche exécutée sur un processeur particulier. Les dépendances sur les accès aux données entre tâches forment un graphe qui va nous servir à déterminer automatiquement la meilleure taille de découpage des différentes tâches lors du pipeline.

La figure 2.15 montre un exemple de chaîne de produits avec des dépendances d'accès aux données. Sans pipeline, l'exécution d'un tel graphe est totalement séquentielle. Nous avons pris

Chemin critique	Paramètres	Gain
	$N_m = 2$ $N_d = 1$ $P \times 1$	$G \simeq \frac{P \times o(n^2) + C_1 \times o(n)}{o(n^2) + C_2 \times o(n)} \simeq P$
	$N_m = 2$ $N_d = 2$ $\sqrt{P} \times \sqrt{P}$	$G \simeq \frac{2\sqrt{P} \times o(n^2) + C_1 \times o(n)}{\sqrt{P} \times o(n^2) + C_2 \times o(n)} \simeq 2$
	$N_m = 3$ $N_d = 1$ $P \times 1 \times 1$	$G \simeq \frac{P \times o(n^3) + C_1 \times o(n^2)}{o(n^3) + C_2 \times o(n^2)} \simeq P$
	$N_m = 3$ $N_d = 2$ $\sqrt{P} \times \sqrt{P} \times 1$	$G \simeq \frac{2\sqrt{P} \times o(n^3) + C_1 \times o(n^2)}{o(n^3) + C_2 \times o(n^2)} \simeq 2\sqrt{P}$
	$N_m = 3$ $N_d = 3$ $\sqrt[3]{P} \times \sqrt[3]{P} \times \sqrt[3]{P}$	$G \simeq \frac{3\sqrt[3]{P} \times o(n^3) + C_1 \times o(n)}{\sqrt[3]{P} \times o(n^3) + C_2 \times o(n)} \simeq 3$

TAB. 2.4 – Gain avec  $N_d$  dimensions distribuées.

comme algorithme cible le produit de matrice car il est au centre de nombreux noyaux de calculs et qu'il permet d'avoir plusieurs types de découpages (voir figure 2.14).

Si au cours de mes travaux précédents, le découpage des calculs et des communications était essentiellement dirigé par les paramètres de la machine cible (grâce à l'orthogonalité des dépendances), l'utilisation d'algorithmes comme le produit de matrice induit la prise en compte de l'algorithme dans le pipeline. De plus, les dépendances entre les données conduisent à des choix sur le type de découpage du calcul pour le pipeline. La figure 2.16 montre les diagrammes de Gantt que nous pouvons obtenir selon des choix de découpage des noyaux de calculs utilisés. Les figures 2.5 et 2.6 donnent le code des différentes tâches en mode pipeline pour la solution la plus efficace en temps d'exécution. Dans la réalité, le noyau de calcul central est toujours un appel à un BLAS de niveau 3.

Nous avons proposé une heuristique [85, 105] qui permet de découper chaque tâche d'une telle chaîne pour obtenir un temps d'exécution minimal. Elle est associée à un calcul du grain optimal en fonction des paramètres de la machine cible et de la modélisation précise des BLAS de niveau trois développée par S. Domas.

Puisque nous modélisons chaque appel de BLAS de manière indépendante, il est donc possible de tenir compte d'une architecture hétérogène où les vitesses des processeurs et des liens de communication ne sont pas les mêmes. Il suffit de mettre dans la modélisation les paramètres de chaque élément de calcul et de chaque lien.

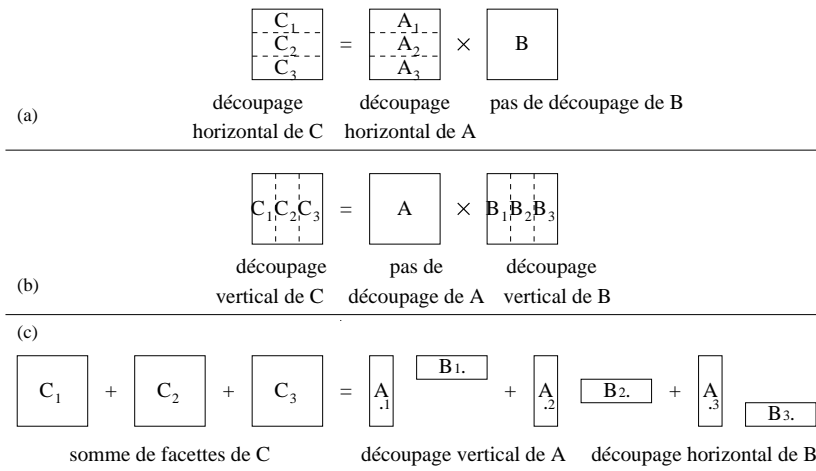


FIG. 2.14 – Trois façons de découper un produit matriciel.

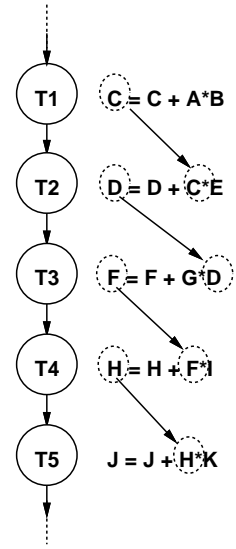


FIG. 2.15 – Pipeline de cinq produits matriciels.

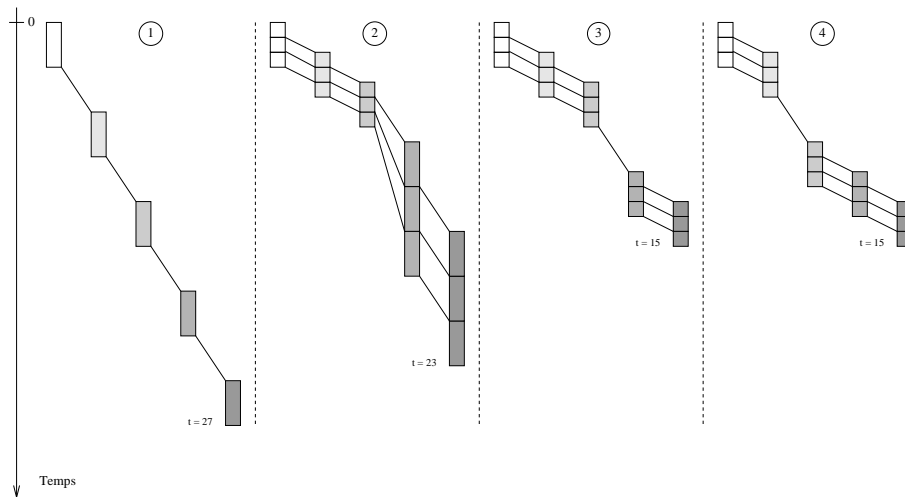


FIG. 2.16 – Diagramme de Gantt pour les quatre solutions de découpage pour le pipeline de cinq produits de matrices.

## 2.6 Calcul de la taille optimale de paquets

### 2.6.1 Introduction

Le grain d'un macro-pipeline est le paramètre qui conditionne l'obtention de performances. Trop petit, il occasionnera le démarrage de trop nombreuses communications ce qui entraînera des pertes de performances dues à leur surcoût et trop gros il réduira le parallélisme. Il peut être calculé de manière analytique en fonction d'une modélisation de l'algorithme et de la machine

<pre> /* tâche 1 (C = C + AB) */ do i_L = 0, M/L   do i = i_L * L, (i_L + 1) * L - 1     do j = 0, N       do h = 0, K         C_ij = C_ij + A_ih B_hj       enddo     enddo   enddo send(sous-matrice C) enddo </pre>	<pre> /* tâche 2 (D = D + CE) */ do i_L = 0, M/L   recv(sous-matrice C)   do i = i_L * L, (i_L + 1) * L - 1     do j = 0, N       do h = 0, K         D_ij = D_ij + C_ih E_hj       enddo     enddo   enddo enddo send(matrix D) </pre>
--	---

TAB. 2.5 – Solution 4 : codes pour les tâches 1 et 2.

<pre> /* tâche 3 (F = F + GD) */ recv(matrice D) do j_L = 0, N/L   do i = 0, M     do j = j_L * L, (j_L + 1) * L - 1       do h = 0, K         F_ij = F_ij + G_ih D_hj       enddo     enddo   enddo send (sous-matrice F) enddo </pre>	<pre> /* tâche 4 (H = H + FI) */ do j_L = 0, N/L   recv(sous-matrice F)   do i = 0, M     do j = j_L * L, (j_L + 1) * L - 1       do h = 0, K         H_ij = H_ij + F_ih I_hj       enddo     enddo   enddo send (sous-matrice H) enddo </pre>	<pre> /* tâche 5 (J = J + HK) */ do k_L = 0, K/L   recv(sous-matrice H)   do i = 0, M     do j = 0, N       do k = k_L * L, (k_L + 1) * L - 1         J_ij = J_ij + H_ih K_hj       enddo     enddo   enddo enddo </pre>
---	--	--

TAB. 2.6 – Solution 4 : codes pour les tâches 3, 4 et 5.

cible ou il peut être estimé dynamiquement à l'exécution en fonction des temps d'attente. Dans cette section, je présente les travaux effectués en collaboration avec P. Ramet et J. Roman.

## 2.6.2 Travaux précédents

De nombreux articles ont abordé ce problème de manière statique ou dynamique que ce soit dans le cadre de parallélisation d'applications par passage de messages ou de la compilation de langages data-parallèles.

Les pipelines ont été modélisés à plusieurs reprises. Dans [172, 173], ils sont modélisés à l'aide de réseaux de Pétri modifiés afin d'évaluer les performances et de calculer le grain. Les articles cités dans la suite de cette section utilisent des modélisations plus classiques avec un calcul de complexité du code pipeline et ensuite un calcul du grain afin de minimiser le temps total d'exécution comme par exemple dans [82] pour le cas d'un pipeline général.

Le calcul statique du grain d'un pipeline nécessite une bonne connaissance de l'architecture cible et de l'algorithme parallélisé et donc leur modélisation précise. Un certain nombre d'articles utilisent cette technique [83, 198] et en particulier, cette option est souvent choisie pour l'optimisation de compilateurs parallélisateurs. C'est le cas de PARADIGM [139, 217, 218] ou Fortrand [257]. La principale difficulté est l'évaluation du coût des calculs et des communications. Pour ces deux compilateurs, les macro-pipelines sont mono-dimensionnels et le calcul du grain n'est donné que pour des cas triviaux et avec une modélisation sommaire. Malgré tout, on trouve dans [218] une modélisation de l'enchaînement de deux pipelines. Le calcul de la taille

des tuiles dans la technique du *tiling* est généralement statique. Dans [12, 11, 13], on trouve le calcul de taille de tuiles pour le *tiling* orthogonal à respectivement deux, trois et  $n$  dimensions. La solution pour trouver le grain du *tiling* revient à résoudre un problème non-linéaire de programmation entière. La taille et la forme des tuiles sont des variables et la fonction objectif est la minimisation du temps total d'exécution. Un algorithme en  $O(n \log n)$  est donné dans le cas d'un espace à  $n$  dimensions. Les programmes sont modélisés à l'aide d'équations récurrentes. Des résultats concernant le *tiling* à deux dimensions sont donnés dans [53]. Une modélisation précise de techniques macro-pipelines est donnée également dans [178] pour la parallélisation d'algorithmes numériques sur un hypercube. Les possibilités de communications  $n$ -ports sont prises en compte dans le calcul du grain. Je reviendrai sur cette technique dans la section 2.8.

Le calcul du grain de manière dynamique est avantageux à plus d'un titre car il permet dans un premier temps d'éviter une modélisation trop fastidieuse de l'architecture cible. En effet, l'apparition de processeurs aux caractéristiques de plus en plus compliquées (pipelines super-scalaires, unités arithmétiques parallèles, mémoires caches à plusieurs niveaux, etc.) empêche une modélisation d'une précision suffisante. Dans un second temps, il permet de s'adapter dynamiquement à la variation de charge des machines et des réseaux. Enfin, si le volume de calcul et de communications varie suivant les données, comme dans le cas de la parallélisation d'algorithmes mettant en œuvre des matrices creuses, cette évaluation doit souvent être effectuée à l'exécution.

Dans [190] puis dans [189], une optimisation du calcul du grain est effectuée à l'exécution qui consiste à prendre des mesures de performances sur les deux premières itérations du pipeline afin d'en déduire le meilleur grain. Deux itérations sont testées pour tenir compte des effets de cache. Ensuite une taille initiale de bloc est choisie. Une recherche des blocs causant des temps d'attente est effectuée et les blocs trouvés sont redécoupés de manière récursive. Enfin, les messages excédentaires sont supprimés en ré-exécutant l'algorithme sur les blocs adjacents qui ne sont pas redécoupés. L'ordonnancement des calculs et des messages est distribué par le nœud 0 à tous les autres. La distribution des données n'est également effectuée qu'après la troisième itération.

Le même type d'optimisation est présenté dans [240, 241] pour la parallélisation de boucles DOACROSS. L'équilibrage dynamique est réalisé par un schéma de type maître-esclave où le grain évolue en fonction de la charge des machines sur lesquelles le pipeline est exécuté. Ici la charge est constamment régulée au cours du calcul et l'un des problèmes consiste à éviter l'effet *ping-pong* entre les différents processeurs. Une modélisation assez précise des gains provenant de l'équilibrage est donnée.

### 2.6.3 Bibliothèque OPIUM

La figure 2.17 présente un pipeline entre deux processeurs. Le premier doit effectuer un calcul **avant** de communiquer (appelé par la suite « calcul avant ») et le second doit effectuer un calcul **après** avoir reçu le message de son prédécesseur (appelé par la suite « calcul après »). Le polynôme qui modélise le calcul avant est  $T_{comp_A}$  et le polynôme qui modélise le calcul après est  $T_{comp_B}$ . Chaque processeur peut avoir un calcul avant, un calcul après ou bien les deux.

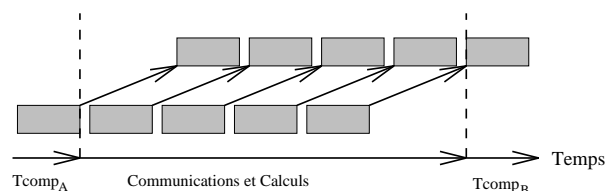


FIG. 2.17 – Schéma pipeline entre 2 processeurs.

Notre but consiste donc, à partir des paramètres de la machine (vitesse des processeurs, latence et bande passante des communications) et des paramètres de l'algorithme (schéma de communication, complexité des fonctions de calcul avant et après l'envoi de données), de calculer le grain de calcul optimal qui donnera le temps d'exécution minimal. Comme nous l'avons vu dans la section précédente, la plupart des auteurs (moi y compris durant ma thèse [84]) ont supposé que les communications et les calculs avaient des complexités qui étaient des fonctions linéaires de la taille des données. Cela n'est pas toujours le cas et suivant les applications on peut avoir des fonctions polynomiales quelconques et logarithmiques (voire des mélanges de différentes complexités entre le calcul avant et le calcul après). Nous avons donc donné des méthodes de calculs des tailles de paquets optimales pour ces fonctions de complexité en utilisant parfois des schémas de résolution numérique de type Newton ou Brent pour les modèles non-linéaires par exemple. Ces solutions nous permettent de traiter des cas plus compliqués et de raffiner les fonctions de complexité (par exemple pour tenir compte d'effets de caches ou d'algorithmes par blocs). Ensuite, nous avons développé une heuristique pour éviter de recalculer la taille de paquet à chaque itération lorsque le volume de calcul et de communication varie peu.

Une fois un modèle théorique et des algorithmes de résolution obtenus, il nous a fallu développer un logiciel pour pouvoir intégrer ces résultats dans nos propres développements. La bibliothèque OPIUM (Optimal Packet size compUtation Methods) a donc été développée en collaboration avec P. Ramet et J. Roman au LaBRI. La figure 2.18 montre les interfaces de programmation d'OPIUM pour le cas linéaire et le cas générique. Remarquez que ces fonctions sont indépendantes des machines cibles sur lesquelles elles sont utilisées. Dans le cas linéaire, il faut donner la fonction décrivant la constante de complexité (qui peut dépendre de la taille de paquet optimale), à la fois pour le calcul avant et le calcul après. Pour cela, nous utilisons la structure de donnée `comp_linear`. Les paramètres du modèle de communication sont stockés quant à eux dans la structure `comm_linear`. La fonction `Opium_Init_linear` doit être appelée pour initialiser ces différents paramètres et effectuer quelques pré-calculs. Ensuite, nous appelons `Opium_linear` pour calculer la taille optimale de paquet pour une taille de données à pipeliner donnée. Le cas générique utilise le même schéma, mais les fonctions décrivent les complexités des calculs et du modèle de communication ainsi que leurs dérivées. L'utilisateur remplit les structures de données, appelle `Opium_Init_Generic` puis, lorsque c'est nécessaire, `Opium_Generic`. Les paramètres de vitesse de calcul et de communication peuvent être calculés au préalable (ou au départ de l'exécution) à l'aide de benchmarks et fournis sous forme de constantes une fois pour toutes. Les fonctions de complexité des calculs et des communications peuvent également être évalués automatiquement par un compilateur paralléliseur [139].

Nous avons appliqué ces résultats à la factorisation  $LU$  distribuée de manière cyclique par colonnes [99] et au SOR [230]. Le chapitre C du document annexe présente les algorithmes que nous avons utilisés et nos résultats sur la factorisation  $LU$ . Les figures 2.19 et 2.20 présentent les résultats des pipelines dans la factorisation  $LU$  sur une SP2 d'IBM avec 16 processeurs. Nous utilisons ici les BLAS d'ESSL comme noyaux de calculs. Les figures 2.21 et 2.22 présentent les résultats des pipelines dans l'algorithme SOR sur une SP2 d'IBM avec 16 processeurs. Dans les deux cas, et pour des schémas de calculs et de communications différents, on voit clairement l'intérêt des macro-pipelines et la précision de notre calcul de taille optimale de paquet.

Pierre Ramet a étendu ces résultats ensuite au cas où le volume de données à calculer (et à communiquer) diminue au cours du calcul comme par exemple lors de la factorisation d'une matrice symétrique définie positive avec Cholesky. Il a montré que l'on pouvait calculer une suite de tailles de paquets optimale [229].

<pre> Cas linéaire : <b>struct</b> comp_linear {   <b>double</b> (*coeff_fct)(<b>int</b>),   <b>double</b> tau_comp;   <b>flag</b> flag_same_func;   <b>flag</b> flag_recalc; }  <b>struct</b> comm_linear {   <b>double</b> startup;   <b>double</b> bandwidth; }  Opium_Init_Linear( <b>struct</b> comp_linear comp_fwd, <b>struct</b> comp_linear comp_bwd, <b>struct</b> comm_linear comm);  Opium_Linear(<b>int</b> pipeline_size); </pre>	<pre> Cas générique : <b>struct</b> comp_generic {   <b>double</b> (*coeff_fct)(<b>int</b>);   <b>double</b> tau_comp;   <b>double</b> (*complexity_fct)(<b>double</b>);   <b>double</b> (*complexity_derivate_fct)(<b>double</b>);   <b>flag</b> flag_same_func;   <b>flag</b> flag_recalc; }  <b>struct</b> comm_generic {   <b>double</b> (*comm_fct)(<b>double</b>);   <b>double</b> (*comm_derivate_fct)(<b>double</b>); }  Opium_Init_Generic( <b>struct</b> comp_generic comp_fwd, <b>struct</b> comp_generic comp_bwd, <b>struct</b> comm_generic comm, <b>double</b> precision);  Opium_Generic(<b>int</b> pipeline_size); </pre>
---	--

FIG. 2.18 – Interfaces OPIUM pour le cas linéaire et pour le cas générique.

## 2.7 Implantation des schémas macro-pipelines – Bibliothèque LOCCS

### 2.7.1 Amélioration de la bibliothèque

La bibliothèque LOCCS<sup>2</sup> a été initiée durant ma thèse. J’avais proposé à l’époque un ensemble de routines permettant d’effectuer des pipelines de calculs avec plusieurs schémas de communication sous-jacents (one-to-one, diffusion, scatter, gather, all-to-all, etc.).

Nous l’avons améliorée avec Pierre Ramet dans un premier temps en la portant au-dessus de MPI et ensuite en ajoutant de nouveaux schémas de communication pour pouvoir traiter des schémas de communications adaptés aux applications de type SOR ou ADI. Il s’agit en fait de chaînes de communications.

### 2.7.2 Utilisation dans un compilateur HPF

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

La première tâche d’un compilateur paralléliseur est donc de découvrir les schémas pipelines dans les boucles. Dans [257], on trouve un algorithme pour découvrir les boucles CROSS-PROCESSOR, c’est-à-dire les boucles dans lesquelles les itérations traversent les frontières des processeurs. Il faut également que le nid soit suffisamment profond pour que le grain de calcul soit intéressant. Il peut être nécessaire d’appliquer une permutation de boucle pour pouvoir exploiter un pipeline. Un algorithme plus général est donné dans [139] dans lequel une estimation du coût de la boucle est également donnée.

La seconde tâche du compilateur consiste à calculer le grain de calcul et de communication (avec les méthodes présentées précédemment). Enfin, le code doit être généré. Cette génération pour

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



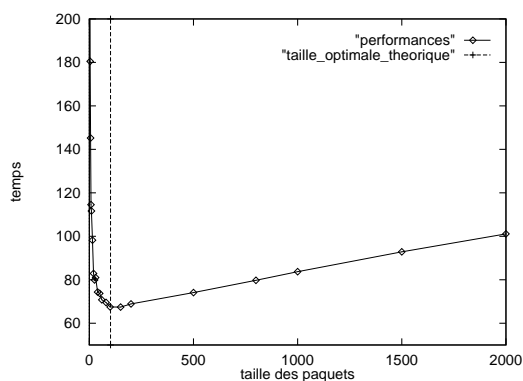


FIG. 2.19 – Temps de factorisation par bloc d'une matrice 3360x3360 en fonction de la taille des paquets (SP2, 16 procs).

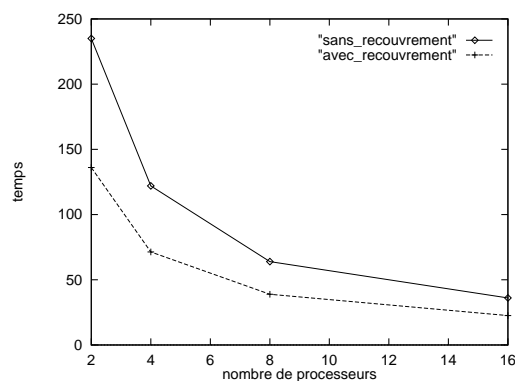


FIG. 2.20 – Temps de factorisation par bloc d'une matrice 3360x3360 (avec et sans recouvrement) en fonction du nombre de processeurs (SP2, 16 procs).

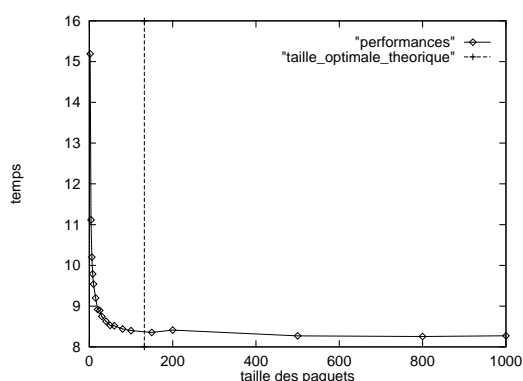


FIG. 2.21 – Temps du SOR pour une matrice 3360x3360 en fonction de la taille des paquets (SP2, 16 procs).

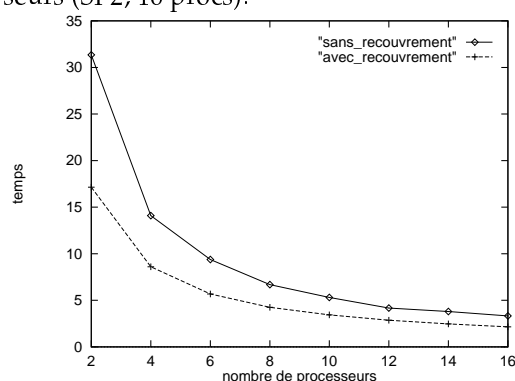


FIG. 2.22 – Temps du SOR pour une matrice 3360x3360 (avec et sans recouvrement) en fonction du nombre de processeurs (SP2, 16 procs).

un schéma macro-pipeline n'est pas aisée. Deux approches peuvent être utilisées : directement en générant le code du macro-pipeline ou en faisant appel à une bibliothèque. En prenant la première solution, on garde une plus grande flexibilité mais le code est fastidieux à générer et il est plus difficile d'ajouter des optimisations comme l'équilibrage dynamique. En générant un appel à une bibliothèque spécialisée, on garantit l'obtention d'un code pipeline performant et optimisé mais on se restreint à un nombre limité de schémas de communication.

La première approche a été utilisée dans le cadre des compilateurs FortranD [257] et PARADIGM [218] ainsi que dans le cadre du développement d'un outil de parallélisation automatique de codes Fortran, CAPTools [121]. Les pipelines utilisés ici sont mono-dimensionnels. Pour ce dernier outil, trois types de recouvrements sont générés : les recouvrements simples (sans dépendance entre les calculs et les communications), les recouvrements partiels (lorsque seule une partie d'une boucle empêche les recouvrements) et les recouvrements pipelines (mais uniquement en 1D dans cette version de l'outil CAPTools). Ces optimisations ont été validées avec succès sur deux codes des benchmarks NAS-PAR et PERFECT puisque le code généré automatiquement est aussi performant qu'un code généré à la main. Dans [241, 240] est décrit une technique d'équilibrage de charge dynamique pour la parallélisation de boucles DOACROSS. Il s'agit de pipeline à une dimension et l'originalité vient uniquement du calcul dynamique du grain.

Avec T. Brandes du GMD/SCAI, nous avons choisi d'utiliser une bibliothèque spécialisée et nous avons intégré la bibliothèque LOCCS au compilateur HPF Adaptor [47, 48]. L'utilisation de techniques macro-pipelines est toutefois laissée à la charge du programmeur mais la gestion des communications, des bufferisations et la gestion des distributions HPF est faite par le compilateur. La figure 2.23 montre la parallélisation du code ADI en HPF, soit en utilisant des directives REDISTRIBUTE (à gauche de la figure), soit en utilisant le driver pour les LOCCS DALIB\_LOCCS\_DRIVER (à droite de la figure). Dans la seconde solution, on fournit la routine `block` qui sera exécutée sur chacun des blocs du pipeline. Le code utilisant les LOCCS est plus compliqué à écrire en HPF mais par contre on obtient de bien meilleurs résultats et les techniques macro-pipelines sont cachées à l'intérieur de la bibliothèque. Ce code peut bien sûr être généré automatiquement (voir section 3.4.5 du chapitre 3). La figure 2.24 montre les facteurs d'accélération obtenus en utilisant les deux programmes. Le gain des pipelines est encore clair. Nous avons obtenu de très bonnes performances sur deux autres applications : Gauss-Seidel et le solveur Helmholtz utilisé pour la simulation météorologique [48].

---

```

PARAMETER (N=...)
REAL, DIMENSION (N, N) :: A, B
!HPF$ DISTRIBUTE (*, BLOCK) :: A, B
...
DO J = 1, N      ! Execution parallèle
  DO I = 2, N
    A(I, J) = A(I, J) - A(I-1, J)*B(I, J)
  END DO
END DO

!HPF$
REDISTRIBUTE (BLOCK, *) :: A, B
DO I = 1, N      ! Execution parallèle
  DO J = 2, N
    A(I, J) = A(I, J) - A(I, J-1)*B(I, J)
  END DO
END DO

REDISTRIBUTE (*, BLOCK) :: A, B

PARAMETER (N=...)
REAL, DIMENSION (N, N) :: A, B
!HPF$ DISTRIBUTE (*, BLOCK) :: A, B
...
DO I = 2, N
  DO J = 1, N
    A(I, J) = A(I, J) - A(I-1, J)*B(I, J)
  END DO
END DO
CALL DALIB_LOCCS_DRIVER(block,6,2,0,
                        A(:,2:N),[0,1],B(:,2:N),[0,0])
...
EXTRINSIC (HPF_LOCAL) SUBROUTINE block(A, B)
REAL A(:, :), B(:, :)
!HPF$ DISTRIBUTE (*, BLOCK) :: A, B
DO J=lbound(A,2),ubound(A,2)
  DO I=lbound(A,1),ubound(A,1)
    A(I, J) = A(I, J) - A(I, J-1)*B(I, J)
  END DO
END DO
END SUBROUTINE block

```

---

FIG. 2.23 – Parallélisation de l'ADI avec redistribution ou utilisation des LOCCS en HPF.

## 2.8 Tiling

Comme nous l'avons vu dans une section précédente, toutes ces techniques sont proches de celle du *tiling* qui consiste à transformer un nid de boucle en modifiant le grain des itérations soit pour améliorer l'utilisation des mémoires hiérarchiques ou pour augmenter le parallélisme d'un code [160, 226, 268]. Dans la technique du *tiling*, deux paramètres sont étudiés, la forme des tuiles et leur taille (ou volume). Dans les problèmes auxquels nous nous intéressons, nous ne regardons pas la forme des tuiles car les dépendances entre les itérations sont parallèles ou orthogonales aux espaces d'itération et nous ne divisons qu'une seule des dimensions. Par contre, les problèmes liés au calcul de la taille des tuiles nous concernent directement. Parmi les travaux les plus récents sur ce sujet, on peut citer les travaux effectués à Rennes et à Valenciennes autour du *tiling* orthogonal multi-dimensionnel [11, 12, 13].

Avec F. Rastello et Y. Robert [89], nous avons étendu les résultats d'Högsted, Carter et Ferrante [145] en étudiant les temps d'inactivité lorsque les espaces d'itération sont des parallélogrammes ou des trapèzes quelconques. Nous avons donné une formulation pour toutes

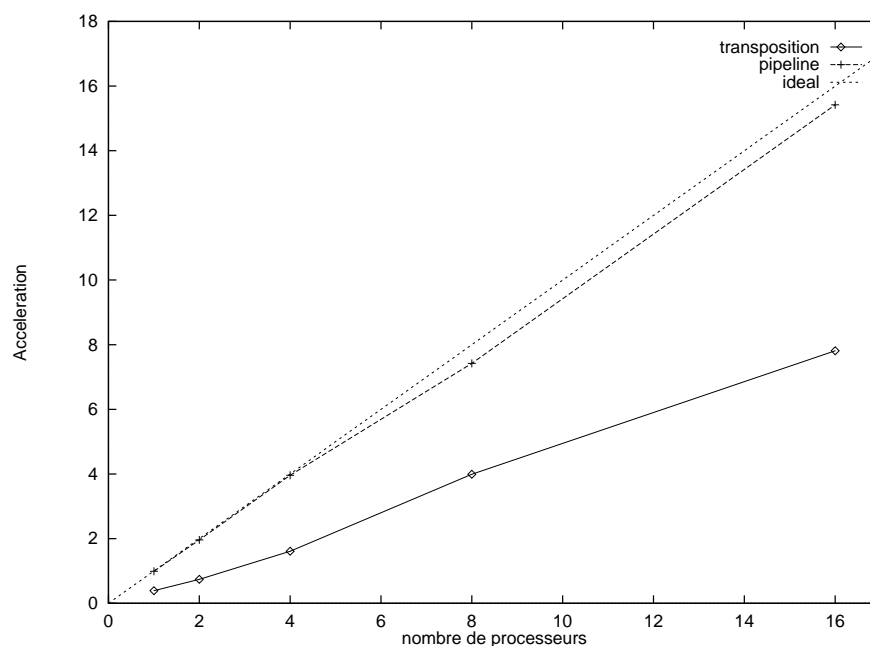


FIG. 2.24 – Facteur d'accélération de l'ADI avec redistribution ou utilisation des LOCCS en HPF.

valeurs des paramètres de pentes pour des distributions blocs et cycliques. Cependant, ces résultats sont limités à des distributions mono-dimensionnelles.

## 2.9 Conclusion et perspectives

Nous avons donc présenté un tour d'horizon assez complet des techniques de recouvrements calculs/communications et macro-pipelines sur différents types d'architectures et pour différentes applications. Ces techniques sont maintenant éprouvées et font partie des optimisations classiques des compilateurs parallélisateurs ou des supports d'exécution. Il reste cependant des travaux à effectuer et ceci dans plusieurs directions.

Nous étudions maintenant le cas où notre machine cible est hétérogène. Dans un premier temps, nous supposons que notre machine est constituée de cartes multiprocesseurs connectées par un réseau rapide. Dans ce type de machine, les communications à l'intérieur d'un nœud de la grappe se font grâce à la mémoire partagée et sont donc plus rapides que les communications entre cartes.

Une voie de recherche qui a été peu évaluée est l'étude conjointe de l'optimisation des (re)distributions et des macro-pipelines. En effet, les travaux précédents concernant le calcul des distributions (voir section 3.3 du chapitre 3) ne supposaient jamais la possibilité de pipeliner les calculs avec un grain variable. Nous avons utilisé conjointement une optimisation de la distribution liée à un macro-pipeline dans le cas de la FFT bidimensionnelle [55] et comparé les redistributions avec l'utilisation de macro-pipelines pour l'ADI [48]. Cependant, il serait intéressant d'étudier des algorithmes de distribution automatique de données supposant que des pipelines peuvent être utilisés.

Les techniques de macro-pipelines multi-dimensionnels restent encore du domaine théorique même si nous présentons quelques résultats dans ce chapitre avec le Sweep3D. Comme nous l'avons vu précédemment (section 2.4), il faut toutefois être sûr que le gain qui pourra être obtenu vaut le surcoût en terme de développement.

En ce qui concerne les optimisations à base de *tiling*, la difficulté de leur implémentation (surtout si elles sont intégrées à un compilateur paralléliseur) les rendent encore peu abordables de manière pratique. En plus de problèmes de choix de stratégies de *tiling* (forme et taille des tuiles, ordonnancement de leur calcul) se pose le délicat problème de la génération de code.

Les travaux autour de la minimisation des entrées/sorties sont encore sommaires [74], surtout si l'on veut tenir compte de l'algorithme. Cependant, l'utilisation d'entrées sorties parallèles ne fait qu'ajouter un niveau à la hiérarchie mémoire et les mêmes techniques peuvent être appliquées. Nous allons étudier ce type de techniques en collaboration avec G. Utard du LARIA.

En ce qui concerne les architectures fortement hétérogènes, les travaux actuels sont encore assez limités. Malgré tout, ce domaine est particulièrement important au vu de l'évolution actuelle des architectures. Une gestion dynamique des pipelines et de leur grain est indispensable si l'on veut obtenir les meilleures performances. Une approche mettant en œuvre conjointement des algorithmes efficaces et un support d'exécution multi-threads puissant permettant d'affiner l'ordonnancement des calculs devrait donner de bons résultats.

On constate donc que le sujet n'est pas clos, même si les architectures actuelles donnent plus d'importance au système et au support d'exécution qu'auparavant.

# Chapitre 3

## TransTool/ALASca ou du Fortran 77 aux machines parallèles à mémoire distribuée

Ce chapitre présente mes travaux autour de la parallélisation de programmes Fortran vers High Performance Fortran et le développement d'outils de parallélisation d'applications numériques, TRANSTOOL et ALASCA.

Ces travaux s'échelonnent d'Avril 1995 à Septembre 1999.

*Travaux effectués en collaboration avec S. Domas, J.J. Dongarra, J.-C. Mignot, A. Petitet, C. Randriamaro et Y. Robert*



### 3.1 Introduction

Dès les premiers balbutiements du parallélisme, de nombreux chercheurs se sont penchés sur une question délicate : est-il possible de transformer automatiquement un code séquentiel en code parallèle ? Cette approche du parallélisme a donc soulevé de nombreuses recherches passionnantes et a conduit à la mise à jour de nombreux problèmes NP-difficiles. Certes des heuristiques peuvent être trouvées mais on est loin des « boîtes noires » promises au début du parallélisme. De nombreux chercheurs ont montré que la parallélisation de codes généraux et ceci de manière totalement automatique était excessivement difficile, voire impossible. Reste donc à l'utilisateur de mettre la « main à la pâte » et d'aider le compilateur à paralléliser son code.

Sur une machine parallèle à mémoire distribuée, le placement des données sur les processeurs est de première importance. En effet, c'est ce placement qui conditionne à la fois l'efficacité de la parallélisation et la réduction du surcoût des communications.

High Performance Fortran [175] suppose un schéma de placement des données à 2 niveaux (figure 3.1). Tout d'abord, les données des différents tableaux utilisés dans les calculs sont alignés entre eux (et sur un template, une sorte de tableau virtuel), puis les templates sont distribués sur des grilles virtuelles de processeurs. Les distributions utilisées par HPF sont classiques : cycliques, blocs et bloc-cycliques. Certaines bibliothèques parallèles (comme ScaLAPACK [37]) utilisent également ce type de distributions. Différents exemples de distributions cycliques sont donnés dans la figure 3.2.

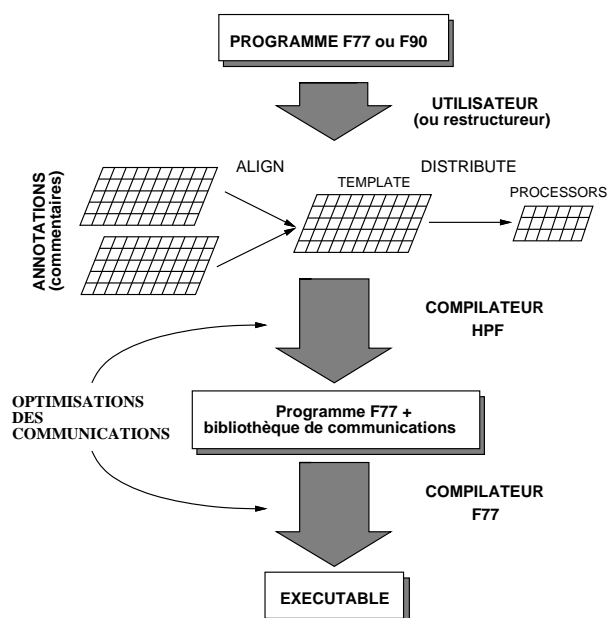


FIG. 3.1 – Placement de données à deux niveaux dans HPF.

Ce type de distribution équilibre de manière assez efficace les calculs et les communications dans la plupart des programmes numériques utilisant des structures de données régulières si, bien sûr, l'on choisit de manière adéquate la taille des blocs pour optimiser l'utilisation des différents niveaux de caches et minimiser les communications.

L'une des premières opérations à effectuer lors de la parallélisation d'une application numérique consiste donc à bien choisir la distribution initiale des données. Ce choix peut être automatisé (partiellement ou totalement). Par contre, il se peut qu'une distribution choisie en début de programme ne soit pas adaptée tout au long de son exécution. On peut alors avoir besoin d'effectuer une redistribution des données. Cette redistribution est une routine de communication

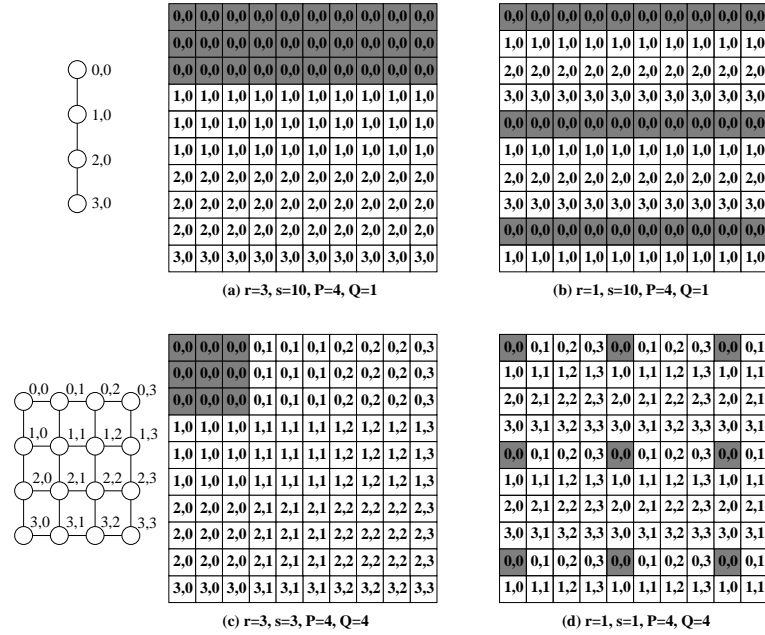


FIG. 3.2 – Distributions de matrices cycliques, blocs et blocs cycliques.

globale qui représente un surcoût qui va tendre à réduire l'efficacité du programme si elle n'est pas correctement utilisée (ou implémentée).

## 3.2 Optimisation des redistributions

L'une première optimisation consiste donc à optimiser les redistributions elles-mêmes afin qu'elles pèsent moins sur le surcoût total du programme. Une redistribution comprend deux phases principales : la création des messages puis leur envoi. L'ordonnancement des messages (voire leur découpage ou leur agrégation) doit être optimisé de manière importante car comme nous le verrons dans la section 3.2.3, une redistribution ne se ramène pas toujours à un échange total.

### 3.2.1 Formulation du problème

Considérons un vecteur  $X[0..M-1]$  de taille  $M$  distribué sur une ligne de  $P$  processeurs (numérotés de  $p = 0$  à  $p = P - 1$ ) suivant une distribution  $CYCLIC(r)$ . Notre but est de redistribuer  $X$  dans un autre vecteur  $Y$  distribué sur une ligne de  $Q$  processeurs (numérotés de  $q = 0$  à  $q = Q - 1$ ) suivant une distribution  $CYCLIC(s)$ . Soit  $L = \text{ppcm}(P \times r, Q \times s)$ , le plus petit multiple commun de  $P \times r$  et  $Q \times s$ . Le schéma de redistribution se répétant après chaque segment de  $L$  éléments, nous considérons, pour simplifier le problème, que la taille  $M$  de  $X$  est un multiple de  $L$ . En supposant que  $X$  est constitué d'un nombre entier de segments, nous pouvons éviter toute discussion sur les effets de bord sans perdre de généralisation. Nous noterons  $m = M \div L$  ce nombre de segments.



## Notation

On note  $(P, r) \rightarrow (Q, s)$  la redistribution d'un vecteur à partir d'une grille source de  $P$  processeurs avec une distribution CYCLIC( $r$ ) vers une grille destination de  $Q$  processeurs avec une distribution CYCLIC( $s$ ), en considérant que seul un segment de  $L = \text{ppcm}(P \times r, Q \times s)$  éléments doivent être redistribués.

Ainsi, toutes les tailles de messages présentées dans cette section doivent être vues comme des unités de tailles (qu'il faudra multiplier par  $m$ , le nombre de segments pour considérer tout le vecteur).

### DÉFINITION 1 Grille de communication

Pour une redistribution  $(P, r) \rightarrow (Q, s)$ , on associe une matrice  $E = (e_{pq})_{\substack{0 \leq p < P \\ 0 \leq q < Q}}$  de taille  $P \times Q$ , appelée grille de communication, représentant l'ensemble des communications point à point à effectuer pour obtenir la redistribution. Chaque élément  $e_{pq}$  de la matrice indique la taille du message que le processeur  $p$  doit envoyer au processeur  $q$ .

Notons que les deux grilles de processeurs peuvent avoir des tailles différentes : un même indice sur chacune des deux grilles peut correspondre à deux processeurs physiques différents. Ainsi, le véritable nombre de processeurs utilisés est une valeur inconnue, comprise entre le maximum de  $(P, Q)$  et  $P + Q$ .

### DÉFINITION 2 Échange total

Un échange total est une redistribution pour laquelle la grille de communication est pleine.

### DÉFINITION 3 Redistribution

Pour une redistribution  $(P, r) \rightarrow (Q, s)$  et la grille de communication associée, une redistribution est la matrice d'étiquettes  $O$  de taille  $P \times Q$ , qui indique, pour chaque couple  $(p, q)$ , à quelle étape se fera la communication de  $p$  vers  $q$ .

### DÉFINITION 4 Ordonnancement synchrone

On dit que l'ordonnancement est synchrone lorsque la matrice d'étiquettes  $O$  est telle que, pour chaque ligne (ou colonne), une étiquette apparaisse au plus une fois.

Dans ce cas, à chaque étape (ou étiquette), un processeur ne peut envoyer et recevoir plus d'un message.

Un ordonnancement est constitué de plusieurs étapes successives dans lesquelles chaque émetteur ne doit pas envoyer plus d'un message ; de façon symétrique, chaque récepteur ne doit pas recevoir plus d'un message. Considérons une redistribution  $(P, r) \rightarrow (Q, s)$ .

- La **grille de communication** est un tableau  $P \times Q$  dont chaque entrée  $\text{taille}(p, q)$  à la position  $(p, q)$  est non nulle si et seulement si  $p$  a un message à envoyer à  $q$ .
- Une **étape de communication** est un ensemble de paires  $\{(p_1, q_1), (p_2, q_2), \dots, (p_t, q_t)\}$  telles que  $p_i \neq p_j$  pour  $1 \leq i < j \leq t$ ,  $q_i \neq q_j$  pour  $1 \leq i < j \leq t$ , et  $\text{taille}(p_i, q_i) > 0$  pour  $1 \leq i \leq t$ . Si  $t = \min(P, Q)$ , c'est-à-dire que tous les émetteurs et tous les récepteurs sont actifs, alors l'étape de communication est dite **complète**, sinon, elle est incomplète. Le coût d'une étape de communication est la valeur maximale de ses entrées, en d'autres termes,  $\max\{\text{taille}(p_i, q_i); 1 \leq i \leq t\}$ .

Un **ordonnancement** est une succession d'étapes de communication telle que chaque entrée de la grille de communication apparaît dans une et une seule étape. Le coût d'un ordonnancement peut être évalué de deux façons différentes :

1. soit le **nombre d'étapes**  $NE$ , qui est tout simplement le nombre d'étapes de communications dans l'ordonnancement,
2. soit le **coût total**  $CT$ , qui est la somme des coûts de chaque étape de communication (comme défini précédemment).

Lors de la planification de l'ordonnement, notre objectif est donc double :

- minimiser le nombre d'étapes de l'ordonnement,
- minimiser le coût total de l'ordonnement.

La grille de communication, illustrée par les tableaux de la section suivante, reprend la taille des communications pour un seul segment du vecteur, c'est-à-dire un vecteur dont la taille est  $L = \text{ppcm}(P \times r, Q \times s)$ . La motivation de l'évaluation des ordonnements par leur nombre d'étapes ou via leur coût total est la suivante :

- le nombre d'étapes  $NE$  est le nombre de synchronisations nécessaires pour exécuter l'ordonnement. Si l'on estime grossièrement comme une mesure unitaire chaque étape de communication incluant tous les processeurs, le nombre d'étapes est alors une bonne évaluation du coût des redistributions,
- on pourrait essayer d'être plus précis. À chaque étape, plusieurs messages de différentes tailles sont échangés ; le temps d'exécution d'une étape est dépendant du plus long de ses messages. Un modèle simple pourrait considérer que le coût d'un message est  $\beta + \max\{\text{taille}(p_i, q_i)\} \times \tau$  où  $\beta$  est la latence, et  $\tau$  l'inverse de la bande passante d'un lien physique de communication.

Bien que cette expression ne tienne pas compte des points critiques et des contentions du réseau, elle s'est avérée être très utile sur plusieurs machines [71, 133]. Suivant ces formules, le coût d'une redistribution est l'expression affine  $\beta \times NE + \tau \times CT$  qui motive notre intérêt aussi bien par le nombre d'étapes que par le coût total.

### 3.2.2 État de l'art

Dans cette section, nous présentons un tour d'horizon des travaux précédents autour de la redistribution de matrices distribuées de manière cyclique par blocs. Nous donnons une attention toute particulière aux travaux de Walker et Otto [263].

#### 3.2.2.1 Calcul des communications

L'une des premières étapes réalisée par un compilateur ou par une bibliothèque de redistribution de données consiste à calculer les éléments qui doivent être émis (et reçus) et également les processeurs expéditeurs et destinataires de ces messages. Nous appelons cette étape le *calcul des communications*.

La plupart de ces travaux sont issus des recherches autour de la compilation d'HPF. Plusieurs publications traitent du problème de génération d'un code efficace pour les instructions d'affectation de vecteurs en HPF (comme  $A[l_1 : u_1 : s_1] = B[l_2 : u_2 : s_2]$ , où  $A$  et  $B$  sont tous deux des tableaux distribués de manière cyclique par blocs sur une ligne de processeurs). Certains chercheurs (Stichnoth et al. [249], van Reeuwijk et al. [259], et Wakatani et Wolfe [262], etc.) se sont principalement préoccupés de cas dans lesquels les vecteurs sont distribués en utilisant une diffusion pure, une distribution cyclique ( $\text{CYCLIC}(1)$  en HPF) ou une distribution par blocs ( $\text{CYCLIC}(\lfloor \frac{n}{p} \rfloor)$ ), où  $n$  est la taille du vecteur et  $p$  le nombre de processeurs).

Par la suite, plusieurs algorithmes ont été publiés qui traitent du cas général de distributions bloc-cycliques  $\text{CYCLIC}(k)$  en utilisant des techniques sophistiquées comme des automates d'états finis [64], des méthodes utilisant la théorie des ensembles [140], des équations diophantiennes [169, 168], les formes de Hermite et les treillis [253] ou la programmation linéaire [8]. Une étude comparative de ces algorithmes est présentée dans l'article de Wang et al. [264], où il est rapporté que, grâce à ces algorithmes, on peut manipuler les distributions bloc-cycliques aussi efficacement que les cas plus simples comme la distribution cyclique ou par blocs.

À la fin de la phase de génération des messages, chaque processeur a calculé plusieurs messages différents, généralement stockés dans une mémoire temporaire. Ces messages doivent être ensuite envoyés à un ensemble de processeurs qui les recevront. De façon symétrique,

chaque processeur calcule le nombre et la taille des messages qu'il devra recevoir pour pouvoir réserver l'espace mémoire correspondant. En résumé, lorsque la phase de génération des messages est achevée, chaque processeur a préparé un message pour tous les processeurs à qui il doit envoyer des données, et chaque processeur possède toutes les informations concernant les messages qu'il recevra (nombre, taille et origine). Il « ne reste alors plus qu'à » ordonnancer ces communications.

### 3.2.2.2 Ordonnancement des communications

Jusqu'à la fin des années 90, peu d'attention avait été portée sur l'ordonnancement des communications induites par l'opération de redistribution. Des stratégies simples ont été élaborées. Par exemple, Kalns et Ni [163] considèrent les communications comme un échange total entre tous les processeurs sans spécifier davantage l'opération. Dans leur étude comparative, Wang et al. [264] utilisent le schéma suivant pour effectuer l'affectation entre vecteurs : après le calcul des échanges, chaque processeur envoie tous ses messages de façon asynchrone, puis il réceptionne ceux qui lui sont destinés. Même si la phase de communication est décrite plus précisément, on peut noter qu'il n'y a pas d'ordonnancement explicite. Cette approche requiert de grandes capacités en termes de tampons de communication. De plus, des inter-blocages peuvent apparaître lors de redistribution de gros vecteurs. Il n'existe pas à ma connaissance de bibliothèque de passage de messages qui puisse supporter un tel bombardement de communications.

La bibliothèque ScaLAPACK [37] fournit un ensemble de fonctions qui permet d'effectuer la redistribution de tableaux. Dans l'algorithme de Prylli et Tourancheau [223], un échange total est organisé entre tous les processeurs qui sont rangés en chenille (virtuelle). L'échange total est programmé comme une succession d'étapes. À chaque étape, les processeurs sont arrangés en paires et effectuent simultanément un envoi et une réception. Ensuite, la chenille effectue un décalage, et il se forme un nouvel ensemble de paires de processeurs. Là encore, même si une attention particulière est portée à la programmation de l'échange total, rien ne concerne l'exploitation du fait que certaines paires de processeurs peuvent ne rien avoir à communiquer. À chaque étape  $k \in [1 \dots P]$ , le processeur  $p$  échange ses données avec le processeur  $(P - p - k) \bmod P$ .

Le premier article qui traite de l'ordonnancement des communications induites par une redistribution est celui de Walker et Otto [263]. Les auteurs introduisent un algorithme pour la programmation des communications d'une redistribution : l'ordonnancement synchrone. Il s'agit d'un algorithme synchrone composé de phases, ou étapes de communication. À chaque étape, chaque processeur peut envoyer ou recevoir au plus un message. Leur algorithme est restreint au cas de redistribution d'une matrice de la distribution  $CYCLIC(r)$  sur une grille de  $P$  processeurs, vers une distribution  $CYCLIC(K \times r)$  sur une grille de même taille. L'ordonnancement obtenu est composé de  $K$  étapes. Les auteurs montrent que lorsque  $K$  est inférieur à  $P$ , le temps de communication est nettement réduit.

Dans cette section, je présente nos résultats sur la généralisation aux redistributions quelconques : une distribution  $CYCLIC(r)$  sur  $P$  processeurs vers une distribution  $CYCLIC(s)$  sur une grille de  $Q$  processeurs. Nous retiendrons cette idée originale : ordonnancer les communications en étapes. À chaque étape, chaque processeur participant à l'échange n'envoie ni ne reçoit plus d'un message pour éliminer les points critiques et les contentions. Cette stratégie est bien adaptée aux architectures réseau actuelles de types commutateurs.

### 3.2.3 Quelques exemples

Nous allons dans un premier temps examiner quelques exemples significatifs de divers schémas de redistributions qui nous ont permis de vérifier la validité de nos algorithmes.

Nous étudions ici cinq exemples différents qui permettent de visualiser les principaux cas de figures rencontrés pour ordonnancer les messages lors de redistributions. Les quatre premiers

exemples sont utilisés pour la redistribution sans entrelacement (section 3.2.5) et le dernier exemple est utilisé pour la redistribution avec entrelacement des messages (section 3.2.6).

1. dans un premier exemple, les deux distributions ont la même grille de processeurs, chaque processeur envoie et reçoit la même quantité de données, sans qu'il y ait un échange total ; ce cas est illustré par la redistribution  $(16, 3) \rightarrow (16, 5)$ ,
2. ensuite, nous nous intéressons aux redistributions dans lesquelles nous avons échange total ; ce cas est illustré par la redistribution  $(16, 7) \rightarrow (16, 11)$ ,
3. notre troisième exemple est celui d'une répartition déséquilibrée des communications : tous les processeurs n'envoient ni reçoivent le même nombre d'éléments ; ce cas est illustré par la redistribution  $(15, 3) \rightarrow (15, 5)$ , nous notons ici qu'un changement du nombre de processeurs suffit à déséquilibrer les communications,
4. dans le quatrième exemple, nous évoquons le cas des redistributions dans lesquelles le nombre de processeurs change ; ce cas est illustré par la redistribution  $(12, 4) \rightarrow (8, 3)$ ,
5. enfin, le dernier exemple (redistribution  $(6, 2) \rightarrow (6, 3)$ ) présente un cas favorable à la redistribution avec entrelacement des messages (section 3.2.6).

**Redistribution régulière** Considérons un premier exemple avec  $P = Q = 16$  processeurs,  $r = 3$  éléments par bloc et  $s = 5$  éléments par bloc. Toutes les communications sont résumées dans le tableau 3.1, que l'on appellera *la grille de communication*. Notons aussi que les grilles de processeurs source et destination sont interprétées comme des grilles disjointes (même si ce n'est pas nécessairement le cas). Les abréviations « N.M. » et « E/R » signifient respectivement « nombre de messages » et « émetteur/récepteur ».

Nous remarquons ici que chaque processeur source  $p \in \mathcal{P} = \{0, 1, \dots, P - 1\}$  envoie 7 messages et que chaque processeur destination  $q \in \mathcal{Q} = \{0, 1, \dots, Q - 1\}$  en reçoit également 7. Ainsi, le schéma de communication ne correspond pas à un échange total : un échange total entre tous les processeurs demanderait 16 étapes de communication, avec un total de 16 messages à envoyer par processeur, ou plus précisément 15 messages et une copie locale. Il serait donc préférable d'ordonner les communications de façon plus efficace. L'idéal serait de pouvoir organiser la redistribution en 7 étapes que l'on appellera phases de communication. À chaque étape, on exécuterait un échange de messages sur 16 paires de processeurs disjointes.

Notons que pour cet exemple, nous pouvons encore optimiser les communications : nous pouvons essayer d'organiser les étapes de communication en nous arrangeant pour que les 8 paires de processeurs échangent un message de même taille à chaque étape. Le temps d'exécution de chaque étape est défini par la taille du plus long message échangé durant l'étape. On remarque que la taille des messages peut varier de façon plus ou moins significative. Les nombres de messages du tableau 3.1 varient de 1 à 3, ils correspondent à des nombres de segments de vecteurs. Par exemple, pour un vecteur  $X$  de taille  $M = 240000$ , on a  $m = 1000$  et la taille des messages varie de 1000 à 3000.

Dans le tableau 3.3, on calcule le coût de chaque étape de communication sur la base de la taille du plus long message impliqué dans cette étape. Le coût total de la redistribution est donc la somme des coûts de toutes les étapes de communication.

**Echange total** Le second exemple est caractérisé par  $P = Q = 16$  processeurs,  $r = 7$  éléments par bloc et  $s = 11$  éléments par bloc. Il montre l'intérêt d'un ordonnancement efficace, même lorsque chaque processeur communique avec tous les autres. La taille des messages varie avec un ratio de 2 à 7 comme l'illustre le tableau 3.4. Nous devons organiser les étapes d'échanges entre tous les processeurs de telle sorte que les messages communiqués à chaque étape soient de même taille.

E/R	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	N.M.
0	3	-	-	3	-	-	3	-	-	2	1	-	1	2	-	-	7
1	2	1	-	1	2	-	-	3	-	-	3	-	-	3	-	-	7
2	-	3	-	-	3	-	-	2	1	-	1	2	-	-	3	-	7
3	-	1	2	-	-	3	-	-	3	-	-	3	-	-	2	1	7
4	-	-	3	-	-	2	1	-	1	2	-	-	3	-	-	3	7
5	2	-	-	3	-	-	3	-	-	3	-	-	2	1	-	1	7
6	3	-	-	2	1	-	1	2	-	-	3	-	-	3	-	-	7
7	-	3	-	-	3	-	-	3	-	-	2	1	-	1	2	-	7
8	-	2	1	-	1	2	-	-	3	-	-	3	-	-	3	-	7
9	-	-	3	-	-	3	-	-	2	1	-	1	2	-	-	3	7
10	1	-	1	2	-	-	3	-	-	3	-	-	3	-	-	2	7
11	3	-	-	3	-	-	2	1	-	1	2	-	-	3	-	-	7
12	1	2	-	-	3	-	-	3	-	-	3	-	-	2	1	-	7
13	-	3	-	-	2	1	-	1	2	-	-	3	-	-	3	-	7
14	-	-	3	-	-	3	-	-	3	-	-	2	1	-	1	2	7
15	-	-	2	1	-	1	2	-	-	3	-	-	3	-	-	3	7
N.M.	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	

TAB. 3.1 – Grille de communication pour  $P = Q = 16$ ,  $r = 3$ , et  $s = 5$ . La taille des messages indiquée est calculée pour la redistribution d'un vecteur de taille  $L = 240$ .

E/R	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	b	-	-	a	-	-	c	-	-	e	g	-	f	d	-	-
1	d	f	-	g	e	-	-	c	-	-	b	-	-	a	-	-
2	-	b	-	-	c	-	-	e	g	-	f	d	-	-	a	-
3	-	g	e	-	-	a	-	-	b	-	-	c	-	-	d	f
4	-	-	c	-	-	e	g	-	f	d	-	-	b	-	-	a
5	e	-	-	c	-	-	a	-	-	b	-	-	d	f	-	g
6	c	-	-	e	g	-	f	d	-	-	a	-	-	b	-	-
7	-	c	-	-	b	-	-	a	-	-	d	f	-	g	e	-
8	-	e	g	-	f	d	-	-	c	-	-	a	-	-	b	-
9	-	-	a	-	-	b	-	-	d	f	-	g	e	-	-	c
10	g	-	f	d	-	-	b	-	-	c	-	-	a	-	-	e
11	a	-	-	b	-	-	d	f	-	g	e	-	-	c	-	-
12	f	d	-	-	a	-	-	b	-	-	c	-	-	e	g	-
13	-	a	-	-	d	f	-	g	e	-	-	b	-	-	c	-
14	-	-	b	-	-	c	-	-	a	-	-	e	g	-	f	d
15	-	-	d	f	-	g	e	-	-	a	-	-	c	-	-	b

TAB. 3.2 – Les étapes de communication pour  $P = Q = 16$ ,  $r = 3$ , et  $s = 5$ .

Etape	a	b	c	d	e	f	g	Total
Coût	3	3	3	2	2	1	1	15

TAB. 3.3 – Les coûts de communication pour  $P = Q = 16$ ,  $r = 3$ , et  $s = 5$ .

**Communications déséquilibrées** Le troisième exemple est la redistribution de  $P = 15$  processeurs et  $r = 3$  éléments par bloc vers  $Q = 15$  processeurs et  $s = 5$  éléments par bloc. Comme on peut le constater dans le tableau 3.7, le schéma de communication est très déséquilibré, dans le sens où les processeurs peuvent avoir un nombre différent de messages à envoyer et/ou à recevoir. Notre algorithme est capable de manipuler ce genre de situation compliquée.

**Grilles différentes de processeurs** L'exemple suivant est la redistribution de  $P = 12$  processeurs et  $r = 4$  éléments par bloc vers  $Q = 8$  processeurs et  $s = 3$  éléments par bloc. Nous l'avons choisi parce qu'il traite du cas  $P \neq Q$ . Nous voulons montrer ici que la taille des grilles de processeurs peut ne pas être la même. Le tableau 3.10 nous montre là encore une grille de communication déséquilibrée.

E/R	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	N.M.
0	7	6	2	6	7	2	5	7	3	4	7	4	3	7	5	2	16
1	4	3	7	5	2	7	6	2	6	7	2	5	7	3	4	7	16
2	5	7	3	4	7	4	3	7	5	2	7	6	2	6	7	2	16
3	6	2	6	7	2	5	7	3	4	7	4	3	7	5	2	7	16
4	3	7	5	2	7	6	2	6	7	2	5	7	3	4	7	4	16
5	7	3	4	7	4	3	7	5	2	7	6	2	6	7	2	5	16
6	2	6	7	2	5	7	3	4	7	4	3	7	5	2	7	6	16
7	7	5	2	7	6	2	6	7	2	5	7	3	4	7	4	3	16
8	3	4	7	4	3	7	5	2	7	6	2	6	7	2	5	7	16
9	6	7	2	5	7	3	4	7	4	3	7	5	2	7	6	2	16
10	5	2	7	6	2	6	7	2	5	7	3	4	7	4	3	7	16
11	4	7	4	3	7	5	2	7	6	2	6	7	2	5	7	3	16
12	7	2	5	7	3	4	7	4	3	7	5	2	7	6	2	6	16
13	2	7	6	2	6	7	2	5	7	3	4	7	4	3	7	5	16
14	7	4	3	7	5	2	7	6	2	6	7	2	5	7	3	4	16
15	2	5	7	3	4	7	4	3	7	5	2	7	6	2	6	7	16
N.M.	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	

TAB. 3.4 – Grille de communication pour  $P = Q = 16$ ,  $r = 7$ , et  $s = 11$ . La taille des messages indiquée est calculée pour la redistribution d'un vecteur de taille  $L = 1232$ .

E/R	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	c	f	n	g	d	p	i	e	m	k	a	j	l	b	h	o
1	k	m	d	h	o	c	f	n	g	e	p	i	b	l	j	a
2	h	b	l	k	a	j	m	d	i	n	e	f	o	g	c	p
3	f	n	g	e	p	i	d	l	j	c	k	m	a	h	o	b
4	l	e	i	n	c	f	o	g	d	p	h	a	m	k	b	j
5	b	l	j	c	k	m	e	i	n	d	f	o	g	a	p	h
6	o	g	c	p	i	d	l	k	b	j	m	e	h	n	a	f
7	a	i	p	d	f	n	g	b	o	h	c	l	j	e	k	m
8	m	k	b	j	l	a	h	p	e	f	n	g	d	o	i	c
9	g	a	o	i	e	l	j	c	k	m	b	h	p	d	f	n
10	i	p	e	f	n	g	a	o	h	b	l	k	c	j	m	d
11	j	d	k	m	b	h	p	a	f	o	g	c	n	i	e	l
12	d	o	h	b	m	k	c	j	l	a	i	p	e	f	n	g
13	p	c	f	o	g	e	n	h	a	l	j	b	k	m	d	i
14	e	j	m	a	h	o	b	f	p	g	d	n	i	c	l	k
15	n	h	a	l	j	b	k	m	c	i	o	d	f	p	g	e

TAB. 3.5 – Les étapes de communication pour  $P = Q = 16$ ,  $r = 7$ , et  $s = 11$ .

Étape	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	Total
Coût	7	7	7	7	7	6	6	5	5	4	4	3	3	2	2	2	77

TAB. 3.6 – Les coûts de communication pour  $P = Q = 16$ ,  $r = 7$ , et  $s = 11$ .

**Exemple avec entrelacement** Considérons la redistribution dont les paramètres sont  $P = Q = 6$  processeurs,  $r = 2$  éléments par bloc et  $s = 3$  éléments par bloc. Cet exemple a été choisi pour montrer que nous pouvons encore optimiser le coût global des communications en permettant l'entrelacement des messages. Ces communications sont présentées dans le tableau 3.13.

Nous pouvons remarquer que les processeurs sources  $p \in \{0, 2, 3, 5\} \subset \mathcal{P} = \{0, 1, \dots, P - 1\}$  envoient 3 messages et que chaque processeur  $q \in \mathcal{Q} = \{0, 1, \dots, Q - 1\}$  reçoit 4 messages. Cependant chaque processeur  $p \in \{1, 4\} \subset \mathcal{P}$  envoie 6 messages. Aussi, doit-on pouvoir trouver un ordonnancement en 6 étapes totalisant 6 messages envoyés. Nous pourrions alors améliorer notre algorithme en cherchant un ordonnancement des communications plus efficace.

La figure 3.3 donne un ordonnancement optimal des communications de notre exemple lorsque les étapes de communication ne se recouvrent pas. L'axe horizontal représente le temps, et chaque paire «  $p, q$  » signifie que le processeur  $p$  de  $\mathcal{P}$  envoie un message au processeur  $q$  de

E/R	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	N.M.
0	3	-	-	3	-	-	3	-	-	3	-	-	3	-	-	5
1	2	1	-	2	1	-	2	1	-	2	1	-	2	1	-	10
2	-	3	-	-	3	-	-	3	-	-	3	-	-	3	-	5
3	-	1	2	-	1	2	-	1	2	-	1	2	-	1	2	10
4	-	-	3	-	-	3	-	-	3	-	-	3	-	-	3	5
5	3	-	-	3	-	-	3	-	-	3	-	-	3	-	-	5
6	2	1	-	2	1	-	2	1	-	2	1	-	2	1	-	10
7	-	3	-	-	3	-	-	3	-	-	3	-	-	3	-	5
8	-	1	2	-	1	2	-	1	2	-	1	2	-	1	2	10
9	-	-	3	-	-	3	-	-	3	-	-	3	-	-	3	5
10	3	-	-	3	-	-	3	-	-	3	-	-	3	-	-	5
11	2	1	-	2	1	-	2	1	-	2	1	-	2	1	-	10
12	-	3	-	-	3	-	-	3	-	-	3	-	-	3	-	5
13	-	1	2	-	1	2	-	1	2	-	1	2	-	1	2	10
14	-	-	3	-	-	3	-	-	3	-	-	3	-	-	3	5
N.M.	6	9	6	6	9	6	6	9	6	6	9	6	6	9	9	

TAB. 3.7 – Grille de communication pour  $P = Q = 15$ ,  $r = 3$ , et  $s = 5$ . La taille des messages indiquée est calculée pour la redistribution d'un vecteur de taille  $L = 225$ .

E/R	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	c	-	-	d	-	-	e	-	-	a	-	-	b	-	-
1	f	h	-	b	i	-	j	d	-	g	c	-	e	a	-
2	-	e	-	-	d	-	-	c	-	-	a	-	-	b	-
3	-	i	f	-	j	c	-	g	e	-	h	b	-	d	a
4	-	-	c	-	-	e	-	-	d	-	-	a	-	-	b
5	d	-	-	e	-	-	a	-	-	b	-	-	c	-	-
6	e	j	-	a	h	-	b	i	-	c	f	-	d	g	-
7	-	d	-	-	e	-	-	a	-	-	b	-	-	c	-
8	-	g	d	-	b	f	-	j	a	-	i	e	-	h	c
9	-	-	e	-	-	a	-	-	b	-	-	c	-	-	d
10	b	-	-	f	-	-	c	-	-	d	-	-	a	-	-
11	a	b	-	c	g	-	d	h	-	e	j	-	f	i	-
12	-	c	-	-	a	-	-	b	-	-	d	-	-	e	-
13	-	a	b	-	c	d	-	e	f	-	g	h	-	j	i
14	-	-	a	-	-	b	-	-	c	-	-	d	-	-	e

TAB. 3.8 – Les étapes de communication pour  $P = Q = 15$ ,  $r = 3$ , et  $s = 5$ .

Etape	a	b	c	d	e	f	g	h	i	j	Total
Coût	3	3	3	3	3	3	2	2	2	2	26

TAB. 3.9 – Les coûts de communication pour  $P = Q = 15$ ,  $r = 3$ , et  $s = 5$ .

$Q$ . Par exemple, durant la troisième unité de temps, le processeur 1 de  $P$  envoie un message au processeur 1 de  $Q$  et le processeur 4 de  $P$  envoie un message au processeur 2 de  $Q$ .

Nous obtenons ici 9 unités de temps, ce qui est optimal sans entrelacement des étapes de communication. Cependant, ce temps peut être réduit si nous autorisons des entrelacements. Le tableau 3.13 montre que chaque processeur peut finir ses communications en 6 unités de temps. Si nous permettons l'entrelacement des étapes de communication, on peut regrouper les messages les plus petits pour reboucher les « trous » de l'ordonnancement. Nous l'avons fait, comme présenté à la figure 3.4 : nous obtenons un ordonnancement optimal en 6 unités de temps. Les liens de communication entre processeurs sont utilisés tout le temps et on n'a découpé aucun message. On peut vérifier à la figure 3.4 que chaque processeur envoie et/ou reçoit au maximum un message à chaque unité de temps.

E/R	0	1	2	3	4	5	6	7	N.M.
0	3	1	-	-	-	-	-	-	2
1	-	2	2	-	-	-	-	-	2
2	-	-	1	3	-	-	-	-	2
3	-	-	-	-	3	1	-	-	2
4	-	-	-	-	-	2	2	-	2
5	-	-	-	-	-	-	1	3	2
6	3	1	-	-	-	-	-	-	2
7	-	2	2	-	-	-	-	-	2
8	-	-	1	3	-	-	-	-	2
9	-	-	-	-	3	1	-	-	2
10	-	-	-	-	-	2	2	-	2
11	-	-	-	-	-	-	1	3	2
N.M.	2	4	4	2	2	4	4	4	

E/R	0	1	2	3	4	5	6	7
0	a	c	-	-	-	-	-	-
1	-	b	a	-	-	-	-	-
2	-	-	c	a	-	-	-	-
3	-	-	-	-	a	c	-	-
4	-	-	-	-	-	b	a	-
5	-	-	-	-	-	-	c	a
6	b	d	-	-	-	-	-	-
7	-	a	b	-	-	-	-	-
8	-	-	d	b	-	-	-	-
9	-	-	-	-	b	d	-	-
10	-	-	-	-	-	a	b	-
11	-	-	-	-	-	-	d	b

TAB. 3.10 – Grille de communication pour  $P = 12$ ,  $Q = 8$ ,  $r = 4$ , et  $s = 3$ . La taille des messages indiquée est calculée pour la redistribution d'un vecteur de taille  $L = 48$ .

Etape	a	b	c	d	Total
Coût	3	3	1	1	8

TAB. 3.12 – Les coûts de communication pour  $P = 12$ ,  $Q = 8$ ,  $r = 4$ , et  $s = 3$ .

E/R	0	1	2	3	4	5	N.M.
0	2	-	2	-	2	-	3
1	1	1	1	1	1	1	6
2	-	2	-	2	-	2	3
3	2	-	2	-	2	-	3
4	1	1	1	1	1	1	6
5	-	2	-	2	-	2	3
N.M.	4	4	4	4	4	4	

TAB. 3.13 – Grille de communication pour  $P = Q = 6$ ,  $r = 2$ , et  $s = 3$ . La taille des messages indiquée est calculée pour la redistribution d'un vecteur de taille  $L = 36$ .

### 3.2.4 Résultats généraux

Dans un premier temps, nous avons simplifié le problème en montrant que le problème général peut se ramener au problème de la redistribution  $(P, r) \rightarrow (Q, s)$ , telle que  $r$  et  $s$  sont premiers entre eux. Ensuite, nous avons déterminé les conditions pour avoir un échange total dans le schéma de communication [88].

Puis nous avons trouvé des solutions au problème de l'ordonnement des communications, dans le cas où l'on ne peut pas entrelacer les communications (section 3.2.5) ou lorsqu'on peut les entrelacer (section 3.2.6).

### 3.2.5 Problème de la redistribution sans entrelacement

Nous avons emprunté quelques outils de la théorie des graphes. Nous considérons la grille de communication comme un graphe  $G = (V, E)$ , pour lequel

- $V = \mathcal{P} \cup \mathcal{Q}$ , où  $\mathcal{P} = \{0, 1, \dots, p-1\}$  est l'ensemble des processeurs qui envoient des messages, et  $\mathcal{Q} = \{0, 1, \dots, q-1\}$  est l'ensemble des processeurs qui en reçoivent,
  - $e = (p, q) \in E$  si et seulement si l'entrée  $(p, q)$  de la grille de communication est nulle.
- $G$  est un graphe biparti (toutes les arêtes relient un sommet de  $\mathcal{P}$  à un sommet de  $\mathcal{Q}$ ). Le degré de  $G$ , défini comme le plus grand degré de ses sommets, est  $d_G = \max\{m_L, m_C\}$ . Plus précisément, on considère que les arêtes sont pondérées : le poids de chaque arête  $(p, q)$  est la taille  $taille(p, q)$



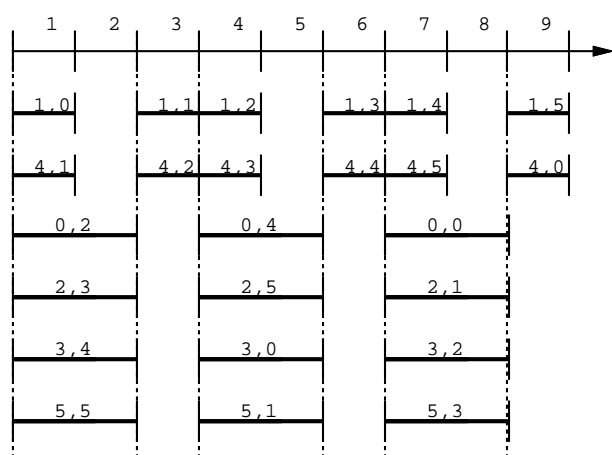


FIG. 3.3 – Ordonnancement des communications pour l'exemple  $P = 6, r = 2, Q = 6$  et  $s = 3$  (sans entrelacement des étapes de communication).

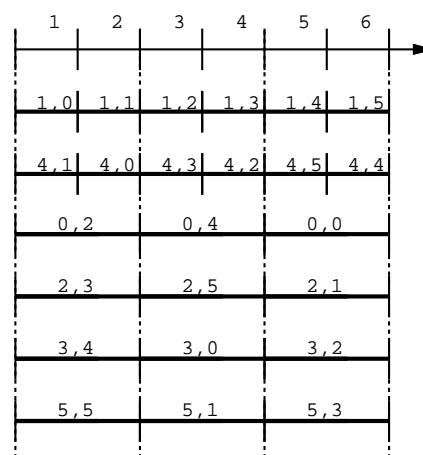


FIG. 3.4 – Ordonnancement des communications pour l'exemple  $P = 6, r = 2, Q = 6$  et  $s = 3$  (avec entrelacement des étapes de communication).

du message envoyé par le processeur  $p$  au processeur  $q$ .

Nous adoptons les deux stratégies : par étapes ou à l'aide d'une heuristique gloutonne.

**algorithme par étapes** Si l'on spécifie le nombre d'étapes, on doit choisir à chaque itération un recouvrement maximum qui sature tous les sommets de degré maximum. On peut choisir n'importe quel recouvrement de ce type. Le poids d'un recouvrement est défini comme la somme des poids de ses arêtes. Ainsi une idée qui semble naturelle est de choisir, parmi tous les recouvrements, un recouvrement de poids maximal.

**heuristique gloutonne** Si l'on spécifie le coût total, on peut adopter une heuristique gloutonne qui sélectionne un recouvrement de poids maximal à chaque étape. On pourrait alors finir avec un ordonnancement dont le nombre d'étapes est supérieur à  $NE_{opt}$ , mais dont le coût total est moindre.

Pour développer ces deux approches, nous avons fait appel aux fondements de la programmation linéaire (voir [135, chapitre 30]).

La stratégie présentée dans ce chapitre rend possible la manipulation directe de redistributions d'une distribution  $CYCLIC(r)$  vers une autre distribution arbitraire  $CYCLIC(r)$  sur une même grille de processeurs, contrairement à la stratégie évoquée par Walker et Otto qui requiert deux redistributions : une de  $CYCLIC(r)$  vers  $CYCLIC(l_{cm}(r, s))$  et la seconde de  $CYCLIC(l_{cm}(r, s))$  vers  $CYCLIC(s)$ .

Les résultats théoriques ont été corroborés par les expériences ([87] et chapitre E du document annexe).

### 3.2.6 Problème de la redistribution avec entrelacement

Dans le paragraphe précédent, nous avons décrit des algorithmes optimaux en terme de coût total mais notre modélisation peut être considérée comme incomplète. En effet, les communications étant indépendantes et point à point, nous avons observé qu'elles sont naturellement *entrelacées*, c'est-à-dire que deux processeurs communiquent dès qu'ils ont achevé l'opération précédente, sans attendre que tous les autres processeurs aient aussi achevé l'opération

précédente. Nous élargissons donc le problème aux « ordonnancements entrelacés », c'est-à-dire que le calcul du coût total tient compte de l'aspect non-collectif des communications. Nous cherchons ainsi à améliorer l'ordonnement par ces entrelacements.

De plus, nous avons préalablement considéré que tous les messages étaient envoyés sous la forme d'un seul paquet, alors qu'il est possible de découper ces messages. Nous avons donc également pris en compte la possibilité de découpage des messages.

Ainsi, un autre objectif dans le problème de redistribution est de minimiser le nombre d'initialisations de communications, ce qui revient à minimiser le nombre de messages envoyés ou reçus. Pour cela, un objectif secondaire sera de construire un ordonnancement avec le moins de « découpages » possible de messages.

Un ordonnancement est alors optimal si le temps de la redistribution est égal au temps de communication du processeur qui communique le plus. Nous avons alors montré qu'un ordonnancement optimal était toujours possible en découplant les messages, ce qui n'est pas le cas sans ce découpage. Nous avons alors présenté une heuristique qui fait le moins de découpages possibles, et nous avons isolé les classes de redistributions pour lesquelles un ordonnancement optimal sans découpage était possible. Nous avons construit cet ordonnancement [86].

### 3.3 Distribution automatique

La distribution des données est un problème difficile, même pour un spécialiste du parallélisme. La complexité des applications alliée à l'utilisation de bibliothèques de calcul de haut niveau rend cette tâche excessivement dépendante de l'architecture cible.

Dans cette section, je présente mes travaux avec C. Randriamaro sur la génération de distributions de type HPF pour des programmes utilisant des appels à des bibliothèques BLAS ou LAPACK. Notre but était d'aider l'utilisateur de telles bibliothèques à transformer son code en un code parallèle en utilisant des modélisations de routines, des (re)distributions ainsi qu'un ensemble d'algorithmes de distribution automatique [100, 231].

L'approche généralement utilisée repose sur le schéma suivant. Le programme est considéré comme une succession de phases (chaque phase est généralement un nid de boucles ou l'appel d'une fonction qui travaille sur des matrices distribuées). Pour chaque phase, on choisit un ensemble de distributions, appelé choix de distributions, puis on sélectionne l'une des distributions de l'ensemble dans le but d'optimiser le temps d'exécution du programme global, en tenant compte du coût des redistributions éventuelles. Dans notre cas, nous considérons les phases comme des appels aux fonctions des bibliothèques PBLAS et ScaLAPACK.

#### 3.3.1 Travaux précédents

Les premiers travaux ont eu pour but de trouver automatiquement l'alignement des données entre elles [65, 186]. Nous nous sommes plutôt intéressés au problème de la distribution des données une fois qu'elles sont alignées (ce qui est le cas si l'on utilise ScaLAPACK comme bibliothèque cible).

Kennedy et Kremer [36] définissent une phase comme un ensemble d'opérations sur des matrices distribuées. Leur approche est principalement basée sur deux structures de données. Premièrement, ils créent **le Graphe de Flot de Contrôle des Phases** (GFCP) dans lequel chaque sommet représente toutes les instructions d'une même phase. Les arêtes sont pondérées par la fréquence et la probabilité d'exécution. Ils construisent alors **le Graphe de Distribution de Données** (GDD) qui est une extension du GFCP. Chaque sommet est multiplié par le nombre de ses distributions potentielles et contient ainsi une phase et une distribution potentielle. Il y a une arête entre deux sommets si au moins l'une des matrices communes aux deux sommets doit être redistribuée entre le sommet source et le sommet destination. Les sommets (et arêtes) sont pondérés par l'estimation du temps d'exécution des instructions (et redistributions). La

détection d'une distribution pour chaque étape revient alors à chercher un *chemin minimal*, un chemin de poids minimum passant une fois par chacune des phases. Ils ont montré que chacun des deux problèmes était NP-Complet [176], aussi bien le choix de la liste des distributions pour construire le GDD, que la recherche du chemin minimal.

Garcia, Ayguadé et Labarta [130] se limitent aux distributions 1D cycliques ou par blocs. Chaque instruction matricielle n'a alors le choix qu'entre deux distributions, ce qui permet de construire un GDD avec deux fois plus de sommets qu'il n'y a d'instructions matricielles. Les auteurs trouvent ensuite le chemin minimal en utilisant les techniques de Kennedy et Kremer.

Lee, Wang et Yang [183] intègrent de plus la distribution bloc-cyclique avec une taille de blocs unique et fixée (1D et 2D). En s'intéressant aux programmes d'imagerie, ils étudient des programmes qui ne travaillent que sur une seule matrice distribuée. Les boucles sont traitées séparément, ce qui évite les cycles dans le GDD. La recherche du chemin minimal est donc optimale avec l'algorithme de Dijkstra.

L'algorithme d'Anderson et Lam [10] travaille sur un graphe dont les sommets correspondent aux boucles et nids de boucles, et les arêtes aux redistributions de données potentielles entre les nids de boucles. Ces arêtes sont pondérées par le coût de leurs redistributions. Le poids des sommets représente le temps d'exécution des nids de boucles lorsque les matrices sont distribuées de façon optimale. Ils sélectionnent la plus grosse arête. Les deux sommets des extrémités sont alors réunis, puis on calcule la meilleure distribution commune à ces sommets (dite *distribution statique*). Si le coût d'exécution des deux phases fusionnées, avec la distribution statique, est inférieur à la somme des coûts des deux précédentes distributions et de la redistribution, alors les deux sommets sont regroupés en un seul. Ceci est répété avec la plus grande arête suivante et ainsi de suite jusqu'à la dernière arête.

À l'inverse, Palermo et Banerjee [217] considèrent tout d'abord le programme comme une seule phase englobant toutes les instructions. La distribution statique optimale est choisie pour cette phase. Elle est alors scindée en deux, générant deux phases de plus petites tailles, chacune ayant sa distribution statique optimale. Ainsi, toutes les divisions possibles sont testées pour choisir celle qui permet de minimiser le coût du programme, sans tenir compte du coût des redistributions éventuelles. Les deux phases sont alors également divisées en deux, et ainsi de suite, de façon récursive, jusqu'à ne plus avoir de division générant de coût inférieur à celui de la phase à diviser. À chaque phase non divisée (dite *phase de base*) est alors associée sa distribution statique optimale, ainsi que la distribution de la phase mère (dont la division a produit cette phase de base), celle de la phase mère de cette dernière, et ainsi de suite jusqu'à la phase regroupant le programme global. Enfin, un graphe ressemblant au GDD est construit. Il est utilisé pour la recherche d'un chemin minimal.

Enfin, l'algorithme de Peizong Lee [184] considère le programme comme une liste d'instructions matricielles. Tous les regroupements possibles entre voisins sont définis : pour  $N$  phases, pour chaque sommet  $x$ , les regroupements définis sont  $(x)$ ,  $(x, x + 1)$ ,  $(x, x + 1, x + 2)$ , ...,  $(x, \dots, N)$ . Pour chaque regroupement la distribution statique optimale est calculée. Ensuite, on construit un graphe dans lequel chaque regroupement est un sommet ; une arête relie deux sommets  $u$  et  $v$  si la dernière instruction de  $u$  précède directement la première instruction de  $v$ . En pondérant les sommets par leur temps d'exécution et les arêtes par les temps de redistribution des matrices, on peut alors chercher le chemin minimal.

Le tableau 3.14 récapitule les différents algorithmes de distribution automatique étudiés pour ALASCA.

Des exemples d'utilisation de ces algorithmes sont donnés dans la thèse de C. Randriamaro [231].

### 3.3.2 Modélisation des coûts

La modélisation précise des routines de calcul et de communication est de plus en plus problématique. Si avec les architectures de l'ancienne génération (jusqu'au début des années

Algorithmes	Référence	Distributions possibles	Choix des distributions	Recherche d'un chemin
Garcia et al.	[130]	cyclique et par bloc	-	NP-Complet
Lee et al.	[183]	idem et Cyclic(r)	-	polynomial
Anderson et Lam	[10]	bloc-cyclique	regroupement	-
Palermo et Banerjee	[217]	bloc-cyclique	découpage	polynomial
Peizong Lee	[184]	bloc-cyclique	découpage	polynomial

TAB. 3.14 – Récapitulatif des différents algorithmes de distribution automatique.

90), on avait des modèles précis, cela n'est plus le cas avec les nouvelles architectures et leurs systèmes évolués.

Lorsqu'on veut choisir de manière statique telle ou telle distribution, il est important de pouvoir évaluer le coût des redistributions de matrices. On peut trouver des modélisations raisonnables avec les routines simples (du style caterpillar [223]), il est quasiment impossible de trouver un modèle pour les routines évoluées qui utilisent un ordonnancement des communications comme ceux présentés dans les sections précédentes. La figure 3.5 montre les temps d'exécution de la routine de redistribution de ScaLAPACK lorsqu'on fait varier la taille des blocs et la forme de la grille. Nous avons donc pris comme modélisation une approximation du nombre maximal de messages échangés à un instant donné. De plus, nous avons limité les redistributions à des modifications de la forme de la grille.

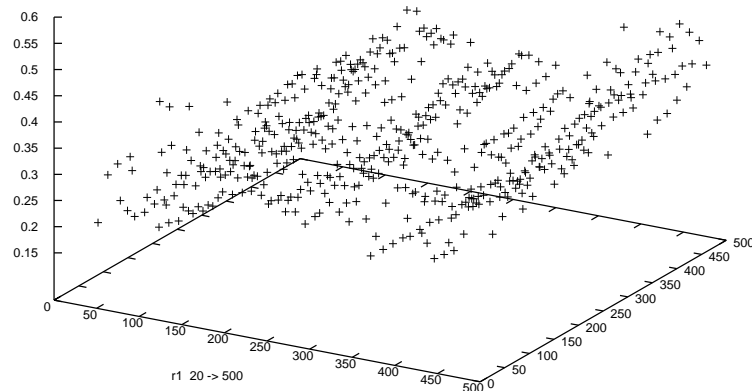


FIG. 3.5 – Temps d'exécution de la routine PDGEMR2D en fonction de la taille des blocs (grilles  $1 \times 4$  à  $4 \times 1$ ).

En ce qui concerne les routines de calculs, on peut arriver plus facilement à des modélisations précises [102, 105]. La figure 3.6 (respectivement 3.7) montre les courbes d'exécution de la routine PDGEMM (respectivement PDTRSM) des PBLAS en fonction de la taille des blocs.

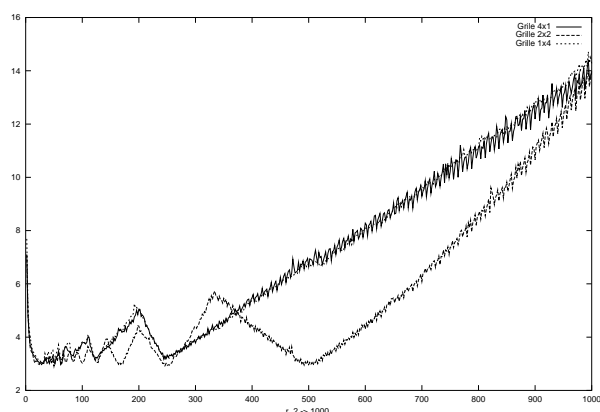


FIG. 3.6 – Temps d'exécution de la routine PDGEMM en fonction de la taille des blocs (4 processeurs, matrices  $1000 \times 1000$ ).

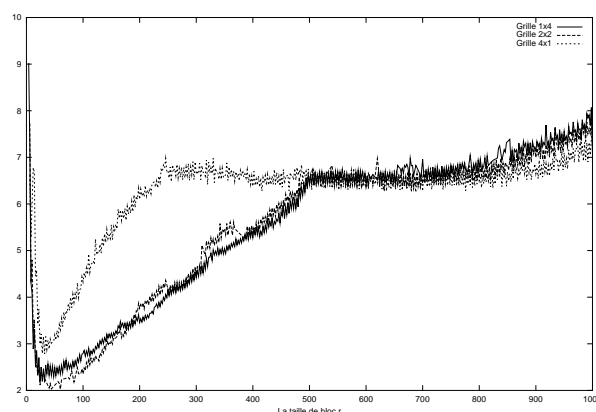


FIG. 3.7 – Temps d'exécution de la routine PDTRSM en fonction de la taille des blocs (4 processeurs, matrices  $1000 \times 1000$ ).

### 3.3.3 Algorithme de regroupement

Nous avons travaillé avec C. Randriamaro sur un algorithme permettant de tirer parti des travaux précédents en les appliquant à nos propres problèmes [100, 231]. Partant de l'algorithme de Peizong Lee comme algorithme de référence, nous avons modifié son graphe en y ajoutant des distributions non-traitées par cet algorithme et en divisant les phases à la manière de Palermo et Banerjee. Cela rajoute des sommets au graphe mais on ajoute de la précision à sa recherche de distributions efficaces. Ensuite, nous appliquons simplement l'algorithme de Dijkstra de recherche du chemin minimal.

### 3.3.4 Algorithme des chemins maximaux

L'algorithme précédent est efficace mais il ne traite que des séquences d'instructions en les regroupant en phases selon leur voisinage. Nous avons constaté que suivant les dépendances d'accès aux données, le graphe n'était pas forcément une séquence simple d'instructions et qu'il fallait le considérer sans orientation et privilégier en premier lieu les redistributions de coût maximal, et ceci quelle que soit leur position dans le graphe. Nous cherchons donc le chemin élémentaire maximal et nous y appliquons l'algorithme de regroupement. Nous poursuivons récursivement jusqu'à ce que l'on ait plus qu'un seul sommet.

Des exemples d'utilisation de ces algorithmes sont donnés dans la thèse de C. Randriamaro [231] et des copies d'écrans montrant l'utilisation de ces algorithmes dans l'outil ALASCA sont montrées dans la section 3.4.7.

## 3.4 TransTool

### 3.4.1 Introduction

Le but du projet TRANS TOOL, démarré au LaBRI en 1995, était de développer un environnement convivial d'écriture de programmes HPF. Cet outil, développé autour de l'éditeur XEmacs, devait permettre de pouvoir restructurer des programmes écrits en Fortran 77 à l'aide de nos résultats théoriques et de nos prototypes. En effet, comme nous l'avons évoqué dans la

section 1.2 du chapitre 1, l'écriture de programmes HPF est loin d'être aisée pour le programmeur d'applications parallèles. Les directives, quoique relativement simples, permettent toutes les fantaisies en termes de distributions et de parallélisme et il est difficile d'obtenir à la fois un programme performant, portable et lisible. Nous avons donc pensé qu'un outil interactif de parallélisation serait une solution acceptable pour le développement d'applications avec HPF et également que l'outil résultant serait une bonne vitrine de nos résultats de recherche dans ce domaine.

Notre but était donc de développer un outil de parallélisation interactive et semi-automatique de programmes Fortran 77 « nettoyés », c'est-à-dire déjà passés dans un outil comme Foresys de Simulog [244]. Nous sommes donc partis d'un éditeur puissant et programmable, XEmacs, auquel nous avons ajouté divers outils : un parseur, un analyseur de dépendances, un outil d'insertion de directives HPF et un noyau d'optimisation permettant de paralléliser les boucles, distribuer les matrices et insérer des appels à l'interface HPF des routines LOCCS. TRANSTOOL, dans sa version finale, devait offrir :

- l'édition de source avec analyse du code,
- l'analyse de dépendances de nids de boucles particuliers,
- application de l'algorithme d'Allen et Kennedy [4] et détection automatique de boucles INDEPENDENT,
- la détection automatique de schémas de calculs pipelines et la génération de code LOCCS,
- un support pour le placement de données (ajout de directives de distribution),
- la visualisation et l'évaluation de placements de données,
- l'interfaçage de bibliothèques de calcul à haute performance (comme ScaLAPACK),
- l'interfaçage avec divers compilateurs HPF,
- une interface de compilation et d'exécution.

Pour développer TRANSTOOL, nous sommes partis du constat que le développement d'une application numérique sur une machine à mémoire distribuée se faisait avec un cycle comme celui décrit dans la figure 3.8. Les outils interactifs peuvent être développés pour aider l'utilisateur dans la première phase (distribution des données). La distribution doit donner le parallélisme le plus important et réduire les communications. Ceci peut être évalué grâce à d'autres outils (profilers, outils de monitoring). À partir de ces distributions, l'utilisateur doit être capable d'écrire des directives HPF de distribution dans la partie déclaration de son code. Celles-ci doivent alors être propagées à l'intérieur des routines. Si les distributions diffèrent entre le code principal et l'intérieur des routines, des redistributions doivent être insérées aux frontières. Des directives de parallélisation de boucles peuvent également être ajoutées. Le code peut alors être compilé et exécuté. Si des traces ont été générées, l'utilisateur peut avoir une idée du comportement de son code et de la qualité des distributions et de la parallélisation. Le code peut alors être optimisé et re-exécuté plusieurs fois pour obtenir les meilleures performances. Une question reste cependant en suspens : ce code est-il portable (en termes de performances) ?

Avec notre utilisation de divers compilateurs HPF sur diverses architectures, nous avons constaté à maintes reprises que ceux-ci ne suivaient pas forcément les directives insérées dans le code. Ceci est un problème pour l'utilisateur puisqu'il peut avoir une vue « corrompue » de son programme. Nous avons donc pensé qu'une interaction forte entre le compilateur et l'outil interactif de parallélisation était donc nécessaire.

TRANSTOOL faisait partie d'un projet plus important appelé HPFIT (High Performance Fortran Integrated Tools) [45, 46]. Outre REMAP, HPFIT faisait collaborer les équipes ALIENOR du LABRI et le GMD/SCAI à Bonn (Allemagne). Notre but était d'unir nos forces pour développer des outils de transformation source-à-source de programme Fortran et de partager la définition des API de nos outils.

### 3.4.2 Travaux connexes

Nous n'étions bien sûr pas les seuls à développer de tels outils de restructuration de programmes Fortran.

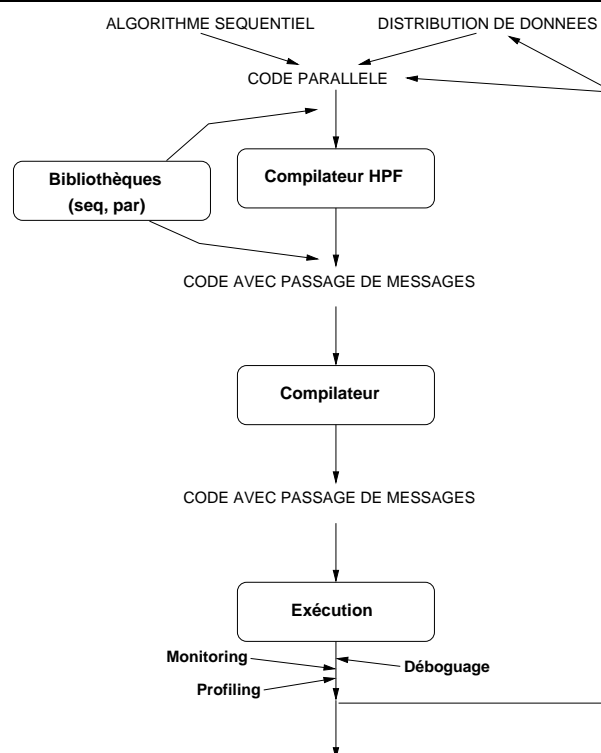


FIG. 3.8 – Cycle de développement d’une application sur une plate-forme à mémoire distribuée.

L’un des premiers projets autour d’un éditeur « intelligent » pour la parallélisation d’applications écrites en Fortran77 est Parascopie (PED) de l’université de Rice [167] développé pour les machines à mémoire partagée. Il a été conçu à partir d’autres outils développés dans cette université comme  $\mathbb{R}_n$ , PFC et PTOOL. PED permettait la recherche de dépendances dans un graphe dont la taille était limitée par des informations de filtrage. Plusieurs optimisations avaient été ajoutées comme la transformation et la parallélisation de boucles, la suppression de dépendances et des optimisations mémoire.

Le D Editor [148] a été développé en partie à partir de PED également dans l’Université de Rice. Il a été développé pour la parallélisation d’applications écrites en Fortran D, un langage data-parallel qui a été en partie à la base de la conception du langage HPF. Le D-editor contient un outil d’analyse interprocédurale, le compilateur Fortran D et des outils pour la distribution automatique, la détection de conflits d’accès aux données, des outils d’estimation de performances statiques et de monitoring. Cet outil contient cinq fenêtres : une fenêtre générale présente un résumé des boucles et des routines du programme (les boucles qui restreignent le parallélisme sont mises en évidence); une fenêtre de dépendances affiche les dépendances portées par les boucles sélectionnées; la fenêtre de communications affiche toutes les communications associées avec la boucle sélectionnée; la fenêtre de placement de données affiche les informations de décomposition de données pour chaque tableau utilisé dans la boucle; et enfin une fenêtre de source affiche le code du programme. L’environnement d’analyse de performances Pablo a également été intégré dans le D-editor [2].

Le Vienna FORTRAN Compilation System (VFCS) [273] est un système de compilation source-à-source basé sur Vienna Fortran, un autre ensemble d’extensions de Fortran pour le data-parallelisme. VFCS a également joué un rôle important dans la conception d’HPF et est assez proche de Fortran D. Il contient un compilateur, un estimateur interactif de performances (P3T [63]) et un système de mesure de performances (VFPMS).

L'environnement Annai [75], développé par le CSCS en collaboration avec NEC, utilise MPI comme interface de communication pour plusieurs plates-formes à mémoire distribuée : la Cenju-3 de NEC, le Cray T3D, la Paragon d'Intel et les stations de travail multi-processeurs. Il est constitué d'un compilateur HPF étendu (avec des extensions pour les calculs irréguliers), un débogueur parallèle et un analyseur de performances, conçu en collaboration avec les développeurs d'applications.

Le *Computer Aided Parallelization Tools* (CAPTools [155]) est un ensemble d'outils développé à l'université de Greenwich. Cet environnement peut montrer les dépendances avec une fenêtre graphique. L'utilisateur peut interagir durant le processus de parallélisation en donnant des informations sur les valeurs des variables ou en « supprimant » des dépendances, etc. Il possède une fenêtre pour partitionner le code et les structures de données et un éditeur pour voir les appels de routines de communication générés. Le code généré par CAPTools est écrit en Fortran 77 avec des appels de routines de communication explicites.

Enfin PIPS, développé à l'École des Mines de Paris à Fontainebleau par François Irigoien et son équipe, permet d'appliquer à un code Fortran tout un ensemble de transformations et de générer du HPF (entre autres). Il possède par ailleurs une analyse interprocédurale puissante [159].

Dans les outils issus de l'industrie, on peut citer ForgeExplorer qui est un outil de parallélisation basé sur un afficheur de source interactif et qui permet également quelques transformations de boucles et également Foresys de Simulog qui, en plus d'effectuer du nettoyage de codes Fortran, effectue quelques transformations de boucles pour l'insertion de directives HPF.

On peut enfin citer d'autres projets importants autour de la parallélisation de programmes Fortran comme SUIF [5] et Paradigm [28]. Pour un tour d'horizon plus complet des outils de parallélisation autour d'HPF, voir le survey de J.-L. Pazat [220].

### 3.4.3 Outils TRANSTOOL

Dans ce paragraphe, je décris les outils insérés dans TRANSTOOL pour lesquels j'ai directement participé au développement. TRANSTOOL, tel qu'il a été démontré lors de la revue finale du projet EuroTOPS contenait également une interface avec l'outil NESTOR [243] développé par G.-A. Silber et A. Darte.

Outre XEmacs, TRANSTOOL a été développé à l'aide des outils suivants :

- un parseur F77 (issu du compilateur Adaptor du laboratoire GMD/SCAI),
- NESTOR [243] comme outil de transformation de boucles (LIP),
- un analyseur de dépendances (Petit [166] de l'Université du Maryland),
- un outil de distribution automatique (ALASCA).

Les premières versions de TRANSTOOL étaient développées en C et TCL/Tk. Les suivantes utilisaient C, C++, GTK et Java ainsi que les outils développés par C. Randriamaro [231] (Wil, LibParsing).

### 3.4.4 Visualisation de dépendances

Dans sa première version, TRANSTOOL permettait de voir les dépendances dans le code (figure 3.9)<sup>1</sup>. Des filtres permettaient de sélectionner des nids de boucles particuliers ou des types de dépendances particuliers (flot, anti, output). Cette vue était intéressante pour le spécialiste puisque l'on pouvait pointer dans le code les dépendances trouvées par Petit, par contre, elle était quasiment inutilisable pour le développeur d'application qui recevait trop d'informations. Nous avons donc abandonné ce type d'informations dans les versions suivantes de l'outil.

---

<sup>1</sup>Développé avec L. Tricon et G. Lebourgeois.



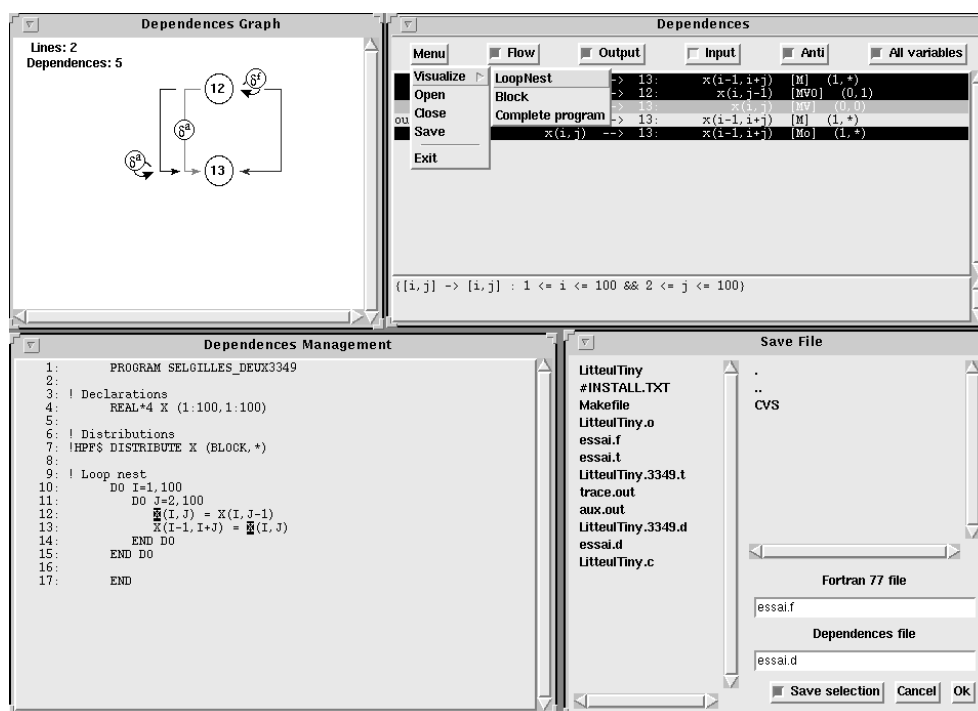


FIG. 3.9 – Vue des dépendances dans la première version de TRANSTOOL.

### 3.4.5 Génération automatique d'appels à des routines LOCCS

Nous avons généré automatiquement des appels aux extensions de la bibliothèque DALIB pour les LOCCS (routine `DALIB_LOCCS_DRIVER`) en utilisant l'algorithme pour trouver les boucles *cross-processor* donné dans la thèse de Tseng [257].

### 3.4.6 HPFize

La première version d'HPFize (développée au LaBRI en collaboration avec M.-C. Counilh) offrait un ensemble de fenêtres pour l'insertion de directives HPF. Notre but était de simplifier la tâche de l'utilisateur en lui indiquant la syntaxe d'HPF et en écrivant pour lui les directives après qu'il ai rempli quelques champs donnant les valeurs de certaines variables. Les directives qui pouvaient être ajoutées au code grâce à HPFize étaient `TEMPLATE`, `PROCESSORS`, `ALIGN`, `DISTRIBUTE`, `FORALL` et `INDEPENDENT`. La figure 3.10 montre deux fenêtres d'HPFize V1.

### 3.4.7 ALASca

La seconde version d'HPFize appelée ALASca<sup>2</sup> étendait la première puisqu'elle permettait d'insérer automatiquement des directives de distribution en fonction des routines de calcul appelées dans le code. Notre but était de pouvoir transformer des codes Fortran contenant des appels à des routines BLAS et LAPACK :

- soit en un code Fortran contenant des appels à ScaLAPACK et aux routines de distribution et redistribution associées,

<sup>2</sup>Automatic LAPACK to ScaLAPACK translation, outil développé avec C. Randriamaro et S. Domas.

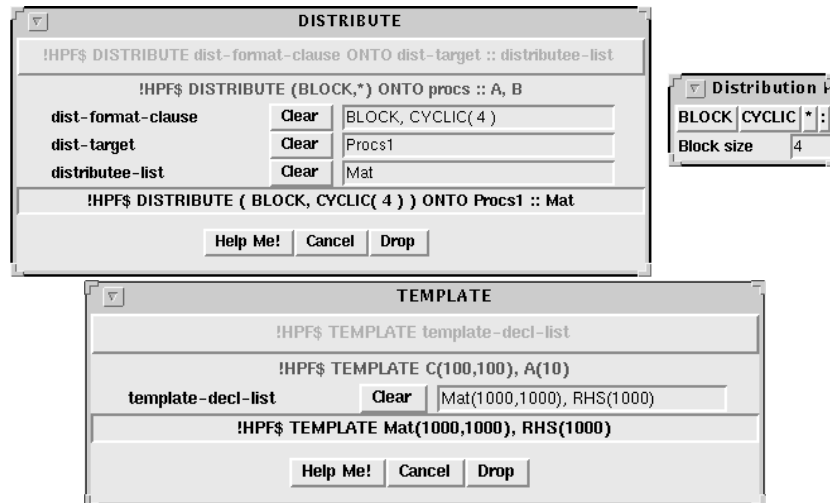


FIG. 3.10 – Deux fenêtres d’HPFize V1 (distribution et templates).

– soit en un code HPF contenant des appels en une interface HPF pour ScaLAPACK et aux directives de distribution et de redistribution du langage.

Les outils utilisés dans ce logiciel sont les algorithmes de distribution automatique présentés dans la section 3.3 et les modélisations de routines de calcul développées par S. Domas.

La figure 3.11 présente la fenêtre XEmacs contenant le programme séquentiel avant transformation ainsi que la fenêtre de visualisation de graphe d’appel. Les appels à des routines BLAS ou ScaLAPACK sont mis en évidence. La figure 3.12 présente le résultat de la transformation du programme avec aLASca avec l’ajout des appels de routines de distribution, de redistribution et de calcul (avec ScaLAPACK). Remarquez également le remplissage automatique des descripteurs de matrices ScaLAPACK. Dans le graphe d’appel, on voit l’ajout de boîtes de redistribution (une distribution est considérée comme étant une redistribution d’une grille contenant un processeur vers la grille cible de calcul). Enfin, la figure 3.13 montre le résultat de la transformation du programme original en un programme HPF. Les algorithmes utilisés sont ceux présentés dans la section 3.3.4.

### 3.4.8 Interface de compilation et d’exécution

Collaborant depuis plusieurs années avec T. Brandes (GMD SCAI) autour de la compilation d’HPF et les pipelines de calcul dans HPF, nous avons naturellement beaucoup travaillé avec Adaptor qui était à l’époque le meilleur du marché avec `pgphpf` de PGI [137]. Dans le cadre d’EuroTOPS, nous avons travaillé avec la société NAS [208] et leur compilateur FortranPlus. Tous ces compilateurs possédaient des appels et des paramètres de compilation différents. Notre idée a donc été de fournir à l’utilisateur une interface simple de compilation<sup>3</sup> lui permettant de donner des paramètres par défaut pour chaque compilateur et de choisir le compilateur grâce à un simple menu. Ainsi, en changeant quelques paramètres dans une fenêtre Tcl/Tk, on pouvait passer d’une architecture à une autre ou essayer plusieurs compilateurs différents.

### 3.4.9 Conclusions

TRANS TOOL a été validé et démontré dans le cadre du projet européen EuroTOPS. Couplé à un outil de nettoyage de code (Foresys de SIMULOG) et à des bibliothèques à hautes perfor-

<sup>3</sup>Développée avec J.-C. Mignot.

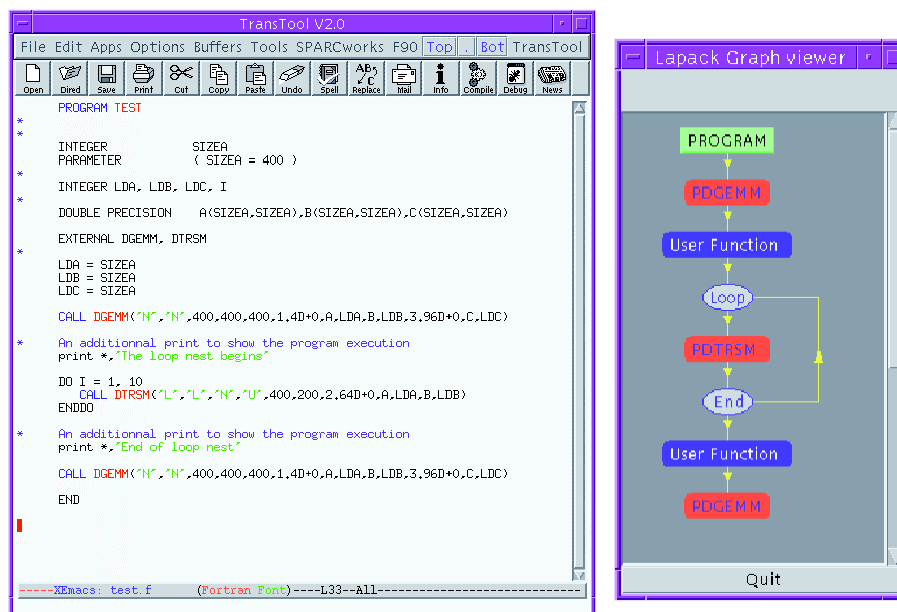


FIG. 3.11 – L’interface d’aLASca intégrée à TransTool montrant un code Fortran séquentiel avec son graphe de succession.

mances comme ScaLAPACK, il pouvait se montrer performant et efficace pour la parallélisation de programmes de tailles importantes.

Par contre, les résultats assez décevants des compilateurs HPF du moment et le désintérêt des utilisateurs pour ce langage nous ont conduit à arrêter ce projet en 1999. TRANSTOOL est donc resté à l’état de prototype évolué<sup>4</sup>. Cependant, l’expérience dégagée dans ce projet quant au développement de « gros » logiciels en collaboration avec de nombreuses équipes m’a été bénéfique et m’a servi pour la conduite des projets suivants tels que SCILAB// et DIET.

### 3.5 Conclusions et perspectives

Les travaux présentés dans ce chapitre autour de la parallélisation de programmes Fortran utilisant des structures de données régulières m’ont permis de comprendre les attentes des utilisateurs en termes de gain de performances mais surtout de simplicité de développement de programme. En effet, il n’est pas forcément important d’atteindre de très hautes performances si le coût de développement de l’application s’avère prohibitif. Nous nous sommes aperçus que c’était l’un des gros problèmes d’un langage comme HPF. Utilisé simplement, il donnait des programmes très portables, par contre, les performances s’avéraient très décevantes. Utilisé de manière très fine (en utilisant des routines extrinsèques avec du passage de message ou des bibliothèques parallèles de calcul), on pouvait obtenir de très bonnes performances mais le code perdait sa portabilité et le coût de mise au point s’avérait totalement prohibitif. De plus, l’extrême difficulté du traitement des structures de données irrégulières dans HPF (malgré le développement de quelques extensions, notamment dans HPF-2) lui ont fermé les portes de l’industrie puisque la plupart des gros codes de simulation manipulent de telles matrices. Enfin, l’évolution des architectures vers des grappes de machines SMP a conduit à l’étude de nouveaux paradigmes de programmation utilisant à la fois du passage de messages (entre

<sup>4</sup>Au grand désespoir d’Y. Robert!

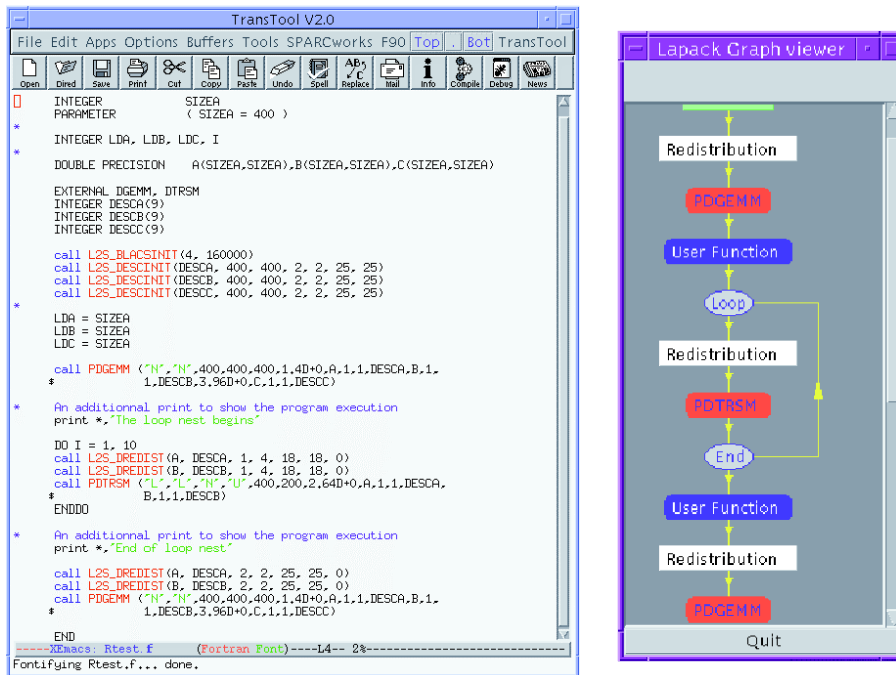


FIG. 3.12 – L'interface d'aLASca intégrée à TransTool montrant le code Fortran parallèle correspondant à la figure 3.11, avec son graphe de succession.

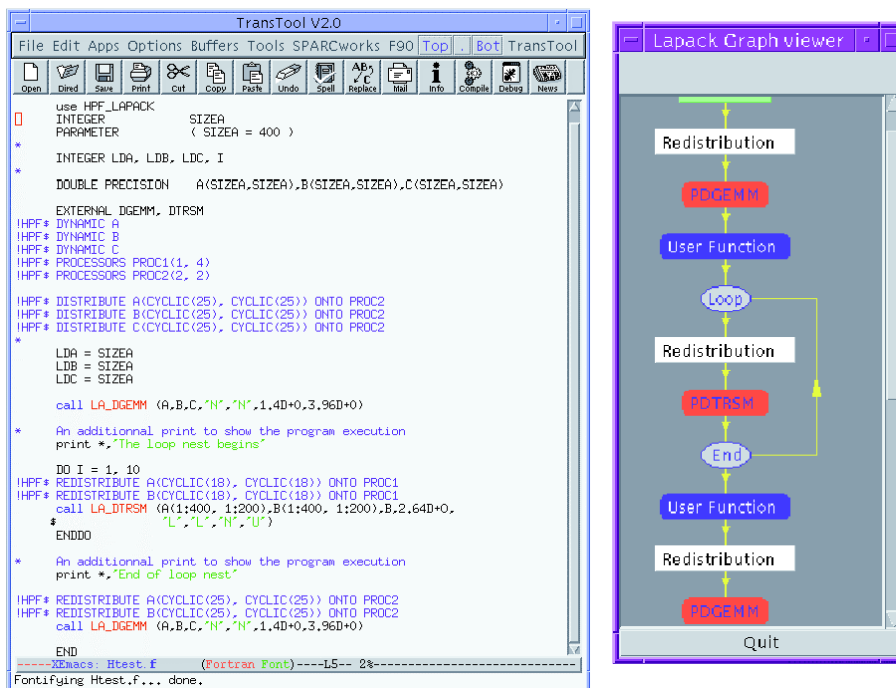


FIG. 3.13 – L'interface d'aLASca intégrée à TransTool montrant le code HPF correspondant à la figure 3.11, avec son graphe de succession.

les grappes) et des bibliothèques de threads (à l'intérieur des machines SMP). C'est le langage OpenMP [215] qui tente maintenant de récupérer les développeurs d'applications numériques de grandes tailles.

Les développements faits autour de TRANSTOOL trouvent maintenant leur application pour la parallélisation du logiciel SCILAB. En effet, nous comptons réutiliser une partie de cet environnement pour la compilation de scripts SCILAB sur plates-formes hétérogènes. Les premiers travaux consistent à étendre NESTOR pour qu'il puisse prendre en entrée un script SCILAB, lui ajouter des directives et commandes de parallélisation de SCILAB// pour enfin rendre un script parallèle.



# Chapitre 4

## Scilab<sub>//</sub>-DIET ou de Scilab aux environnements de calcul distribués

Ce chapitre présente mes travaux sur la parallélisation du logiciel SCILAB et sur le développement de serveurs de calcul dans un environnement de metacomputing.

Ces travaux s'échelonnent de Juin 1997 à aujourd'hui.

*Travaux effectués en collaboration avec E. Caron, S. Chaumette, S. Contassot-Vivier, E. Fleury, C. Gomez, M. Goursat, D. Lazure, F. Lombard, M. Quinson, J.-M. Nicod, L. Philippe, P. Ramet, J. Roman, F. Rubi, S. Steer, F. Suter et G. Utard.*





## 4.1 Introduction

SCILAB [134], développé à l'INRIA Rocquencourt dans le projet Métalau, est un outil de type MATLAB qui permet d'effectuer de nombreuses opérations comme de l'automatique, de l'algèbre linéaire, du traitement du signal, de l'analyse et de l'optimisation de réseaux et de la résolution de systèmes non-linéaires. Il y a plusieurs raisons à son succès : (1) le langage est simple et facile à apprendre (surtout pour les utilisateurs de MATLAB!), (2) SCILAB contient des centaines de fonctions dans les domaines cités précédemment, (3) il possède une interface graphique, (4) il utilise un langage de haut niveau avec une syntaxe proche de celle de Fortran 90 pour les notations matricielles. On peut manipuler simplement des matrices avec des opérations aussi diverses que la concaténation, l'extraction et la transposition. SCILAB peut manipuler des structures de données telles que des matrices denses et creuses, des polynômes, des nombres rationnels, des listes, les chaînes de caractères, etc. Des opérations compliquées qui requièrent habituellement des dizaines de lignes de C ou Fortran peuvent être écrites en quelques lignes de SCILAB. (5) On peut facilement développer nos propres modules. (6) SCILAB peut être facilement interfacé avec d'autres langages comme C ou Fortran et des outils de calcul symbolique comme Maple ou Mupad. (7) SCILAB peut générer des programmes Fortran et enfin (8) SCILAB est un logiciel du domaine public.

L'un des inconvénients principaux de l'interprétation est qu'un tel langage ne peut donner des performances comparables à un langage compilé comme C ou surtout Fortran. En revanche, la perte de performance (entre une et dix fois) doit être opposée à la simplicité de programmation. Tous les avantages d'un outil comme MATLAB se retrouvent dans SCILAB. Il est relativement simple de modifier le code, de modifier la taille des données (ce qui ne nous simplifie pas la tâche!), d'afficher des variables ou de modifier interactivement la formulation d'un problème. Cette dernière caractéristique simplifie grandement le prototypage de codes. De plus, pour les applications avec des durées d'exécution importantes, le surcoût d'interprétation devient négligeable. SCILAB doit être considéré comme un « vrai » langage permettant le développement d'applications diverses. Le problème se pose toutefois si l'application codée en SCILAB manipule des structures de données de très grandes tailles ou si les temps d'exécution deviennent prohibitifs. Le programmeur n'a plus qu'à recoder son application vers un langage qui pourra bénéficier du parallélisme comme C ou Fortran puis paralléliser l'application de manière classique. Le temps de développement (et le temps d'apprentissage) pour passer d'un code de haut niveau séquentiel à un code parallèle efficace étant connu comme important, l'utilisateur n'aura que deux alternatives, avoir les performances ou la convivialité et la facilité de programmation.

Mes travaux autour de SCILAB remontent au projet Paralin (1997). Ce projet INCO-DC avait pour but de paralléliser un certain nombre d'applications dans le domaine de l'énergie au Chili. L'une des applications consistait à optimiser un réseau électrique à l'aide d'un code écrit dans l'équipe de Frédéric Bonnans de l'INRIA Rocquencourt. Les premiers prototypes de ce code étaient à l'époque écrits en SCILAB. Nos travaux sur sa parallélisation étaient effectués en même temps que son développement lui-même et il nous est vite apparu évident que, plutôt que de paralléliser une version du code traduite dans un langage plus adéquat comme Fortran ou C, il serait plus pertinent (et certainement plus efficace en terme de temps de développement) de paralléliser l'outil SCILAB lui-même. Nous nous sommes donc lancés dans l'étude d'une interface entre SCILAB et la bibliothèque de communication « standard » de l'époque, PVM. Par la suite, le challenge de la parallélisation de l'outil dans son ensemble, avec plusieurs approches suivant le niveau de connaissance de l'utilisateur, nous est apparu intéressant et porteur. Ensuite, nous avons obtenu un financement de l'INRIA sous forme d'une Action de Recherche Coopérative appelée OURAGAN<sup>1</sup> dont le but était de développer diverses approches pour la parallélisation de SCILAB. Les partenaires de cette ARC étaient dans un premier temps le projet Méta-2<sup>2</sup> de l'INRIA Rocquencourt, le projet Résédas de l'INRIA Lorraine et le LaBRI. Par la suite, deux autres

<sup>1</sup>Outils pour la résolution de problèmes numériques de grande taille, URL : <http://www.ens-lyon.fr/~desprez/OURAGAN/>.

<sup>2</sup>Devenu Métalau au cours de l'ARC.

équipes nous ont rejoint, l'équipe Paladin du LaRIA<sup>3</sup> et l'équipe SDRP du LIFC<sup>4</sup>.

Ce chapitre présente donc deux environnements, étroitement liés, SCILAB// dont le but est de paralléliser SCILAB avec diverses approches et DIET dont le but est de développer un ensemble d'outils pour le développement d'applications de type ASP (*Application Service Provider*).

## 4.2 SCILAB//

Notre objectif consiste à apporter l'accès à de bonnes performances et à de grandes capacités mémoire aux utilisateurs de SCILAB. Notre but est également de cacher au maximum l'utilisation du parallélisme dans un tel environnement. L'utilisateur moyen de SCILAB l'a choisi principalement pour son langage de haut niveau et son degré d'abstraction. Nous ciblons principalement les opérations matricielles car ce sont elles qui ont généralement le plus besoin de grandes capacités de traitement.

Nous avons trois types d'utilisateurs cibles. Le premier type d'utilisateur est un gourou du parallélisme. Il sait comment écrire des programmes en utilisant le passage de messages et des bibliothèques parallèles. Il souhaite conserver un contrôle sur la manière dont les données et les calculs sont distribués. Pour ce type d'utilisateur, nous ne fournissons « que » des interfaces vers les bibliothèques de calculs et de communications. Le deuxième type d'utilisateur est un scientifique. « *Calcul para-quoi ? ? Je ai juste besoin que d'une station de travail à 45 Gflops et avec 30 Go de mémoire. Est-ce que vous pouvez avoir un accès à une telle machine ?* » Tout est caché dans SCILAB. L'outil décide lui-même s'il doit (ou pas) redistribuer les données, démarrer des nouveaux processus (et où) et ainsi de suite. Cela est effectué en utilisant la surcharge d'opérateurs. Enfin, nous ciblons un troisième et dernier type d'utilisateur qui est intermédiaire. Ce type d'utilisateur veut avoir un accès le plus transparent possible aux bibliothèques mais il est concerné par les performances de son code. Il possède de bonnes connaissances en calcul parallèle et il souhaite programmer ses applications en utilisant le passage de messages et des bibliothèques de calcul, mais tout en profitant des capacités de SCILAB et du niveau d'abstraction de son langage.

La figure 4.1 montre l'architecture générale de SCILAB//. Notre première approche consiste à interfacer SCILAB avec PVM et MPI pour pouvoir effectuer du passage de messages entre processus SCILAB (A sur la figure). Puisque nous ciblons les opérations matricielles, nous fournissons des interfaces pour des bibliothèques parallèles comme SCALAPACK et son prototype *out-of-core* (B sur la figure). Notre seconde approche consiste à utiliser des serveurs de calculs basés sur NETSOLVE [58] (C sur la figure). De plus, nous avons amélioré NETSOLVE en bien des points. Nous lui avons ajouté la persistance des données sur les serveurs, une meilleure évaluation des paires (routines, machines) et nous avons modifié ses couches de communication. Ensuite, nous lui avons ajouté une interface vers un solveur creux, PASTIX (D sur la figure) et un outil de visualisation de matrices distribuées, VISIT (E sur la figure).

Remarquez que les développements présentés dans ce chapitre s'appliquent à d'autres environnements comme MATLAB [192], Mathematica [269] ou Octave [118].

### 4.2.1 Pourquoi un MATLAB distribué ?

À en croire Cleve Moler, créateur de MATLAB, dans son article de *Matlab News and Notes* [197], parallélisme et MATLAB ne font pas bon ménage (*Why there isn't a parallel MATLAB*) ! D'après lui, il y a trois raisons principales d'éviter de développer une version parallèle de MATLAB. Premièrement, le modèle de mémoire des machines parallèles (i.e. à mémoire distribuée) conduit à faire de fréquents aller-retour entre la station de travail où s'exécute l'environnement interactif et la machine parallèle. Ces déplacements de données sont en général plus coûteux que les opérations elles-mêmes. La deuxième raison est la granularité des opérations MATLAB.

<sup>3</sup>Laboratoire de Recherche en informatique d'Amiens.

<sup>4</sup>Laboratoire d'Informatique de Franche-Comté.

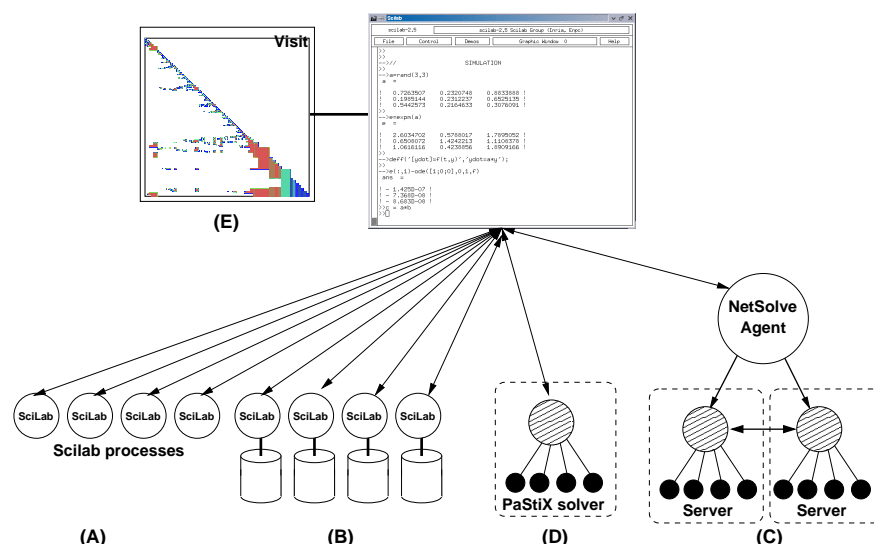


FIG. 4.1 – Architecture générale de SCILAB//.

Cleve Moler dit que le temps d'exécution d'une application est perdu dans des opérations qui ne sont pas parallélisables telles que l'analyse syntaxique, l'interprétation et les routines graphiques. Enfin, il donne un dernier argument économique sur le peu d'intérêt des utilisateurs de MATLAB pour l'utilisation des machines parallèles.

En revanche, le nombre de projets [101] autour de la parallélisation de cet outil tend à prouver le contraire. Tous ces chercheurs se seraient-ils trompés de problème ? Posons-nous la question du choix d'un environnement tel que MATLAB pour le développement d'applications numériques. Souvenons-nous des chapitres précédents et de la complexité à utiliser des routines de calcul et plus encore à utiliser des versions parallèles de ces routines. De nombreuses recherches sur les langages ont tenté de fournir aux développeurs d'applications numériques des langages simples d'utilisation, faciles à déboguer et à compiler. L'opposé de cette simplicité est certainement représenté par le langage C qui permet certes de tout faire (et d'optimiser le code « à la main ») mais il est bien loin des équations. Le langage MATLAB représente donc certainement une bonne alternative aux langages classiques tels que C ou Fortran en étant plus proche de la formalisation mathématique des problèmes. On peut tout simplement comparer le code Fortran qu'il faut écrire pour obtenir un produit de matrices en BLAS

```
DGEMM('Y', 'N', M, N, K, 1.0, A, LDA, B, LDB, 1.0, C, LDC)
```

à son équivalent en MATLAB

```
c = a*b.
```

Cet exemple est bien sûr extrême mais il va de soi que la mise au point de programmes mathématiques sera bien plus aisée en MATLAB qu'en Fortran ou plus encore qu'en C. Par contre, ce type d'environnement pêche encore par ses performances et c'est là que le parallélisme peut faire quelque chose s'il est bien employé. Par contre, un environnement de type MATLAB rencontrera souvent des problèmes de tailles de données si l'application de simulation est trop précise.

## 4.2.2 Travaux connexes autour de la parallélisation de MATLAB

L'idée d'accéder au calcul parallèle depuis MATLAB n'est pas nouvelle. Il existe deux approches classiques pour y parvenir. La première approche consiste à compiler les scripts MATLAB vers

un autre langage (comme Fortran [191, 237] ou C [115]) puis ensuite y appliquer des optimisations classiques issues de la parallélisation automatique et de la vectorisation et pourquoi pas utiliser au passage des bibliothèques numériques à haute performance, parallèles ou non [69, 115, 227]. Les avantages de cette approche sont ses grandes performances et l'utilisation de techniques de compilation sophistiquées. En revanche, l'interactivité est perdue, l'inférence des types est un problème compliqué et la mise à jour du programme nécessite une recompilation. De plus, si l'application utilise des packages propres à MATLAB alors la compilation devra trouver dans les bibliothèques disponibles le moyen de réaliser les mêmes opérations. Le tableau 4.1 donne les principaux projets autour de la compilation de scripts MATLAB.

Nom	Langage généré	Bibliothèques
CONLAB Compiler	C	PICL (com), BLAS, LAPACK, Motif
FALCON	FORTRAN 90 and pC++	-
Menhir	FORTRAN, C	LAPACK and ScaLAPACK
Otter	C	MPI (com), ScaLAPACK, parallel FFT
MATCH	C, RTL VHDL	specific
Paradigm	FORTRAN (?)	ScaLAPACK
ParAL	C	PVM(com), specific
RTEexpress	C	MPI(com)

TAB. 4.1 – Projets autour de la parallélisation de langages de type MATLAB.

La seconde approche conserve l'interactivité de MATLAB et fournit des extensions pour le parallélisme. Cette approche peut être elle-même divisée en deux sous-approches. La première idée consiste à dupliquer des instances de l'outil lui-même (ou d'une partie de cet outil) sur chaque nœud de la machine cible [219, 256]. Cette approche a un avantage : le processus maître n'a qu'à envoyer des commandes MATLAB classiques aux processus de travail qui les interprètent à la réception avant de retourner le résultat. Son principal inconvénient est bien entendu les pertes de performances durant les multiples interprétations, des processus lourds et la nécessité d'interfacer toutes les bibliothèques qui doivent être ajoutées à l'outil. De plus, il faut que MATLAB soit disponible sur tous les processeurs de (ou des) machine(s) cible(s). La seconde sous-approche consiste à utiliser un serveur de bibliothèques qui attend des commandes [59, 153, 200]. Le tableau 4.2 donne les projets principaux qui conservent l'interactivité de MATLAB.

Nom	Approche	Bibliothèques de communications	Bibliothèques de calcul
Netsolve	Serv.	-	LAPACK, ScaLAPACK, Petsc
PPServer	Serv.	Sockets et MPI	ScaLAPACK, S3L, PARPACK, PETSc
Multimatlab	Dup.	-	-
DP Toolbox	Dup.	PVM	-
Paramat	Dup.	-	-
Parallél Toolbox	Dup.	PVM	-
Matpar	Serv.	PVM	BLAS, BLACS, PBLAS, ScaLAPACK
PSI	Serv.	MPI	PLAPACK
PMI	Dup.	MPI	-

TAB. 4.2 – Projets qui conservent l'interactivité de MATLAB.

Une dernière approche mélange les techniques de compilation et de support exécutif. Les scripts MATLAB sont compilés dans un langage intermédiaire simple qui est à son tour exécuté sur une machine virtuelle (à la Java). Cette machine virtuelle MATLAB du projet Match offre un environnement d'exécution à hautes performances pour des plates-formes hétérogènes [29].

### 4.2.3 Duplication de processus SCILAB

Notre première approche a consisté à ajouter à SCILAB la capacité d'effectuer ses opérations sur des machines distantes. Il a été alors nécessaire de pouvoir lancer des processus SCILAB « classiques » sur des processeurs distants et de les faire communiquer.

#### 4.2.3.1 Interfaces de passage de messages

Nos premiers développements ont donc consisté à offrir aux utilisateurs SCILAB une interface de passage de messages classique. Ceci a été effectué dans un premier temps en utilisant PVM [131]. Cette interface permet à un utilisateur de développer des programmes parallèles tout en bénéficiant de l'ensemble des fonctionnalités de SCILAB. Nous avons choisi PVM car cet environnement permet de démarrer dynamiquement des processus après le démarrage du programme, ce qui n'est pas le cas de MPI [246]. Nous avons également ajouté une interface MPI puisque les implémentations de ce standard sont généralement plus performantes. Cependant, l'utilisation de MPI dans une session SCILAB implique que l'utilisateur doit décider au départ le nombre de processus maximum qu'il souhaite démarrer.

Ces interfaces de passage de messages offrent des fonctions de bas niveau pour obtenir les meilleures performances mais leur utilisation est plutôt réservée à des utilisateurs « experts ». Une instance de SCILAB est capable de communiquer et d'interagir avec d'autres instances de SCILAB et l'utilisateur peut envoyer (et recevoir) des données de tous types (y compris des matrices, des listes, des fonctions, etc.) en utilisant des appels à des fonctions PVM (ou MPI). Une instance de SCILAB peut par exemple envoyer une sous-matrice d'une matrice  $A$  (et pas forcément des blocs consécutifs) en utilisant la commande

```
pvm_send(dest, A(1 : 2 : N, :), tag)
```

ou on peut définir une fonction  $f$  et l'envoyer à une autre instance qui sera alors capable de l'exécuter sur ses propres données :

```
deff('[x]=f(y)', 'x = 1/y'), pvm_send(dest, f, tag).
```

Les performances de ces interfaces sont équivalentes aux performances des bibliothèques utilisées dans des programmes C ou Fortran lorsqu'on envoie une matrice complète et le surcoût d'interprétation est négligeable [92]. Par contre, lorsqu'on envoie des sous-matrices, il y a un surcoût naturel de copie des données dans un buffer temporaire. Par exemple, l'expression `send(A(1 : 2 : 100, 2 : 2 : 100), ...)` enverra une sous-matrice de taille  $50 \times 50$  qui n'est pas contigue en mémoire.

Ces premières interfaces offrent un moyen d'exécuter simplement des programmes parallèles sans perdre le pouvoir d'expression et les fonctionnalités de SCILAB.

#### 4.2.3.2 Interface pour SCALAPACK

Afin d'obtenir une bonne portabilité et une bonne efficacité, SCILAB// intègre également des interfaces pour des bibliothèques parallèles d'algèbre linéaire comme les PBLAS, SCALAPACK et son prototype *out-of-core*. L'utilisateur peut alors distribuer ses matrices sur l'ensemble des processus SCILAB démarrés et exécuter des routines parallèles sur la grille virtuelle ainsi créée. Cette approche reste toutefois du domaine de l'« expert » de l'algèbre linéaire parallèle et des bibliothèques intégrées puisque l'utilisateur doit spécifier les distributions et redistributions éventuelles dans son code. Par contre, nous lui simplifions la tâche en ayant des arguments par défaut ou en appelant automatiquement les fonctions pour les réels double précision ou les complexes suivant le type des données en entrée.

Le code donné dans la figure 4.2 illustre l'utilisation des routines BLACS et donne un exemple d'utilisation de la fonction `pblas_gemm` qui permet de calculer  $C = \alpha A * B + \beta C$ . Ce code simple est exécuté par tous les processus SCILAB dans un mode SPMD. Le script suivant commence par initialiser une grille de  $2 \times 2$  processeurs. Une fois que l'initialisation des zones mémoire locales pour le stockage des matrices  $A$ ,  $B$  et  $C$  est effectuée sur chaque processeur, l'appel à la routine

`pblas_gemm` peut être exécuté sur chaque processeur. L'API utilisée dans SCILAB// est très proche de celle de SCALAPACK. Par contre, certains paramètres peuvent être « omis » et de ce fait remplis avec des valeurs par défaut (comme par exemple le fait de transposer ou pas les matrices, les indices de ligne et/ou de colonne, etc.).

---

```
[mypnum,nprocs] = blacs_pinfo();
blacs_setup(4);
icontxt = blacs_get();
ictxt = blacs_gridinit(icontxt,'R',2,2);
M=1000;K=1000;N=1000;MB = M/2; NB=K/2;
desc_A = sca_descinit(ictxt,M,K,MB,NB,0,0,M);
desc_B = sca_descinit(ictxt,K,N,MB,NB,0,0,K);
desc_C = sca_descinit(ictxt,M,N,MB,NB,0,0,K);
A = rand(M/2,K/2)
B = rand(K/2,N/2)
C = zeros(M/2,N/2)
pblas_gemm(M,N,K,1,"A",desc_A,"B",desc_B,0,"C",desc_C)
```

FIG. 4.2 – Utilisation de PDGEMM dans SCILAB//.

---

Les utilisateurs de SCILAB étant généralement des numériciens non-spécialistes du parallélisme, leur but est tout simplement de transposer leurs équations en un langage simple mais malgré tout de conserver de bonnes performances et des tailles de problèmes raisonnables. Afin de fournir à ces utilisateurs de bonnes performances pour leurs opérations d'algèbre linéaire sans avoir à gérer des échanges de messages ou à avoir à utiliser des interfaces de bibliothèques comme SCALAPACK, nous avons décidé d'ajouter un nouveau type distribué dans SCILAB//. Du point de vue de l'utilisateur, le fait qu'une matrice soit distribuée ou pas changera peu la manière d'écrire des programmes SCILAB. Les seules commandes additionnelles sont :

**scip\_init** : initialise la grille, c'est-à-dire le nombre de processeurs qui seront utilisés dans la session. Un fichier de configuration peut être utilisé pour spécifier la configuration par défaut (nombre d'hôtes et nom des machines) ;

**scip\_init\_dist** : initie une distribution spécifique si l'utilisateur ne veut pas utiliser la distribution par défaut ;

**scip\_distribute** : distribue une matrice scalaire. C'est la routine principale qui distribuera une matrice, définie dans la console SCILAB ou rangée dans le système de fichiers, aux autres processus SCILAB qui ont été au préalable démarrés avec la fonction `scip_init`.

Nous avons ensuite surchargé les fonctions et opérations SCILAB classiques pour qu'elles fonctionnent avec le type distribué. Ainsi, les opérations effectuées sur des matrices distribuées seront effectuées en parallèle. Pour l'instant, toutes les opérations qui possèdent leur équivalent dans les deux bibliothèques parallèles PBLAS et SCALAPACK ont été surchargées. Le point important est que l'utilisateur peut toujours utiliser les notations matricielles de SCILAB « classiques » pour écrire des programmes parallèles.

Bien entendu, toutes les fonctions SCILAB travaillant sur les types scalaires n'ont pas été surchargées. Lorsqu'une opération est effectuée sur une donnée distribuée et qu'il n'y a pas de fonction parallèle qui lui corresponde, l'utilisateur peut choisir entre plusieurs modes différents : le premier (et le plus simple) consiste à générer une erreur ; le second consiste à systématiquement ramener la matrice distribuée dans la console SCILAB (si elle peut rentrer en mémoire) et exécuter l'opération sur la matrice locale correspondante. Un autre problème peut survenir lorsqu'une opération est effectuée entre une donnée distribuée et une donnée non-distribuée. Ici encore, l'utilisateur peut choisir parmi différents modes : générer une erreur ; récupérer la matrice localement ou bien propager la distribution sur la donnée de type scalaire. Ce dernier choix nous permet de ne distribuer qu'une matrice importante au départ du programme et de laisser ensuite l'interpréteur propager automatiquement la distribution sur les autres types

scalaires. L'exemple donné dans la figure 4.3 montre comment effectuer une multiplication de matrices très simple sur une grille de  $P \times Q$  processus. Dans cet exemple, la distribution utilisée est cyclique par blocs dans les deux dimensions avec une taille de blocs fixée à  $MB \times NB$  (mais l'utilisateur aurait pu utiliser des valeurs par défaut). Ensuite, les matrices scalaires  $A$  et  $B$  sont distribuées et le produit de matrices est effectué en utilisant tout simplement le symbole  $*$  dans la console SCILAB.

```

CTXT = scip_init(P,Q);
DIST = scip_init_dist("CC",0,0,CTXT(3),MB,NB);
A = rand(M,N); B = rand(M,N);
MatA = scip_distribute("A", DIST);
MatB = scip_distribute("B", DIST);
Res = MatA(1:1000,1:500)*MatB(1:500,1:1000);
size(Res)
ans = !   1000.   1000. !

```

FIG. 4.3 – Produit de matrice parallèle avec surcharge de l'opérateur  $*$  dans SCILAB.

Le tableau 4.3 liste les fonctions SCILAB qui supportent la surcharge d'opérateur pour les matrices distribuées. Comme nous pouvons le remarquer dans l'exemple précédent, la récupération et la modification de sections de tableaux fonctionnent également de manière transparente sur les matrices distribuées.

Fonction	Description	Fonction	Description
$+$ , $-$ , $*$ , $.*$ , $./$	Op. matricielle binaire classique	size	Taille d'un objet
chol	Factorisation de Cholesky	hess	Forme d'Hessenberg
inv	Inversion de matrice	linsolve	Résol. de système linéaire
lu	Factorisation $LU$	qr	Factorisation $QR$
rcond	Nombre condition inverse	schur	Factorisation de Schur
spec	Valeurs propres	svd	Décomp. en valeurs sing.

TAB. 4.3 – Opérations surchargées pour les matrices distribuées dans SCILAB//.

La figure 4.4 montre les performances obtenues en utilisant dans la console SCILAB l'opérateur standard  $*$  sur des matrices distribuées. L'abscisse représente la taille des matrices et l'ordonnée le temps nécessaire pour exécuter deux multiplications de matrices  $Res=A*B*C$  avec  $A$ ,  $B$  et  $C$  des matrices carrées sur une Origin 2000 de SGI. La courbe avec des  $- + -$  présente le temps séquentiel (avec l'opérateur scalaire  $*$  de SCILAB), celle avec les  $- \times -$  présente les temps sur 4 processeurs et enfin celle avec  $(- * -)$  est obtenue avec 16 processeurs. À partir de matrices de taille  $200 \times 200$ , il est plus intéressant d'utiliser un produit de matrice parallèle. Le surcoût de traitement de SCILAB n'est pas tellement un problème lorsque la dimension du problème devient assez importante.

Par ailleurs, SCILAB// possède des extensions pour une version optimisée de SCALAPACK *out-of-core*. Ces extensions sont développées au LaRIA (Université d'Amiens) et sont décrites dans [56, 57] ainsi que dans le chapitre H du document annexe.

#### 4.2.4 Approche Serveurs de calcul

#### 4.2.5 Travaux reliés

Les problèmes de très grande taille peuvent désormais être résolus via Internet grâce aux environnements de metacomputing [125]. Plusieurs approches co-existent telles que les langages orientés objet [185], les environnements à base de passage de messages [203, 216], les boîtes à

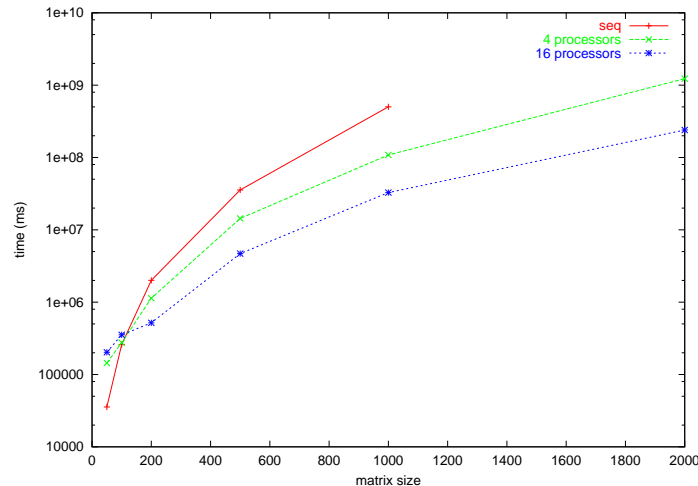


FIG. 4.4 – Performances du produit de matrices en utilisant la surcharge d’opérateur distribué \* dans Scilab//.

outils [126], les environnements de calcul global [212, 271], ceux basés sur le Web [158, 164, 266], etc.

On peut également effectuer des calculs à distance en utilisant le modèle RPC<sup>5</sup> pour le calcul scientifique [193, 194]. Plusieurs outils fournissent ce type de fonctionnalités comme NETSOLVE [58], NINF [213, 205], NEOS [210, 78], RCS [14] ou même OVM [43] pour une utilisation sur des grappes.

NETSOLVE [58] est un outil de type NES<sup>6</sup> qui permet de démarrer des calculs à distance sur une plate-forme de metacomputing. Il utilise le modèle client-agent-serveurs. Une politique d’équilibrage des charges simple est appliquée pour répartir les calculs entre les différents serveurs. L’agent est chargé d’interroger les serveurs sur leur capacité à résoudre le problème demandé et de mesurer le coût de transfert des données vers les serveurs retenus. NETSOLVE permet d’effectuer des requêtes bloquantes et non-bloquantes et il est interfacé avec les langages C et Fortran ainsi qu’avec MATLAB et une interface JAVA.

#### 4.2.5.1 Optimisation de NETSOLVE

Nos premiers travaux ont consisté à partir de NETSOLVE et à l’améliorer avec nos propres développements. Nous avons réalisé une interface Scilab pour NETSOLVE puis nous avons travaillé sur la persistance de données et sur l’évaluation des performances des serveurs.

Suivant les calculs effectués, il arrive souvent que les résultats d’un calcul soient utilisés par le calcul suivant. NETSOLVE dans sa version de base ne permet pas de laisser les données sur les serveurs après un calcul ni d’effectuer des redistributions entre serveurs lorsque c’est nécessaire. Cela induit des surcoûts de communication importants, surtout lorsqu’il est utilisé sur un réseau de type WAN où les coûts de communication sont très importants. Ce problème a été en partie traité par le séquençement de requêtes [15] mais celui-ci ne permet pas d’utiliser plusieurs serveurs et il nécessite l’utilisation de directives particulières pour signaler le début et la fin d’un séquençement.

Nous avons ajouté plusieurs routines pour déplacer les données entre les serveurs (une ou plusieurs données, en entrée ou en sortie) et pour terminer un serveur. Ces routines peuvent en-

<sup>5</sup>Remote Procedure Call.

<sup>6</sup>Network Enabled Server.



suite être utilisées par un client ou un agent pour optimiser l'ordonnement des tâches sur la plate-forme NES. La figure 4.5 montre le résultat d'une multiplication de matrices complexes entre Nancy et Bordeaux. Dans cette expérience, nous avons utilisé la décomposition suivante :  $C_{r_1} = A_r \times B_r$  (1);  $C_{i_1} = A_r \times B_i$  (2);  $C_{i_2} = A_i \times B_r$  (3);  $C_r = C_{r_1} - C_{r_2}$  (4);  $C_i = C_{i_1} + C_{i_2}$  (5) où  $A_r$  (respectivement  $B_r$  et  $C_r$ ) est la partie réelle d'une matrice complexe  $A$  (respectivement  $B$  et  $C$ ),  $A_i$  (respectivement  $B_i$  et  $C_i$ ) est la partie imaginaire d'une matrice complexe  $A$  (respectivement  $B$  et  $C$ ). Les quatre produits de matrices ont été exécutés sur un nœud du SP2 du LaBRI à Bordeaux alors que les deux additions étaient effectuées localement à Nancy. Les étapes 1 et 2 ainsi que les étapes 3 et 4 peuvent être effectuées en parallèle. Avec la redistribution, les objets  $A_r$ ,  $B_r$ ,  $A_i$  et  $B_i$  ne sont pas renvoyés au client entre les étapes 1, 2 et les étapes 3, 4. Les performances sont 1.77 fois meilleures pour une taille de matrice de dimension 1024 qu'avec la version « classique » de NETSOLVE.

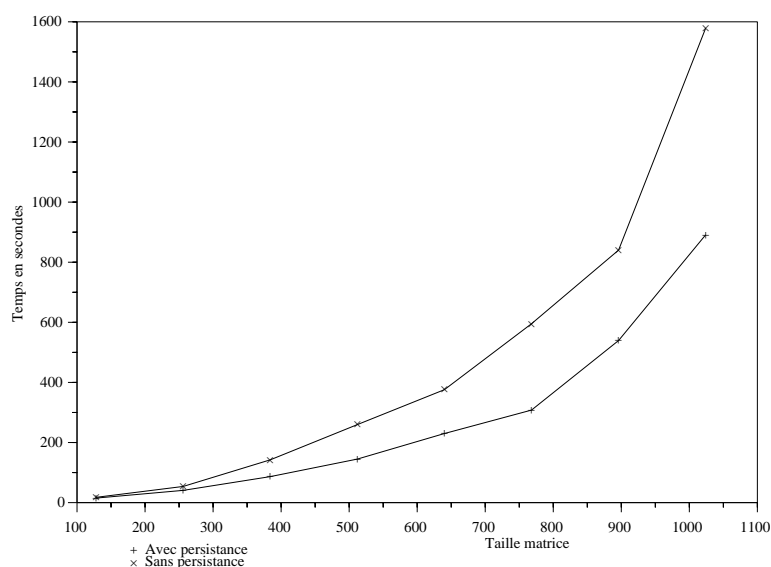


FIG. 4.5 – Produit de matrices complexes entre Nancy et Bordeaux en utilisant NETSOLVE.

La seconde optimisation consiste à améliorer la précision de l'évaluation des paires (routine, machine) et des coûts de communications entre les divers éléments de NETSOLVE (clients, agent, serveurs) avec l'outil FAST. Nous présentons ces travaux dans la section 4.3.2.

#### 4.2.5.2 Interface Pastix

Une interface a également été développée avec le solveur creux PASTIX du LaBRI [143]. Ce solveur de type  $LDL^T$  a pour originalité d'effectuer le placement de ses données et l'ordonnement de ses calculs grâce à une simulation préalable. Ce type de problème ne peut être résolu sur une station du fait des capacités mémoire nécessaires et des temps de calculs prohibitifs. De plus, on constate qu'une utilisation classique requiert une factorisation pour plusieurs résolutions et montre donc l'intérêt de la persistance des données. Cette application se prête donc bien à une approche de type NES. La principale difficulté ici est l'assemblage de la matrice qui doit être effectuée sur le serveur lui-même.

## 4.3 DIET, Distributed Interactive Engineering Toolbox

### 4.3.1 Introduction

En 2000, nous nous sommes tournés plus précisément vers les plates-formes de metacomputing en créant le projet DIET (*Distributed Interactive Engineering Toolbox*<sup>7</sup>). En travaillant avec NET-SOLVE, nous nous sommes aperçus qu'il y avait moyen de l'améliorer et surtout de développer un ensemble d'outils utiles au développement d'applications de type ASP (*Application Service Provider*).

### 4.3.2 SLIM et FAST

L'objectif de l'agent est de choisir un (ou plusieurs) serveur capable de résoudre rapidement un problème demandé par un client.

Pour cela, il doit être capable, dans un premier temps, d'effectuer la liaison entre la représentation du problème dans le client et les routines disponibles dans les serveurs. C'est le but de SLIM<sup>89</sup>. La difficulté principale consiste à trouver une manière uniforme d'exprimer le problème et de décrire les données (types, formes de stockage). NETSOLVE propose son propre langage de description de problème mais il n'est pas standard. NINF possède également son propre IDL sous-forme de prototypes C [193, 251]. CORBA semble aussi être une bonne alternative mais, à notre connaissance, aucun des travaux n'ont abouti pour la définition de services pour le calcul scientifique. Nous sommes donc partis dans un premier temps des descriptions de problèmes de SCILAB et nous regardons maintenant du côté des taxonomies définies pour les bases de données de logiciels scientifiques comme GAMS [40] ou RIB [51]. Nous stockons les informations dans un arbre LDAP [151].

Ensuite, on doit être capable de connaître le coût de résolution d'un problème sur un ou plusieurs serveurs. Ce coût est constitué du coût de calcul lui-même mais aussi des transferts de données entre le client et le serveur choisi et entre les différents serveurs lorsque des données sont issues de calculs précédents. Avec M. Quinson et F. Suter, nous avons donc conçu FAST<sup>10</sup> pour remplir cette fonction importante [98]. FAST utilise le Network Weather Service (NWS [270]), un outil d'évaluation de performances réseau développé à l'Université du Tennessee, Knoxville. Pour les performances de calculs, nous effectuons des benchmarks des différentes routines accessibles dans un serveur pour différentes tailles de problèmes et différents nombres de processeurs puis nous effectuons des approximations polynomiales. Pour les performances réseaux, nous utilisons les senseurs NWS qui scrutent à intervalles réguliers les différents liens que pourront emprunter les données. Enfin, nous examinons l'état de la mémoire et la charge du serveur. Ces données sont soit statiques, soit dynamiques. Je résume ces paramètres dans la table 4.4. Les données statiques sont rangées dans l'arbre LDAP et les données dynamiques sont gérées par NWS. Une description plus complète de cet outil est donnée dans [98].

Nos outils fonctionnent donc comme décrit dans la figure 4.6. L'agent envoie la description du problème à résoudre et des données impliquées à SLIM (1). Celui-ci contacte la base de données et recherche l'ensemble des implémentations susceptibles de résoudre ce problème (2). Cet ensemble est ensuite envoyé à FAST qui prédit le temps d'exécution de chacune des solutions pour chacun des serveurs disponibles (3). Celui-ci récupère les données statiques de la base de données (4) et les données dynamiques de NWS (5). Enfin, ces données sont combinées par FAST en une liste de couples {serveur ; temps estimé} qui est renvoyée à l'application appelante (6).

---

<sup>7</sup>URL : <http://www.ens-lyon.fr/DIET/>

<sup>8</sup>Scientific Libraries Metaserver.

<sup>9</sup>Développé avec N. Viollet.

<sup>10</sup>Fast Agent's System Timer.

	Matériel	Réseau	Calcul
Statique	<ul style="list-style-type: none"> <li>• Mémoire disponible</li> <li>• Vitesse processeur</li> <li>• Traitement par lots</li> </ul>	<ul style="list-style-type: none"> <li>• Bandes passantes théoriques</li> <li>• Latences théoriques</li> <li>• Topologie</li> <li>• Protocoles</li> </ul>	<ul style="list-style-type: none"> <li>• Faisabilité</li> <li>• Temps d'exécution</li> <li>• Espace nécessaire</li> </ul>
Dynamique	<ul style="list-style-type: none"> <li>• Statut (up/down)</li> <li>• Charge</li> <li>• Charge mémoire</li> <li>• Statut queue de batch</li> </ul>	<ul style="list-style-type: none"> <li>• Bande-passante réelle</li> <li>• Latence réelle</li> </ul>	

TAB. 4.4 – Connaissances nécessaires pour l'agent.

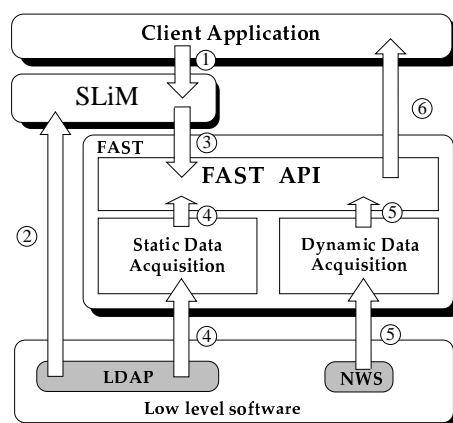


FIG. 4.6 – Vue générale de SLiM et FAST.

Nous avons testé une première version de la bibliothèque et nous l'avons comparée aux prédictions de NETSOLVE dans sa version 1.3. Nous avons utilisé un serveur unique placé sur un nœud de notre grappe de PC PoPC, un agent placé sur une station de travail Sparc quadriprocesseurs qui héberge notre base de données LDAP et enfin un client placé sur une station de travail Sparc Ultra 1. Toutes ces machines sont reliées par un réseau Fast Ethernet. Les résultats sont donnés dans la figure 4.7. La partie (a) compare les prédictions en terme de temps de calcul de NETSOLVE et de FAST au temps mesuré, tandis que la partie (b) compare les temps de communication.

Bien qu'il s'agisse d'une première implémentation de FAST, ces résultats sont déjà très encourageants. La partie (a) montre clairement les avantages qu'il y a à utiliser une modélisation des performances ne dépendant pas seulement du problème à traiter mais aussi de la machine sur laquelle on l'exécute. La partie (b) démontre quant à elle les bénéfices potentiels d'un outil de surveillance du réseau spécialisé comme NWS.

### 4.3.3 Composants de l'architecture DIET

Comme nous l'avons vu précédemment, les approches NES utilisent une architecture à un niveau de hiérarchie dans laquelle un ensemble de clients converse avec un agent qui lui-même échange des informations avec des serveurs accessibles sur le réseau. Il n'y a pas de notion de hiérarchie dans cette architecture et les serveurs sont traités de la même manière quelle que soit leur position dans le réseau. Si au niveau d'une petite configuration, cette approche est suffisante, cela n'est pas le cas dès que l'on souhaite effectuer des opérations à grande échelle.

Notre architecture cible est le réseau VTHD qui connecte les unités de recherche INRIA par un

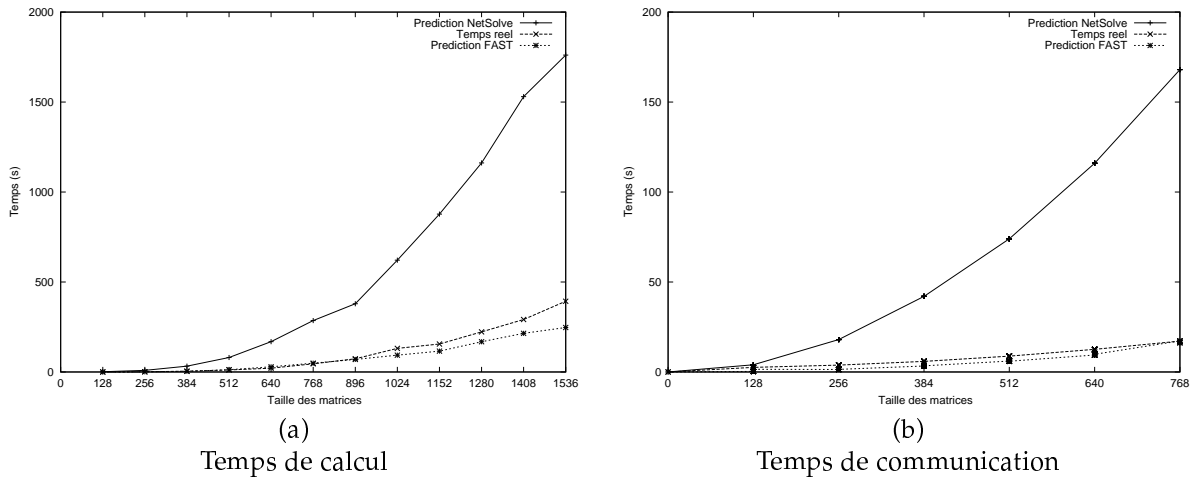


FIG. 4.7 – Comparaison des prédictions de FASTet de NETSOLVE 1.3 par rapport au temps mesuré.

réseau à 2.5 Gb/s. Notre architecture cible est donc fortement hiérarchique puisque le réseau VTHD connecte les UR INRIA entre elles, qui elles-mêmes possèdent à l'intérieur de leurs réseaux propres des grappes de machines connectées par des réseaux plus ou moins rapides. Les machines d'où sont lancés les calculs sont soit directement connectées au réseau VTHD ou simplement connectées par les réseaux internes des laboratoires ayant accès à cette plate-forme. Dans DIET, un serveur se compose de *Computational Resources Daemons* (CRD) et d'un *Server Daemon* (SeD). Nous avons ensuite une hiérarchie (organisée de manière arborescente) d'agents dont les *Leader Agents* (LA) et les *Master Agents* (MA). Un *ReDirector* (ReD) est utilisé pour choisir un MA proche du client. La figure 4.8 montre une vue globale de notre architecture.

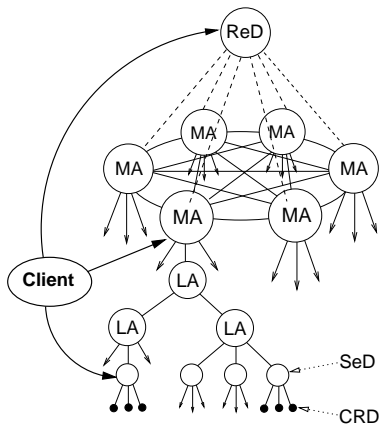


FIG. 4.8 – Vue générale de DIET.

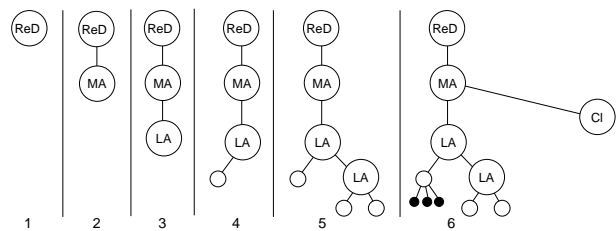


FIG. 4.9 – Initialisation d'un système DIET.

**Computational Resources Daemons (CRD)** Une ressource de calculs est un ensemble de composants matériels et logiciels qui peuvent effectuer des calculs séquentiels ou parallèles sur des données envoyées par un client (ou par un autre serveur). Sur une machine interactive, chaque nœud doit avoir un CRD. Dans les autres cas (comme pour les systèmes à traitement par lots), les calculs sont lancés directement par le Server Daemon.

**Server Daemon (SeD)** Un SeD est le point d'entrée d'un serveur de calculs. Il gère l'ensemble des CRDs et se trouve sous la responsabilité d'un LA. Il tient à jour une liste des données disponibles sur un serveur (éventuellement avec leur distribution et le moyen d'y accéder), une liste des problèmes qui peuvent y être résolus et toutes les informations concernant sa charge (mémoire disponible, nombre de CRDs disponibles, etc.).

**Leader Agent (LA)** Un LA compose un niveau hiérarchique dans les agents DIET. Il peut être le lien entre un Master Agent et un SeD, entre un autre LA et un SeD ou entre deux LA. Son but est de diffuser les requêtes et les informations entre les MAs et les SeD. Il tient à jour une liste des requêtes en cours de traitement et, pour chacun de ses sous-arbres, le nombre de serveurs pouvant résoudre un problème donné, ainsi que des informations à propos des données.

**Master Agent (MA)** Un MA est directement relié aux clients. Il reçoit des requêtes de calculs des clients et choisit un (ou plusieurs) SeDs qui sont capables de résoudre le problème en un temps raisonnable. Un MA possède les mêmes informations qu'un LA, mais il a une vue globale (et de haut niveau) de tous les problèmes qui peuvent être résolus et de toutes les données qui sont distribuées dans tous ses sous-arbres.

**ReDirector (ReD)** Le ReD est le point d'entrée principal de DIET. Il est unique et est utilisé par un client à sa première connexion pour connaître l'adresse du MA le plus approprié.

#### 4.3.4 Scénarios d'utilisation de la plate-forme

Nous proposons un ensemble d'algorithmes pour gérer une telle plate-forme logicielle (initialisation, ajout de serveurs, demande de résolution de problèmes, etc.) [93]. L'article est donné dans le chapitre I du document annexe. J'en donne ici les grandes idées.

**Initialisation de DIET** La figure 4.9 montre chaque étape de l'initialisation d'un système de metacomputing simple. Le ReDirector est la première entité à être démarrée. C'est en effet le point d'entrée du système. Un Master Agent vient s'enregistrer auprès du ReD. Un Leader Agent est lancé et vient se connecter au MA dont il dépend. Un Server Daemon, qui peut gérer des ressources de calculs allant du processeur au serveur de calculs à plusieurs nœuds (machines parallèles, grappes de stations, etc), peut alors se connecter à un LA. Un autre LA peut également venir se connecter au précédent, ajoutant ainsi un niveau de hiérarchie. Le système est alors fonctionnel et un client peut se connecter directement à un MA, dont la référence lui a précédemment été fournie par le ReD.

Le nœud père étant un paramètre de lancement de chaque LA et chaque LA transmettant les changements locaux de configuration à son père, l'administration peut elle aussi être hiérarchique. Par exemple, un LA peut gérer les serveurs d'une organisation comme un laboratoire, chaque équipe lançant et administrant un LA fils pour gérer ses propres serveurs. Chaque MA gère un domaine (comme une université), donnant aux calculs de ses clients un accès privilégié aux serveurs situés dans son domaine. Il est de la responsabilité du ReD de collecter les informations sur les MAs et de déterminer lequel d'entre eux est le mieux adapté pour la connexion d'un nouveau client.

**Résolution d'un problème** Un nouveau client de DIET doit d'abord contacter le ReDirector pour connaître le Master Agent le plus approprié (suivant sa proximité dans le réseau). Une fois que son Master Agent est identifié, le client peut lui soumettre un problème. Pour choisir le serveur le plus approprié pour résoudre ce problème, le Master Agent propage une requête dans ses sous-arbres afin de trouver à la fois les données impliquées (parfois issues de calculs précédents et ainsi déjà présentes sur certains serveurs) et les serveurs capables de résoudre l'opération demandée. Ensuite, le Master Agent renvoie l'adresse du serveur choisi au client et effectue le transfert des données persistantes impliquées dans le calcul. Le client communique

ses données locales au serveur et alors la résolution du calcul peut être effectuée. Les résultats pourront être renvoyés au client en fonction du problème. En général, nous essayons autant que possible de laisser les données sur place.

**Déplacement des données** Lorsqu'un serveur doit accéder à des données se trouvant sur le client pour résoudre un problème, celles-ci doivent être déplacées. Nous supposons que les données peuvent être modifiées durant le processus de résolution. Une copie d'une donnée est donc invalidée dès qu'elle est envoyée vers un autre site. Ainsi, une donnée n'est possédée que par un serveur à un instant donné. Pour des raisons de sécurité, toute communication de données d'un client à un serveur est initiée par le client, suite à un ordre du MA.

Les données sont communiquées par le plus court chemin possible. En l'absence de pare-feu, la communication est directe. Dans le cas contraire, la donnée doit transiter par un ou plusieurs proxies. Dans tous les cas, il est préférable que l'architecture DIET reflète l'architecture physique du réseau. Les senseurs placés sur les nœuds de l'arbre DIET doivent correspondre aux routeurs du réseau pour que les prévisions de performances de FAST soient exactes. Le non respect de cette règle entraîne une surestimation des temps de communication.

Lorsqu'une donnée doit être déplacée d'un serveur vers un autre, la communication est initiée par le serveur ayant besoin de la donnée (sur un ordre du Master Agent qui a localisé la donnée lors de la réponse des serveurs). Là aussi, la communication est la plus directe possible. L'information de localisation doit cependant être mise à jour dans l'architecture.

**Tolérance aux pannes** La tolérance aux pannes est un point crucial pour les environnements de calcul distribué. Dans un système hiérarchique comme DIET, deux types de pannes peuvent avoir lieu :

- mort inattendue d'un agent. Les connaissances de cet agent sont dérivées de celles de ses fils. Elles peuvent donc être reconstituées sans problème. L'agent peut par conséquent être relancé sans que le système soit perturbé. En cas d'impossibilité de relancer l'agent, ses fils peuvent également être rattachés à l'un de ses frères ou à son père ;
- panne d'un serveur. Les serveurs constituant les feuilles de l'arbre, des données et des résultats risquent d'être perdus. Nous envisageons dans ce cas l'utilisation d'un système de points de contrôle. Un point de contrôle correspond à la sauvegarde du contexte (pointeur d'instruction courante et données intermédiaires) d'un calcul sur un serveur dédié à la sauvegarde. En cas de panne d'un serveur, son calcul pourra être repris au dernier point de contrôle par un autre serveur capable qui téléchargera les informations du serveur de sauvegarde.

### 4.3.5 Conclusions

L'architecture expliquée dans cette section représente notre première vue d'une approche hiérarchique pour les applications de type NES. Nous pensons que cette hiérarchisation des agents allée à la prise en compte de la localité des calculs et des données devrait donner de bonnes performances sur une plate-forme telle que VTHD. Un prototype est en cours de réalisation.

## 4.4 Algorithmique parallèle mixte

L'un des problèmes algorithmiques auxquels nous nous sommes intéressés autour de SCILAB// concerne l'utilisation conjointe du parallélisme de tâches et du parallélisme de données. En effet, lorsqu'on lance une série de calculs depuis SCILAB sur des serveurs utilisant des bibliothèques numériques parallèles, on utilise à la fois du parallélisme de tâches (chaque calcul est une tâche atomique lancée sur un serveur) et du parallélisme de données (ces tâches sont exécutées en mode data-parallèle à l'intérieur d'un serveur). L'exploitation du parallélisme mixte a plusieurs

avantages. L'un d'entre eux est la capacité d'augmenter l'extensibilité en permettant l'utilisation de plus de parallélisme lorsque le maximum de parallélisme de tâches ou données qui peut être exploité est atteint.

Un bon état de l'art de ce sujet est donné dans [25]. La plupart des recherches à propos de l'exploitation simultanée du parallélisme de données et du parallélisme de tâches ont été effectuées dans le domaine des langages de programmation pour fournir des accès simples et de haut niveau à plus de parallélisme, et dans le domaine de la compilation, où des problèmes tels que l'ordonnancement et le placement de tâches data-parallèles concurrentes sont étudiés [232]. Dans [228], Ramaswamy introduit la structure de Macro Dataflow Graph (MDG) pour décrire les programmes en parallélisme mixte. Un MDG est un graphe acyclique direct où les nœuds représentent des calculs séquentiels ou data-parallèles et les arcs représentent les relations de précédence. Deux nœuds sont distingués, l'un précédant et l'autre suivant tous les autres nœuds. Une fois le MDG extrait du code, un algorithme est appliqué pour ordonnancer et placer simultanément les tâches sur les ressources de calculs.

Avec F. Suter, nous avons commencé par étudier les algorithmes de Strassen et Winograd de produit de matrices qui se prêtent bien à ce type de parallélisme [102]. Ils ont été très étudiés sur des machines mono-processeur pour augmenter les performances de calculs des applications numériques [23, 66, 146, 154, 254]. Plusieurs implémentations de ces deux algorithmes ont déjà été proposées, mais la plupart d'entre elles utilisent soit le parallélisme de données [136] soit le parallélisme de tâches [73]. Des versions utilisant le parallélisme mixte [117, 228] existent également. La première [117] utilise une distribution unidimensionnelle des matrices et la seconde [228] est l'application d'un outil de parallélisation automatique.

Dans notre approche, l'algorithme de Strassen est utilisé au premier niveau de récursion et la fonction de produit matriciel du niveau 3 de la bibliothèque PBLAS est utilisée au niveau inférieur. Nous avons supposé que les matrices étaient distribuées sur des grilles disjointes de processeurs en raison de calculs précédents.  $A$  et  $B$  sont deux matrices carrées de taille  $M$  distribuées sur deux grilles carrées de processeurs. Nous voulons calculer le produit  $C = AB$ , avec  $C$  distribuée sur la même grille que  $A$ . Dans un tel cas, il y a deux manières « classiques » de calculer ce produit. Tout d'abord, on peut redistribuer  $B$  sur la même grille que  $A$  puis calculer le produit sur cette grille. Mais cette méthode n'utilise que la moitié des processeurs disponibles. Ensuite, on redistribue  $A$  et  $B$  sur la grille globale, effectuer le produit, puis redistribuer  $C$  sur la sous-grille où est située  $A$ . Nous avons proposé une troisième voie mixte, qui conserve les matrices en place et qui distribue équitablement les tâches (e.g., additions et multiplications de quarts de matrices) sur les deux sous-grilles. Nos algorithmes optimisent l'utilisation des temporaires et les redistributions entre les grilles.

Nous étudions maintenant la faisabilité d'une application de la récursion sur ce type d'algorithmes en parallélisme mixte. Le problème principal est une augmentation importante du nombre de communications lors des redistributions de matrices entre les sous-grilles. Nous étudions actuellement l'application des algorithmes que nous avons développés à des plates-formes hétérogènes. Nous pensons qu'ils sont plus adaptés à ce type de plates-formes que les algorithmes réguliers habituels. Par contre, ils nécessitent alors d'avoir des routines de redistribution adaptées et cela représente un problème difficile.

Les algorithmes, leur modélisation et les évaluations de performances sont donnés dans le chapitre G du document annexe.

## 4.5 Conclusion et perspectives

Les travaux autour de SCILAB// ont montré qu'il était possible d'obtenir de bonnes performances avec un outil de type MATLAB et ceci avec diverses approches. Nos travaux se poursuivent et de nombreuses voies restent encore à explorer.

Pour ce qui est de SCILAB lui-même, nous sommes encore loin du niveau applicatif où l'utilisateur programme en séquentiel sans aucun ajout de code. Même si nous avons ajouté la

surcharge d'opérateurs, nous avons pour l'instant conservé des commandes de (re)distribution de données. Nous disposons cependant des algorithmes pour pouvoir distribuer automatiquement les matrices (grâce par exemple à nos travaux sur ALASCA) mais plusieurs problèmes se posent. Le premier concerne l'inférence des types SCILAB et le fait que les tailles de matrices (et leur dimensions) peuvent varier à tout moment. Deuxièmement, si la distribution doit être faite au niveau de l'outil SCILAB lui-même, il faut être capable de lui ajouter de l'« intelligence ». Pour l'instant, seule l'approche de type NES permet de mettre une analyse de l'environnement et de l'algorithme dans l'agent, le client SCILAB se contentant de lui passer les demandes de résolutions de problèmes. L'ajout de fonctionnalités complexes à SCILAB pose de nombreux problèmes (essentiellement logiciels) et rend nos développements trop liés à cet environnement. Nous préférons donc dans un futur proche nous concentrer sur l'approche NES tout en poursuivant nos travaux autour de SCILAB// et son interfaçage avec DIET.

Une autre approche de parallélisation de SCILAB reste encore à explorer et a été demandée à plusieurs reprises par des utilisateurs. Il s'agit d'une version multi-threads de SCILAB. Une bonne solution serait certainement de coupler SCILAB à PM<sup>2</sup> [206] en pouvant générer des threads pour certaines tâches de SCILAB et en leur laissant la possibilité de migrer sur d'autres processeurs suivant la charge de la machine cible. En plus des problèmes classiques d'ordonnancement, cela pose également certains problèmes de réalisation (gestion multi-threads de la pile de SCILAB, utilisation de fonctions SCILAB ou de bibliothèques numériques, etc.).

De plus, on peut également imaginer un outil d'aide à l'écriture de programmes de type SCILAB en vue de leur parallélisation vers un environnement tel que celui développé dans le cadre de l'ARC OURAGAN. La version Fortran 77/HPF de TRANSTOOL s'est terminée en même temps que le projet EuroTOPS. En revanche, nous souhaitons l'utiliser pour générer du code SCILAB//. En effet, quel que soit le langage d'entrée, on retrouve les mêmes problèmes lorsqu'on souhaite transformer des appels à des noyaux d'algèbre linéaire vers des bibliothèques parallèle. SCILAB (ou MATLAB) ajoute de nouveaux problèmes tels que l'inférence des types et les tailles dynamiques de matrices.

En ce qui concerne DIET, cet environnement en est à ses balbutiements et pratiquement tout reste à faire. Nous suivons de près les travaux du Grid Forum et en particulier du groupe de travail autour des environnements de programmation de la grille. Je donne des perspectives sur ce domaine dans la conclusion générale du document.



# Chapitre 5

## Conclusions et perspectives



Les perspectives actuelles du calcul à hautes performances vont donc vers l'utilisation de plateformes de metacomputing. Ceci ne va pas sans poser de nombreux problèmes de recherche vers lesquels je vais m'orienter. Dans ce chapitre, je présente donc mes perspectives de recherche pour les cinq prochaines années.

## 5.1 Algorithmique hétérogène

Peu de travaux existent encore en ce qui concerne les bibliothèques numériques adaptées à la grille. Cela vient principalement du fait que les applications actuelles de la grille sont principalement des applications à gros grain et peu couplées (du fait des faibles performances de cette même grille). Cependant, il existe quelques résultats sur le placement de données dans un environnement hétérogène et notamment les travaux de Robert et al. [31, 44]. Le développement d'environnements de metacomputing efficaces pour les applications de simulation numérique ne peut se faire sans une expertise en algorithmique parallèle. L'utilisation de plates-formes hétérogènes avec des latences de communication importantes complique la recherche d'algorithmes efficaces et génériques. Les problèmes théoriques sont difficiles et seules des heuristiques permettront d'obtenir des solutions efficaces. Cependant, il reste à savoir si ces solutions trouvées seront réellement implémentables dans des bibliothèques.

La modélisation d'une telle plate-forme est un problème en soi. Les modélisations actuelles sont hiérarchiques et relativement statiques. L'utilisation d'outils tels que NWS couplée à une modélisation assez fine de l'architecture et de ses protocoles de routage permettra d'obtenir des « modèles » relativement fiables. Ces recherches doivent être effectuées avec les spécialistes du système et des réseaux.

L'ordonnancement de tâches dans un tel contexte ne peut utiliser les algorithmes développés auparavant dans un contexte homogène. Des approches où l'on couple un ordonnancement statique (basé sur une « photographie » de l'architecture à un instant donné) à un raffinement dynamique pourraient donner les meilleurs résultats. Par contre, l'utilisation d'un support d'exécution à hautes performances est obligatoire. D'autres stratégies basées sur la réplification des calculs pourraient s'avérer être nécessaires pour exploiter efficacement les grappes et réseaux hétérogènes.

Enfin, le modèle de programmation lui-même doit être revu. Actuellement, la plupart des projets utilisent toujours une programmation au niveau expert où l'utilisateur doit connaître parfaitement son application et sa « machine » cible pour obtenir une application parallèle. C'est l'héritage du portage d'applications Fortran qui perdure. L'approche et les méthodologies dites à objets connaissent depuis plusieurs années un grand succès dans les développements logiciels. Concernant les aspects méthodologiques, l'accent est mis sur des notions telles que la réutilisation (héritage), la modularité (séparation des spécifications et implémentations) et l'encapsulation forte du code et des données (objets, méthodes). Des tentatives d'utilisation de ces nouvelles approches ont été faites dans le domaine du parallélisme, mais elles s'étaient longtemps heurtées à l'absence de standards et à une crainte des utilisateurs sur le niveau de performance qu'on pouvait en attendre. Aujourd'hui, les mentalités ont changé et les standards, comme JAVA et CORBA, existent et suscitent un vif intérêt. Des implémentations performantes de JAVA et CORBA commencent à apparaître et devraient atténuer les craintes des utilisateurs de codes scientifiques. Dans le contexte du metacomputing et des environnements fortement hétérogènes, les technologies à objets et la programmation par composants logiciels apportent une solution élégante aux problèmes posés par l'hétérogénéité en encapsulant les spécificités techniques (protocoles, représentation des données, migration) dans l'implémentation d'objets. Ces approches simplifient également l'interopérabilité entre les logiciels et donc la réutilisation de codes déjà développés.

Mes travaux sur les bibliothèques de calcul et la redistribution de matrices peuvent s'appliquer aux outils pour les plates-formes de metacomputing. Et à court terme, je souhaite faire évoluer mes recherches en algorithmique numérique parallèle vers une utilisation du parallélisme mixte

(utilisation simultanée du parallélisme de tâches et du parallélisme de données) en milieu hétérogène. Quelques résultats existent déjà dans ce domaine mais il reste encore beaucoup de travaux à effectuer, surtout si l'on veut obtenir une généricité suffisante qui permettra le développement de bibliothèques pour ce type d'architectures. Je souhaite également étudier l'impact d'une telle approche sur l'algorithmique numérique creuse parallèle. Ce type de solveurs représente le noyau de la plupart des applications de simulation et les bibliothèques actuelles sont loin d'être complètes et aucune, à ma connaissance, ne permet d'utiliser une architecture hétérogène. Le projet NSF-INRIA que nous venons d'obtenir avec le LaBRI, l'ENSEEIH, le projet Aladin de l'IRISA et l'Université du Minnesota autour d'algorithmes utilisant à la fois les méthodes itératives et les méthodes directes va m'ouvrir de nouvelles perspectives de recherche et de nouvelles collaborations avec les spécialistes du domaine.

## 5.2 Serveurs de calculs

Comme nous l'avons vu dans le chapitre 4, nous sommes en train de mettre en place des serveurs de calcul dans un environnement de metacomputing. Le but de ces serveurs est de fournir à travers le réseau la puissance de calcul et de grandes capacités mémoire. À partir d'un client (un programme dans un langage quelconque, un logiciel tel que SCILAB ou MATLAB ou même un navigateur W3), on lancera des calculs sur le réseau sans savoir où ils seront exécutés. Les utilisateurs d'un tel environnement sont par exemple un programmeur d'application qui ne souhaite pas installer de nombreuses bibliothèques ou qui ne possède pas une machine de puissance suffisante ou une société qui souhaite laisser utiliser un code (et des données) sans pour autant le laisser sortir de ses murs.

Les problèmes théoriques et logiciels sont nombreux et importants. Dans un premier temps, si l'on veut conserver la plus grande des transparences d'utilisation tout en ayant les meilleures performances, il faut que l'environnement soit capable de distribuer les données et les calculs automatiquement (problèmes d'ordonnancement *on-line* et *off-line* sur architecture hétérogène) et d'évaluer les performances dynamiquement de manière suffisamment précise. L'hétérogénéité de l'architecture matérielle et logicielle complique encore la réalisation d'un tel environnement. D'autres aspects liés à l'utilisation d'un réseau de type WAN ne sont pas à négliger comme la tolérance aux pannes, la sécurité, la confidentialité et l'administration de l'environnement complet. Comme nous l'avons vu précédemment, des environnements de ce type existent déjà comme NINF ou NETSOLVE mais ils sont pour l'instant encore trop limités (pas de persistance de données sur les serveurs, équilibrage des charges, évaluation des performances et extensibilité limitées, pratiquement pas d'utilisation de middleware pour la grille). Notre but est de développer des couches logicielles qui permettront de mettre facilement en place des environnements de type serveurs de calcul en partant de logiciels existants et si possible standard. Le choix du middleware est clairement l'un des points importants pour le développement de telles architectures logicielles [127].

CORBA est une référence incontournable pour le développement d'applications réparties. Autrefois éloigné des applications numériques du fait de ses faibles performances et de son manque de certaines fonctionnalités, il est maintenant un des middlewares majeurs pour le développement de telles applications [30, 234, 272] et le fait qu'il soit un standard l'aidera à entrer dans le monde industriel. De plus, de nombreux projets ont pour but de fournir un ORB à hautes performances basé sur des couches de communications plus adaptées aux grappes de machines parallèles.

JAVA semble être un des outils logiciels pour la programmation d'applications réparties et un nombre colossal de projets tentent de l'imposer<sup>1</sup> [141, 209]. Une voie intéressante semble être la possibilité de charger à distance des proxies sur les clients pour gérer le protocole de communication entre eux, les agents et les serveurs. Jini [17, 161] a été créé pour gérer ce type d'approche

---

<sup>1</sup>Voir le succès des conférences JAVA Grande de l'ACM.

et le fait que l'on puisse envoyer un code exécutable à distance sur un client rend plus simple le développement de telles applications.

### 5.3 Outils de haut niveau

Dans [39], R. Boisvert prédit que dans le futur, le pourcentage de personnes qui utiliseront directement des bibliothèques mathématiques se réduira ! Au contraire, les environnements de résolution de problèmes (PSE) se développeront [52, 150] et s'amélioreront et ainsi les chercheurs et ingénieurs pourront se concentrer sur leurs problèmes applicatifs plutôt que de chercher sur le réseau une (ou plusieurs) bibliothèques qui permettront de les implémenter. Il y a gros à parier que des environnements comme Matlab, Mathematica, Octave, Scilab, Maple ou Mupad auront leur rôle à jouer s'ils permettent d'accéder de manière transparente à de grosses puissances de calculs et de grandes capacités de stockage. Leur capacité à résoudre des problèmes d'un niveau de plus en plus haut devra augmenter et l'on aura certainement des couplages d'applications dans ce type d'environnement.

Par contre, cette prédiction ne se réalisera que si les environnements actuels évoluent vers plus de transparence. Cette transparence n'est pas uniquement liée aux logiciels utilisés ou aux architectures cibles choisies mais aussi en ce qui concerne les méthodes mathématiques elles-mêmes. Certains travaux existent déjà autour de l'automatisation du choix de solveurs en fonction des caractéristiques des données en entrée [16] et qui utilisent parfois des bases de données et des systèmes experts comme PYTHIA-II [60].

Ces outils, dotés de capacités de résolution énormes (à la fois en nombre de problèmes et en puissance) pourront alors servir à tous les niveaux, des chercheurs spécialistes d'un domaine précis aux étudiants.

### 5.4 Conclusion personnelle

Les travaux présentés dans ce document représentent environ dix ans de ma vie de chercheur. Ils ont évolué au gré de mes rencontres avec des chercheurs et des industriels de différents domaines, des évolutions de l'informatique et des réseaux, de mes goûts et de mes collaborations avec les étudiants avec qui j'ai eu la chance de travailler.

Comme nous avons pu le voir durant ces 3 chapitres, mes recherches ont traversé toutes les couches logicielles depuis les communications à bas niveau dans les réseaux rapides jusqu'aux outils de programmation et de parallélisation de haut niveau. Chacun de ces travaux, même s'il n'a pas toujours débouché sur des publications ou des logiciels, m'a permis d'acquérir cette expérience dont je tente maintenant de faire profiter ceux avec qui je travaille.

Comme le dit souvent Yves Robert, je veux tout faire en même temps et je pense effectivement que 24 heures par jour ne sont souvent pas suffisants pour tout étudier. Quoiqu'il en soit, même si je ne sais pas si ces travaux auront fait « avancer la science »<sup>2</sup>, ils m'auront au moins motivés pour une partie non négligeable de ma carrière !

---

<sup>2</sup>Hein, Jean-Michel?



# Bibliographie

- [1] T. S. Abdelrahman. Latency hiding on COMA multiprocessors. *The Journal of Supercomputing*, 10(3) :225–242, 1996. <http://www.eecg.toronto.edu/www-tsa/jsuper96.ps>.
- [2] V.S. Adve, J.C. Wang, J. Mellor-Crummey, D.A. Reed, M. Anderson, and K. Kennedy. An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs. Technical Report CRPC-TR94513-S, Center for Research on Parallel Computation, Rice University, December 1994.
- [3] R. Agarwal, F. Gustavson, , and M. Zubair. A High Performance Matrix Multiplication Algorithm on a Distributed-Memory Parallel Computer. *IBM Journal of Research and Development*, 38(6) :673–681, Nov. 1994.
- [4] J.R. Allen and K. Kennedy. Automatic Translations of Fortran Programs to Vector Form. *ACM Toplas*, 9 :491–542, 1987.
- [5] P. Amarasinghe, J.M. Anderson, M.S. Lam, and C.-W. Tseng. The SUIF Compiler for Scalable Parallel Machines. In *Seventh SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.
- [6] P.R. Amestoy, I.S. Duff, J.Y. L'Excellent, and J. Koster. A Fully Asynchronous Multi-frontal Solver using Distributed Dynamic Scheduling. Technical Report RT/APO/99/2, ENSEEIHT-IRIT, 1999. accepted to SIAM Journal on Matrix Analysis and Applications, [ftp://ftp.enseeiht.fr/pub/numerique/AMESTOY/N7\\_RT\\_99\\_2.ps.Z](ftp://ftp.enseeiht.fr/pub/numerique/AMESTOY/N7_RT_99_2.ps.Z).
- [7] M. B. Amin, A. Grama, and V. Singh. Fast Volume Rendering Using an Efficient Parallel Formulation of the Shear-Warp Algorithm. In *Proceedings 1995 Parallel Rendering Symp.*, 1995. [ftp://ftp.cs.umn.edu/dept/users/kumar/parallel\\_rendering.ps](ftp://ftp.cs.umn.edu/dept/users/kumar/parallel_rendering.ps).
- [8] C. Ancourt, F. Coelho, F. Irigoien, and R. Keryell. A Linear Algebra Framework for Static HPF Code Distribution. *Scientific programming*, 1996. Available as CRI-Ecole des Mines Technical Report A-278-CRI, and at <http://www.cri.enscm.fr>.
- [9] E. Anderson and J.J. Dongarra. LAPACK Working Note : Evaluating Block Algorithm Variants in LAPACK. Technical Report 19, The University of Tennessee - Knoxville, 1990.
- [10] J. Anderson. *Automatic Computation and Data Decomposition for Multiprocessors* . PhD thesis, Stanford University, March 1997.
- [11] R. Andonov, H. Bourzoufi, and S. Rajopadhye. Two-Dimensional Orthogonal Tiling : From Theory to Practice. In *International Conference on High Performance Computing (HiPC)*, pages 225–231, Trivandrum, India, 1996. <http://www.univ-valenciennes.fr/limav/andonov/pub/rech/HiPC-camera.ps.g%z>.
- [12] R. Andonov and N. Yanev. n-Dimensional Orthogonal Tiling. Technical Report LIMAV-RR 96-6, Université de Valenciennes, LIMAV, September 1996. <http://www.univ-valenciennes.fr/limav/andonov/pub/rech/orti.ps.gz>.
- [13] R. Andonov, N. Yanev, and H. Bourzoufi. Three-Dimensional Orthogonal Tile Sizing Problem : Mathematical Programming Approach. Technical Report LIMAV-RR 96-3, Université de Valenciennes, LIMAV, May 1996. <http://www.univ-valenciennes.fr/limav/andonov/pub/rech/asap.ps.gz>.
- [14] P. Arbenz, W. Gander, and J. Moré. The Remote Computational System. *Parallel Computing*, 23(10) :1421–1428, 1997.

- [15] D. C. Arnold, D. Bachmann, and J. J. Dongarra. Request Sequencing : Optimizing Communication for the Grid. In A. Bode, T. Ludwig, W. Karl, and R. Wismuller, editors, *Euro-Par 2000 Parallel Processing, 6th International Euro-Par Conference*, volume 1900 of *Lecture Notes in Computer Science*, pages 1213–1222, Munich Germany, August 2000. Springer Verlag. <http://www.netlib.org/utk/people/JackDongarra/PAPERS/sequencing.pdf>.
- [16] D. C. Arnold, S. Blackford, J. J. Dongarra, and V. Eijkhout. Seamless Access to Adaptive Solver Algorithms. In *16th IMACS World Congress*, 2000.
- [17] G. Aschemann, R. Kehr, and A. Zeidler. A Jini-based Gateway Architecture for Mobile Devices. In *JIT'99*, Duesseldorf, Germany, September 1999.
- [18] ASCI. <http://www.asci.doe.gov/>.
- [19] O. Aumage, L. Bougé, A. Denis, J.-F. Méhaut, G. Mercier, and L. Prylli. A Portable and Efficient Communication Library for High-Performance Cluster Computing. In *Proceedings of the IEEE Int. Conf. on Cluster Computing (CLUSTER 2000)*, 2000.
- [20] D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4) :345–420, December 1994.
- [21] S. B. Baden and S. J. Fink. Communication Overlap in Multi-tier Parallel Algorithms. In *Proc. of SC '98*, Orlando, Florida, Nov. 1998. [ftp://ftp.cs.ucsd.edu/pub/scg/papers/1998/k2\\_sc98.ps.gz](ftp://ftp.cs.ucsd.edu/pub/scg/papers/1998/k2_sc98.ps.gz).
- [22] R. Bagrodia, E. Deelman, S. Doco, and T. Phan. Performance Prediction of Large Parallel Applications using Parallel Simulations. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1999. <ftp://pcl.cs.ucla.edu/pub/papers/ppopp99.ps>.
- [23] D. Bailey, K. Lee, and H. Simon. Using Strassen's Algorithm to Accelerate the Solution of Linear Systems. *Journal of Supercomputing*, 4(4) :357–371, Jan 1991.
- [24] M. Baker. Cluster Computing White Paper, 2000.
- [25] H. Bal and M. Haines. Approaches for Integrating Task and Data Parallelism. *IEEE Concurrency*, Jul-Sep 1998.
- [26] V. Bala, S. Kipnis, L. Rudolph, and M. Snir. Designing Efficient, Scalable, and Portable Collective Communication Libraries. In R.F. Sincovec, D.E. Keyes, M.R. Leuze, L.R. Petzold, and D.A. Reed, editors, *Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 862–872. SIAM, 1993.
- [27] S. Balay, W. D. Gropp, L. Curfman McInnes, and B. F. Smith. PETSc Home Page. <http://www.mcs.anl.gov/petsc>, 1998.
- [28] P. Banerjee, J.A. Chandy, M. Gupta, E.W. Hodges-IV, J.G. Holm, A. Lain, D.J. Palermo, S. Ramaswamy, and E. Su. The PARADIGM Compiler for Distributed-Memory Multi-computers. *IEEE Transactions on Computers*, 28(10) :37–47, October 1995.
- [29] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, and D. Zaretsky. A MATLAB Compiler For Distributed, Heterogeneous, Reconfigurable Computing Systems. In *Proceedings of the Int. Symp. on FPGA Custom Computing Machines (FCCM-2000)*, April 2000.
- [30] P. Beaugendre, T. Priol, G. Alléon, and D. Delavaux. A Client/Server Approach for HPC Applications within a Networking Environment. In *Proceedings of HPCN'98*, pages 518–525, Amsterdam, The Netherlands, April 1998.
- [31] O. Beaumont, V. Boudet, A. Legrand, F. Rastello, and Y. Robert. Heterogeneity Considered Harmful to Algorithm Designers. In *Cluster'2000*, pages 403–404. IEEE Computer Society Press, 2000.
- [32] R. Bianchini, L. Kontothanassis, R. Pinto, M. De Maria, M. Abud, and C. L. Amorim. Hiding Communication Latency and Coherence Overhead in Software DSMs. In *Proceedings of the 7th ACM/IEEE International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 7)*, October 1996. <http://www.cos.ufrj.br/~ricardo/>.



- [33] A.J.C. Bik, P.J.H. Brinkhaus, P.M.W. Knijnenburg, and H.A.G. Wijshoff. The Automatic Generation of Sparse Primitive. *ACM Transactions on Mathematical Software*, 24(2) :190–225, June 1998.
- [34] J. Bilmès, K. Asanović, C.W. Chin, and J. Demmel. Optimizing Matrix Multiply Using PHiPAC : A Portable, High-Performance, ANSI C Coding Methodology. In *Proceedings of the International Conference on Supercomputing*, pages 340–347, Vienna, Austria, 1997. ACM SIGARCH.
- [35] BIP. <http://lhpc.univ-lyon1.fr/bip.html>.
- [36] R. Bixby, K. Kennedy, and U. Kremer. Automatic Data Layout using 0-1 Integer Programming. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT94)*, pages 111–122, Montreal, Canada, August 1994.
- [37] L.S. Blackford, J. Choi, A. Cleary, E. D’Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. *ScaLAPACK Users’ Guide*. SIAM, 1997.
- [38] B. Blount and S. Chatterjee. An Evaluation of Java for Numerical Computing. In *ISCOPE*, pages 35–46, 1998.
- [39] R. Boisvert. Mathematical Software : Past, Present, and Future. In *International Symposium on Computational Sciences*, Purdue University, May 1999.
- [40] R. Boisvert, S. Howe, and D. Kahaner. GAMS - A Framework for the Management of Scientific Software. *ACM Transaction on Mathematical Software*, 11 :313–355, 1985. <http://gams.nist.gov/>.
- [41] R. F. Boisvert, J. J. Dongarra, R. Pozo, K. A. Remington, and G. W. Stewart. Developing Numerical Libraries in Java. *Concurrency : Practice and Experience*, 10(11–13) :1117–1129, 1998.
- [42] C. Bonello, F. Desprez, and B. Tourancheau. Parallel BLAS and BLACS for Numerical Algorithms on a Reconfigurable Network. In S. Atkins and A.S. Wagner, editors, *NATUG6 - Transputer Research and Applications 6*, pages 21–38. IOS Press, 1993.
- [43] G. Bosilca, G. Fedak, and F. Cappello. OVM : Out-of-Order Execution Parallel Virtual Machine. In *Proceedings of CCGRID’2001*. IEEE/ACM, IEEE press, May 2001.
- [44] V. Boudet, A. Petitet, F. Rastello, and Y. Robert. Data Allocation Strategies for Dense Linear Algebra Kernels on Heterogeneous Two-Dimensional Grid. In *International Conference on Parallel and Distributed Computing and Systems (PDCS’99)*, pages 561–569. IASTED Press, 1999.
- [45] T. Brandes, S. Chaumette, M.-C. Counilh, A. Darte, J.C. Mignot, F. Desprez, and J. Roman. HPFIT : A Set of Integrated Tools for the Parallelization of Applications Using High Performance Fortran : Part I : HPFIT and the TransTOOL Environment. *Parallel Computing*, 23(1-2) :71–87, 1997.
- [46] T. Brandes, S. Chaumette, M.-C. Counilh, J.C. Mignot, F. Desprez, and J. Roman. HPFIT : A Set of Integrated Tools for the Parallelization of Applications Using High Performance Fortran : Part II : Data Structures Visualization and HPF Support for Irregular Data Structures with Hierarchical Scheme. *Parallel Computing*, 23(1-2) :89–105, 1997.
- [47] T. Brandes and F. Desprez. Implementing Pipelined Computation and Communication in an HPF Compiler. In *Europar’96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Science*, pages 459–462. Springer Verlag, August 1996.
- [48] T. Brandes and F. Desprez. Pipelining Data Parallel Computations in an HPF Compiler. In M. Gerndt, editor, *Sixth Workshop on Compilers for Parallel Computers*, volume 21, pages 73–89, Aachen, Germany, December 1996. Forschungszentrum Jülich.
- [49] J. Briat and A. Carissimi. Intégration de *threads* et communications : une étude de cas. In *11<sup>ème</sup> Rencontres francophones du parallélisme, des architectures et des systèmes*, Rennes, France, June 1999.

- [50] S. Brown, J. Dongarra, E. Grosse, and T. Rowan. The Netlib Software Repository. *D-LIB Magazine*, September 1995. <http://www.netlib.org/>.
- [51] S. Browne, P. McMahan, and S. Wells. Repository in a box toolkit for software and resource sharing. Technical Report UT-CS-99-424, University of Tennessee Computer Science Dept, 1999.
- [52] Rajkummar Buyya, Tom Eidson, Dennis Gannon, Erwin Laure, Satoshi Matsuoka, Thierry Priol, Joel Saltz, Ed Seidel, and Yoshio Tanaka. Problem Solving Environment Comparison. [http://www.eece.unm.edu/~apm/WhitePapers/APM\\_Sys\\_Comp.pdf](http://www.eece.unm.edu/~apm/WhitePapers/APM_Sys_Comp.pdf), 2001.
- [53] P.-Y. Calland, J.J. Dongarra, and Y. Robert. Tiling With Limited Resources. Technical Report UT-CS-97-350, CS Dept, The University of Tennessee, February 1997.
- [54] C. Calvin, L. Colombet, F. Desprez, B. Jargot, P. Michallon, B. Tourancheau, and D. Trystram. Towards Mixed Computation - Communication in Scientific Libraries. In *CONPAR'94 - VAPP VI*, volume 854 of *Lecture Notes in Computer Science*, pages 605–615, Lintz, 1994. Springer Verlag.
- [55] C. Calvin and F. Desprez. Minimizing Communication Overhead Using Pipelining for Multi-Dimensional FFT on Distributed Memory Machines. In D.J. Evans, H. Liddell J.R. Joubert, and D. Trystram, editors, *Parallel Computing'93*, pages 65–72. Elsevier Science Publishers B.V. (North-Holland), 1993.
- [56] E. Caron, S. Chaumette, S. Contassot-Vivier, F. Desprez, E. Fleury, C. Gomez, M. Goursat, E. Jeannot, D. Lazure, F. Lombard, J.M. Nicod, L. Philippe, M. Quinson, P. Ramet, J. Roman, F. Rubi, S. Steer, F. Suter, and G. Utard. Scilab to Scilab / / , the OURAGAN Project. *Parallel Computing*, 2001. to appear.
- [57] E. Caron, D. Lazure, and G. Utard. Performance Prediction and Analysis of Parallel Out-of-Core Matrix Factorization. In *Conference on High Performance Computing (HiPC'2000)*, volume 1593 of *Lecture Notes in Computer Science*, pages 17–20, Bangalore, India, December 2000. Springer Verlag.
- [58] H. Casanova and J. Dongarra. A Network-Enabled Server for Solving Computational Science Problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3) :212–223, 1997. and proceedings of Supercomputing'96, Pittsburgh.
- [59] H. Casanova and J. Dongarra. Using Agent-Based Software for Scientific Computing in the Netsolve System. *Parallel Computing*, 24 :1777–1790, 1998.
- [60] A.C. Catlin, E.N. Houstis, N. Ramakrishnan, J.R. Rice, and V.S. Verykios. PYTHIA-II : A Knowledge/Database System for Managing Performance Data and Recommending Scientific Software. *ACM Transaction on Mathematical Software*, 26(2) :227–253, June 2000.
- [61] R. D. Chamberlain and M. A. Franklin. Performance Effects of Synchronization in Parallel Processors. In *Symposium on Parallel and Distributed Processing (SPDP '93)*, pages 611–616, Los Alamitos, Ca., USA, December 1993. IEEE Computer Society Press.
- [62] C.-C. Chang, G. Czajkowski, C. Hawblitzel, and T. von Eicken. Low-Latency Communication on the IBM RISC System/6000 SP. In *ACM/IEEE Supercomputing '96*, Pittsburgh, PA, November 1996. <http://www.cs.cornell.edu/tve/papers-uni/sc96.ps>.
- [63] B.M. Chapman, T. Fahringer, and H.P. Zima. Automatic Support for Data Distribution on Distribution on Distributed Memory Multiprocessor Systems. Technical Report TR 93-2, Institute for Software Technology and Parallel Systems, University of Vienna, Austria, August 1993.
- [64] S. Chatterjee, J. R. Gilbert, F. J. E. Long, R. Schreiber, and S.-H. Teng. Generating Local Addresses and Communication Sets for Data-Parallel Programs. *Journal of Parallel and Distributed Computing*, 26(1) :72–84, 1995.

- [65] S. Chatterjee, J. R. Gilbert, R. S. Schreiber, and S.-H. Tseng. Automatic Array Alignment in Data-Parallel Programs. In ACM Press, editor, *Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–28, Charleston, South Carolina, January 1993.
- [66] S. Chatterjee, A. Lebeck, P. Patnala, and M. Thottethodi. Recursive Array Layouts and Fast Parallel Matrix Multiplication. In *Proceedings of Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*, Saint-Malo, France, June 1999. <ftp://ftp.cs.unc.edu/pub/users/sc/papers/recursive.ps>.
- [67] F. Chaussumier, F. Desprez, and L. Prylli. Asynchronous Communications in MPI – the BIP/Myrinet Approach. In J. Dongarra, E. Luque, and Tomas Margalef., editors, *Proceedings of the EuroPVM/MPI'99 conference*, number 1697 in Lecture Notes in Computer Science, pages 485–492, Barcelona, Spain, September 1999. Springer Verlag.
- [68] F. Chaussumier, F. Desprez, and J. Zory. On the Gain of Multi-Dimensionnal Computation Pipelines. Technical report, INRIA, 2001. En cours de rédaction.
- [69] S. Chauveau and F. Bodin. Menhir : An Environment for High Performance Matlab. In David O'Hallaron, editor, *Languages, Compilers and Run-Time Systems for Scalable Compilers (LCR'98, volume 1511 of Lecture Notes in Computer Science*, pages 27–40, Pittsburgh, PA, May 1998. Springer Verlag.
- [70] C.-L. Chiang, J.-J. Wu, and N.-W. Lin. Toward Supporting Data Parallel Programming on Clusters of Symmetric Multiprocessors. In *Proc. of the 1998 International Conference on Parallel and Distributed Systems*, pages 607–614, Tainan, Taiwan, 1998. IEEE Computer Society Press.
- [71] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK : A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance. *Computer Physics Communications*, 97 :1–15, 1996. (also LAPACK Working Note #95).
- [72] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. LAPACK Working Note : ScaLAPACK : A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performances. Technical Report 95, The University of Tennessee - Knoxville, 1995.
- [73] C.-C. Chou, Y. Deng, and Y. Wang. A Massively Parallel Method for Matrix Multiplication Based on Strassen's Method. Technical Report SUNYSB-AMS-93-17, Center for Scientific Computing, The University of Stony Brook, November 1993.
- [74] M.J. Clement and M.J. Quinn. Overlapping Computations, Communications and I/O in Parallel Sorting. *Journal of Parallel and Distributed Computing*, 28 :162–172, 1995.
- [75] C. Clémançon, A. Endo, J. Fritsher, A. Müller, R. Rühl, and B.J.N. Wylie. The "Annai" Environment for Portable Distributed Parallel Programming. In *28th Hawaii International Conference on System Sciences (HICSS-28)*, volume II, pages 242–251. IEEE Computer Society Press, January 1995.
- [76] F. Coehlo, C. Germain, and J.-L. Pazat. State of the Art in Compiling HPF. Technical Report TR-EMP-CRI A-286, CRI, Ecole des Mines de Paris, 1996. <http://chailly.ensmp.fr/doc/A-286.ps.gz>.
- [77] D.E. Culler, J. Pal Singh, and A. Gupta. *Parallel Computer Architecture : A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1998. ISBN 1-55860-343-3.
- [78] J. Czyzyk, M. Mesnier, and J. Moré. NEOS : The Network-Enabled Optimization System. Technical Report MCS-P615-1096, Mathematical and Computer Science Division, Argonne National Lab., 1996.
- [79] A. Darte. De l'organisation des calculs dans les codes répétitifs. Document d'habilitation à diriger des recherches de l'université de Lyon I, Décembre 1999.

- [80] E. F. Van de Velde. *Concurrent Scientific Computing*. Springer Verlag, 1994.
- [81] J. W. Demmel and N. J. Higham. Stability of Block Algorithms with Fast Level-3 BLAS. *ACM Transactions on Mathematical Software*, 18(3) :274–291, September 1992.
- [82] R.F. Van der Wijngaart, S.R. Sarukkai, and P. Mehra. The Effects of Interrupts on Software Pipeline Execution on Message-passing Architectures. In *Proc. of the 1996 Int. Conf. on Supercomputing (FCRC'96)*, pages 189–196, Philadelphia, Penn., May 1996. ACM SIGARCH.
- [83] F. Desprez. A Library for Coarse Grain Macro-Pipelining in Distributed Memory Architectures. In *IFIP 10.3 Conference on Programming Environments for Massively Parallel Distributed Systems*, pages 365–371. Birkhaeuser Verlag AG, Basel, Switzerland, 1994.
- [84] F. Desprez. *Procédures de Base pour le Calcul Scientifique sur Machines Parallèles à Mémoire Distribuée*. PhD thesis, Institut National Polytechnique de Grenoble, January 1994. LIP ENS-Lyon.
- [85] F. Desprez and S. Domas. Efficient Pipelining of Level 3 BLAS Routines. In *4th international meeting VECPAR 2000*, volume 3, pages 675–688, Porto, June 2000.
- [86] F. Desprez, S. Domas, J. Dongarra, A. Petitet, C. Randriamaro, and Y. Robert. More on Scheduling Block-Cyclic Array Redistribution. In *Proc. of 4th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98)*, volume 1511 of *Lecture Notes in Computer Science*, pages 275–287. Springer-Verlag, Pittsburgh, PA, 1998.
- [87] F. Desprez, J. Dongarra, A. Petitet, C. Randriamaro, and Y. Robert. Block-cyclic Array Redistribution on Networks of Workstations. In M. Bubak, J. Dongarra, and J. Wasniewski, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, LNCS 1332, pages 343–350. Springer Verlag, 1997.
- [88] F. Desprez, J.J. Dongarra, A. Petitet, C. Randriamaro, and Y. Robert. Scheduling Block-cyclic Array Redistribution. *IEEE Transaction on Parallel and Distributed Systems*, 9(2) :192–205, 1998.
- [89] F. Desprez, J.J. Dongarra, F. Rastello, and Y. Robert. Determining the Idle Time of a Tiling : New Results. *Journal of Information Science and Engineering (Special Issue on Compiler Techniques for High-Performance Computing)*, 14(1) :167–190, March 1997.
- [90] F. Desprez, J.J. Dongarra, and B. Tourancheau. Performance Study of LU Factorization with Low Communication Overhead on Multiprocessors. *Parallel Processing Letters*, 5(2) :157–169, 1995.
- [91] F. Desprez, A. Ferreira, and B. Tourancheau. Efficient Communication Operations on Passive Optical Star Networks. In N. J. Davis, editor, *Massively Parallel Processing Using Optical Interconnections*, pages 52–58, Cancun, Mexico, April 1994. IEEE Computer Society Press.
- [92] F. Desprez, E. Fleury, C. Gomez, S. Steer, and S. Ubéda. Bringing Metacomputing to Scilab. In *Computer Aided Control System Design (CACSD 99)*, Hawaiï, USA, August 1999. IEEE.
- [93] F. Desprez, E. Fleury, F. Lombard, J.-M. Nicod, M. Quinson, and F. Suter. Une approche extensible des serveurs de calcul. *Soumission au numéro spécial metacomputing de Calculateurs Parallèles*, 2001.
- [94] F. Desprez, P. Fraigniaud, and B. Tourancheau. Successive Broadcast on Hypercube. Technical Report CS-93-210, The University of Tennessee - Knoxville USA, 1993.
- [95] F. Desprez and M. Garbey. Parallel Computing of a Combustion Front. In *Parallel CFD'93 - Implementations and Results Using Parallel Computers*. Elsevier Science Publishers B.V., 1993.
- [96] F. Desprez and M. Garbey. Numerical Simulation of a Combustion Problem on a Paragon Machine. *Parallel Computing*, 21 :495–508, 1995.
- [97] F. Desprez and M. Pourzandi. A Comparison of Three Matrix Product Algorithms on the Intel Paragon and Archipel Volvox Machines. In Bl. Sendov I.T.Dimov and P.S.Vassilevski, editors, *Advances in Numerical Methods and Applications NM&A - O(h<sup>3</sup>)'94*, pages 234–244. World Scientific, 1994.

- [98] F. Desprez, M. Quinson, and F. Suter. Dynamic Performance Modeling for Network Enabled Solvers in a Metacomputing Environment. In *Submitted to the seventh European Conference on Parallel Computing (Euro-Par 2001)*, 2001.
- [99] F. Desprez, P. Ramet, and J. Roman. Optimal Grain Size Computation for Pipelined Algorithms. In *Europar'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Science*, pages 165–172. Springer Verlag, August 1996.
- [100] F. Desprez and C. Randriamaro. Redistribution entre appels de routines ScaLAPACK. In *Actes des 12èmes Rencontres Francophones du Parallélisme RenPar'00*, pages 113–118, Besançon, France, June 2000.
- [101] F. Desprez and F. Suter. A Survey of High Performance Matlab-like Tools and Compilers. Ouragan White Paper <http://www.ens-lyon.fr/~desprez/OURAGAN/>, 2001.
- [102] F. Desprez and F. Suter. Mixed Parallel Implementations of the Top Level of Strassen and Winograd Matrix Multiplication Algorithms. In *In proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS'01)*, San Francisco, April 2001. To appear.
- [103] F. Desprez and B. Tourancheau. LOCCS : Low Overhead Communication and Computation Subroutines. *Future Generation Computer Systems*, 10(2&3) :279–284, June 1994.
- [104] F. Desprez and J. Zory. Performance des Macro-Pipelines dans les Programmes Data-Parallèles. In A. Schiper and D. Trystram, editors, *Actes des 9es Rencontres Francophones du Parallélisme RenPar'9*, pages 17–20, Lausanne, Switzerland, May 1997.
- [105] S. Domas. *Contribution à l'écriture et à l'extension d'une bibliothèque d'algèbre linéaire parallèle*. PhD thesis, ENS Lyon, 1997.
- [106] S. Domas, F. Desprez, and B. Tourancheau. Optimization of the ScaLAPACK LU Factorization Routine Using Communication/Computation Overlap. In *Europar'96 Parallel Processing*, volume 1124 of *Lecture Notes in Computer Science*, pages 3–10. Springer Verlag, August 1996.
- [107] J. Dongarra, P. Mayes, and G. Radicati di Brozolo. The IBM RISC System 6000 and Linear Algebra Operations. *Supercomputer*, 8 :15–30, 1991.
- [108] J. Dongarra, R. Pozo, and D. Walker. An Object Oriented Design for High Performance Linear Algebra on Distributed Memory Architectures. In *Proceedings of the Object Oriented Numerics Conference*, pages 257–264, 1993.
- [109] J. Dongarra and R. C. Whaley. LAPACK Working Note 94 : A User's Guide to the BLACS v1.0. Technical Report CS-95-281, The University of Tennessee - Knoxville, 1995.
- [110] Jack Dongarra and David Walker. The Quest for Petascale Computing. *Computing in Science and Engineering*, pages 22–29, May/June 2001.
- [111] J.J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. on Mathematical Soft.*, 16(1) :1–17, 1990.
- [112] J.J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An Extended Set of Fortran Basic Linear Algebra Subroutines. *ACM Trans. on Mathematical Soft.*, 14(1) :1–17, March 1988.
- [113] J.J. Dongarra, I.S. Duff, D.C. Sorensen, and H.A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers*. SIAM, 1998.
- [114] J.J. Dongarra, R. Van De Geijn, and D.W. Walker. A Look at Dense Linear Algebra Libraries. Technical Report ORNL/TM-12126, Oak Ridge National Laboratory, July 1992.
- [115] P. Drakenberg, P. Jacobson, and B. Kaagstrom. A CONLAB Compiler for a Distributed Memory Multicomputer. In R.F. Sincovec, D.E. Keyes, M.R. Leuze, L.R. Petzold, and D.A. Reed, editors, *Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 814–821. SIAM, 1993.

- [116] I.S. Duff, M. Marrone, and G. Radicati. A Proposal for User Level Sparse BLAS. Technical Report TR/PA/92/85, CERFACS - Toulouse, December 1992.
- [117] B. Dumitrescu, J.L. Roch, and D. Trystram. Fast Matrix Multiplication Algorithms on MIMD Architectures. Technical Report RT-87, LMC-IMAG, INPG Grenoble, France, July 1992.
- [118] J. W. Eaton. Octave : A High-Level Interactive Language for Numerical Computations. <http://www.octave.org/>.
- [119] R. Buyya (Ed.). *High Performance Cluster Computing*, volume 2 : Programming and Applications. Prentice Hall, 1999. ISBN 0-13-013784-7.
- [120] T. Von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauer. Active Message : a Mechanism for Integrated Communication and Computation. In ACM IEEE Computer Society, editor, *The 19th Annual Symposium on Computer Architecture*, pages 256–266. ACM Press, May 1992.
- [121] E.W. Evans, S.P. Johnson, P.F. Leggett, and M. Cross. Overlapped Communications Automatically Generated in a Parallelisation Tool. In B. Hertzberger and P. Sloot, editors, *Proceedings of High Performance Computing and Networking (HPCN'97)*, number 1225 in Lecture Notes in Computer Science, pages 801–810, Vienna, Austria, April 1997. Springer Verlag.
- [122] S. J. Fink and S. Baden. Non-Uniform Partitioning for Finite Difference Methods Running on SMP Clusters.
- [123] P. Fischer and R. Probert. Efficient Procedures for Using Matrix Algorithms. In *Automata, Languages and Programming*, volume 14 of Lecture Notes in Computer Science, pages 413–427. Springer-Verlag, Berlin, 1974.
- [124] MPI Forum. <http://www.mpi-forum.org/>.
- [125] I. Foster and C. Kesselman (Eds.). *The Grid : Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1998.
- [126] I. Foster and C. Kesselman. The Globus Project : A Status Report. In *Proceedings of the Heterogeneous Computing Workshop*, pages 4–18, 1998.
- [127] G. C. Fox, W. Furmanski, T. Haupt, E. Akarsu, and H. Ozdemir. HPcc as High Performance Commodity Computing on Top of Integrated Java, CORBA, COM and Web Standards. In D. Pritchard and J. Reeves, editors, *EuroPAR'98*, volume 1470 of *Lecture Notes in Computer Science*, pages 55–73, Southampton, UK, September 1998.
- [128] G.C. Fox, R.D. Williams, and P.C. Messina. *Parallel Computing Works !* Morgan Kaufmann Publisher, Inc., 1994. ISBN 1-55860-253-4.
- [129] K. Gallivan, B. Marsolf, W. Jalby, and A. Sameh. On the Development of Libraries and their use in Applications. In *NATO Series on Parallel Numerical Algorithms*, May 1995. Also Center for Supercomputing Research and Development (University of Illinois at Urbana-Champaign) Report 1341.
- [130] J. Garcia, E. Ayguadé, and J. Labarta. Dynamic Data Distribution with Control Analysis. In *Supercomputing'96*, November 1996.
- [131] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM : Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing*. The MIT Press, 1994.
- [132] A. George, J.R. Gilbert, and J.W.-H. Liu. *Graph Theory and Sparse Matrix Computation*. Springer Verlag, 1993.
- [133] G. H. Golub and C. F. Van Loan. *Matrix Computations*, volume 3 of *Johns Hopkins Series in the Mathematical Sciences*. The Johns Hopkins University Press, second edition, 1989.
- [134] C. Gomez, editor. *Engineering and Scientific Computing with Scilab*. Birkhäuser, 1999.
- [135] R.L. Graham, M. Grötschel, and L. Lovász. *Handbook of Combinatorics*. Elsevier, 1995.

- [136] B. Grayson and R. Van de Geijn. A High Performance Parallel Strassen Implementation. *Parallel Processing Letters*, 6(1) :3–12, 1996.
- [137] Portland Group. <http://www.pgroup.com/>.
- [138] A. Gupta, F. Gustavson, M. Joshi, G. Karypis, and V. Kumar. Design and Implementation of a Scalable Parallel Direct Solver for Sparse Symmetric Positive Definite Systems. In *Proc. of the 8th SIAM Conf. on Parallel Processing*, March 1997.
- [139] M. Gupta and P. Banerjee. Compile-Time Estimation of Communication Costs of Programs. In *Proc. of Intl. Parallel Processing Symposium*, pages 470–475, 1993. <http://www.ece.nwu.edu/cpdc/Paradigm/ipps92.gb.ps.Z>.
- [140] S. K. S. Gupta, S. D. Kaushik, C.-H. Huang, and P. Sadayappan. Compiling Array Expressions for Efficient Execution on Distributed-Memory Machines. *Journal of Parallel and Distributed Computing*, 32(2) :155–172, 1996.
- [141] K.A. Hawick, H.A. James, J.A. Mathew, and P.D. Coddington. Java Tools and Technologies for Cluster Computing. Technical Report DHP-077, Distributed and High Performance Computing Group, Dept. of Computer Science, Univ. of Adelaide, Australia, 1999.
- [142] G. Hedayat. Numerical Linear Algebra and Computer Architecture : An Evolving Interaction. Technical Report UMCS-93-1-5, Department of Computer Science, The University of Manchester, England, 1993.
- [143] P. Hénon, P. Ramet, and J. Roman. PaStiX : A Parallel Sparse Direct Solver Based on a Static Scheduling for Mixed 1D/2D Block Distributions. In *Proceedings of Irregular'2000, LNCS 1800*, pages 519–525. Springer Verlag, May 2000.
- [144] P. Henon, P. Ramet, and J. Roman. PaStiX : A Parallel Direct Solver for Sparse SPD Matrices based on Efficient Static Scheduling and Memory Management. In *SIAM Conference PPSC'2001, Portsmouth, Virginie, USA*, March 2001. <http://dept-info.labri.u-bordeaux.fr/~ramet/pastix/>.
- [145] K. Högsted, L. Carter, and J. Ferrante. Determining the Idle Time of a Tiling. In *Principles of Programming Languages*, pages 160–173. ACM Press, 1997. Extended version available as Tech. Rep. UCSD-CS96-489 at <http://www.cse.ucsd.edu/~carter/>.
- [146] N.J. Higham. Exploiting Fast Matrix Multiplication Within the Level 3 BLAS. Technical Report TR89-984, Dept of CS - Center of Applied Mathematics - Cornell University, April 1989.
- [147] S. Hiranandani, K. Kennedy, and C. Tseng. Evaluating Compiler Optimizations for Fortran D. *Journal of Parallel and Distributed Computing*, 21 :27–45, 1994.
- [148] S. Hiranandani, K. Kennedy, C.-W. Tseng, and S. Warren. Design and Implementation of the D Editor. In J.J. Dongarra and B. Tourancheau, editors, *Second Workshop on Environments and Tools for Parallel and Scientific Computing*, pages 1–10, Townsend, TN, May 1994. SIAM.
- [149] A. Hoisie, O. Lubeck, and H. Wasserman. Scalability Analysis of Multidimensional Wavefront Algorithms on Large-Scale SMP Clusters. In *Frontiers of Massively Parallel Computing*, 1999. [http://www.c3.lanl.gov/cic19/teams/par\\_arch/Web\\_papers/Wavefront/Wavefr%ont.html](http://www.c3.lanl.gov/cic19/teams/par_arch/Web_papers/Wavefront/Wavefr%ont.html).
- [150] E. N. Houstis and J. R. Rice. On the future of problem solving environments. <http://www.cs.purdue.edu/homes/jrr/pubs/kozo.pdf>, 2000.
- [151] I. A. Howes, M. C. Smith, and G. S. Good. *Understanding and deploying LDAP directory services*. Macmillan Technical Publishing, 1999.
- [152] HPF2. <http://dacnet.rice.edu/Depts/CRPC/HPFF/versions/hpf2/index.cfm>.
- [153] P. Husbands, C. Isbell, and A. Edelman. Interactive Supercomputing with MITatlab. <http://www-math.mit.edu/~edelman>.

- [154] S. Huss-Lederman, E. Jacobson, J. Johnson, A. Tsao, and T. Turnbull. Strassen's Algorithm for Matrix Multiplication : Modeling, Analysis, and Implementation. Technical Report CCS-TR-96-147, Center for Computing Sciences, Argonne National Lab., 1996.
- [155] C.S. Ierotheou, S.P. Johnson, M. Cross, and P.F. Leggett. Computer Aided Parallelisation Tools (CAPTools) - Conceptual Overview and Performance on the Parallelization of Structured Mesh Codes. *Parallel Computing*, 22 :163–195, 1996.
- [156] J.B. White III and S.W. Boya. Where's Overlap ? An Analysis of Popular MPI Implementations. In A. Skjeellum, P.V. Bangalore, and Y.S. Dandass, editors, *Proceedings of the Third MPI Developer's and User's Conference*, pages 1–6, Atlanta, Georgia, 1999. MPI Software Technology Press.
- [157] E.-J. Im and K. Yelick. Model-Based Memory Hierarchy Optimizations for Sparse Matrices. In *Workshop on Profile and Feedback-Directed Compilation*, pages 1–10, Paris, France, October 1998.
- [158] iMW. <http://www-unix.mcs.anl.gov/metaneos/softtools/imw.html>.
- [159] F. Irigoien, P. Jouvelot, and R. Triolet. Semantical Interprocedural Parallelization : An Overview of the PIPS Project. In *International Conference on Supercomputing*, Cologne, June 1991.
- [160] F. Irigoien and R. Triolet. Supernode Partitioning. In *Proc. 15th Annual ACM Symp. Principles of Programming Languages*, pages 319–329, San Diego, CA, January 1988.
- [161] Jini. <http://www.jini.org/>.
- [162] B. Kågström, P. Ling, and C. Van Loan. GEMM-Based Level 3 BLAS : High Performance Model Implementations and Performance Evaluation Benchmark. Technical Report UMINF-95.18, Umeå University, October 1995.
- [163] E. T. Kalns and L. M. Ni. Processor Mapping Techniques Towards Efficient Data Redistribution. *IEEE Trans. Parallel Distributed Systems*, 6(12) :1234–1247, 1995.
- [164] N. Kapadia, J. Robertson, and J. Fortes. Interfaces Issues in Running Computer Architecture Tools via the World Wide Web. In *Workshop on Computer Architecture Education at ISCA 1998*, Barcelona, 1998. <http://www.ecn.purdue.edu/labs/punch/>.
- [165] K.K. Kee and S. Hariri. Efficient Communication Algorithms for Pipeline Multicomputers. In *Proceedings of Supercomputing'94*, pages 468–477, Washington, D.C., Nov. 1994. IEEE Comp. Soc. Press.
- [166] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. *New User Interface for Petit and Other Extensions*. CS Dept, University of Maryland, April 1996. <http://www.cs.umd.edu/projects/omega>.
- [167] K. Kennedy, K.S. McKinley, and C.-W. Tseng. Interactive Parallel Programming Using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3) :329–341, July 1991.
- [168] K. Kennedy, N. Nedeljkovic, and A. Sethi. A Linear-Time Algorithm for Computing the Memory Access Sequence in Data-Parallel Programs. In *Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 102–111. ACM Press, 1995.
- [169] K. Kennedy, N. Nedeljkovic, and A. Sethi. Efficient Address Generation for Block-Cyclic Distributions. In *1995 ACM/IEEE Supercomputing Conference*. <http://www.supercomp.org/sc95/proceedings>, 1995.
- [170] C. W. Keßler. Applicability of Automatic Program Comprehension to Sparse Matrix Computations. In P. Fritzsion, editor, *Seventh International Workshop on Compilers for Parallel Computers*, pages 218–230, Linköping, Sweden, June 1998.
- [171] C. W. Keßler and C. H. Smith. The SPARAMAT Approach to Automatic Comprehension of Sparse Matrix Computations. In *Proc. 7th Int. Workshop on Program Comprehension*. Los Alamitos : IEEE Computer Society Press, May 1999.



- [172] C.-T. King, W.-W. Chou, and L.M. Ni. Pipelined Data-Parallel Algorithms : Part II – Design. *IEEE Trans. on Parallel and Distributed Systems*, 1(4) :486–499, 1990.
- [173] C.-T. King, W.-W. Chou, and L.M. Ni. Pipelined Data-Parallel Algorithms : Part1 – Concept and Modeling. *IEEE Trans. on Parallel and Distributed Systems*, 1(4) :470–485, 1990.
- [174] K. R. Koch, R. S. Baker, and R. E. Alcouffe. Solution of the First-Order Form of Threedimensional Discrete Ordinates Equations on a Massively Parallel Machine. *Transactions of the American Nuclear Society*, 65(198), 1992.
- [175] C.H. Koebel, D.B. Loveman, R.S. Schreiber, G.L. Steele Jr., and M.E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994. ISBN 0-262-11185-3.
- [176] U. Kremer. NP-Completeness of Dynamic Remapping. In *Fourth Workshop on Computers for Parallel Computers*, December 1993.
- [177] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction To Parallel Computing : Design and Analysis of Algorithms*. The Benjamin /Cummings Publishing Company, Inc., 1994. ISBN 0-8053-3170-0.
- [178] M. Valero-Garcia L. Diaz de Cerio and A. Gonzalez. Overlapping Communication and Computation in Hypercubes. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Second International EuroPar Conference*, number 1123 in Lecture Notes in Computer Science, pages 253–257, Lyon, France, August 1996. Springer Verlag.
- [179] P. Lacroute. *Fast Volume Rendering Using a Shear-Warp Factorization of The Viewing Transformation*. PhD thesis, Stanford University, 1995. <http://www-graphics.stanford.edu/~lacroute/>.
- [180] P. Lacroute. Real-Time Volume Rendering on Shared Memory Multiprocessors Using the Shear-Warp Factorization. In *Proceedings of the 1995 Parallel Rendering Symposium*, 1995. <http://www-graphics.stanford.edu/~lacroute/>.
- [181] A. Lain and P. Banerjee. Techniques to Overlap Computation and Communication in Irregular Iterative Applications. In *Supercomputing '94 : International conference*, pages 236–245, Manchester, UK, 1994. Univ of Manchester. <http://www.ece.nwu.edu/cpdc/Paradigm/ics94.lb.ps.Z>.
- [182] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. on Mathematical Soft.*, 5 :308–323, 1979.
- [183] C. Lee, Y.-F. W., and T. Yang. Global Optimization for Mapping Parallel Image Processing Tasks on Distributed Memory Machines. *Journal of Parallel and Distributed Computing*, 45 :29–45, 1997.
- [184] P. Lee. Efficient Algorithms for Data Distribution on Distributed Memory Parallel Computer. *IEEE Trans. Parallel Distributed Systems*, 8 :825–839, August 1997.
- [185] Legion. <http://legion.virginia.edu/>.
- [186] J. Li and M. Chen. Index domain alignment : Minimizing cost of cross-referencing between distributed arrays. In *Frontiers 90 : The 3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 424–433, College Park, MD, October 1990. IEEE Computer Society Press.
- [187] D. J. Lilja. The Impact of Parallel Loop Scheduling Strategies on Prefetching in a Shared-Memory Multiprocessor. *IEEE Transaction on Parallel and Distributed Systems*, 5(6) :573–584, June 1994.
- [188] B.-H. Lim and R. Bianchini. Limits on the Performance Benefits of Multithreading and Prefetching. Technical Report RC 20238, IBM T. J. Watson Research Center, 1995.
- [189] D. K. Lowenthal. Accurately Selecting Block Size at Run Time in Pipelined Parallel Programs. Submitted to the International Journal of Parallel Programming. <http://www.cs.uga.edu/~dkl/research/papers/blocksize.ps>.

- [190] D. K. Lowenthal and M. James. Run-Time Selection of Block Size in Pipelined Parallel Programs. In *Proceedings of the 2nd Merged IPPS/SPDP Conference*, pages 82–87, April 1999. <http://www.cs.uga.edu/~dkl/research/papers/ipp99.ps>.
- [191] A. Malishevsky, N. Seelam, and M.J. Quinn. Translating MATLAB Scripts into Parallel Programs Utilizing ScaLAPACK and Other Parallel Numerical Libraries. *Submitted to IEEE Trans. on Software Engineering*, 1999. Available at <http://www.cs.orst.edu/~quinn/matlab.html>.
- [192] The Mathworks Inc. *MATLAB Reference Guide*, 1992.
- [193] S. Matsuoka and H. Casanova. Network-Enabled Server Systems and the Computational Grid. <http://www.eece.unm.edu/~dbader/grid/WhitePapers/GF4-WG3-NES-whitepaper%-draft-000705.pdf>, July 2000. Grid Forum, Advanced Programming Models Working Group whitepaper (draft).
- [194] S. Matsuoka, H. Nakada, M. Sato, , and S. Sekiguchi. Design Issues of Network Enabled Server Systems for the Grid. <http://www.eece.unm.edu/~dbader/grid/WhitePapers/satoshi.pdf>, 2000. Grid Forum, Advanced Programming Models Working Group whitepaper.
- [195] K. McManus, S. Johnson, and M. Cross. Communication Latency Hiding in a Parallel Conjugate Gradient Method. In *Proc. Domain Decomposition 11*, Greenwich, July 1998. <http://www.gre.ac.uk/~k.mcmanus/doc/dd11-98.ps.gz>.
- [196] S. Miguet and Y. Robert. Elastic Load Balancing for Image Processing Algorithms. In H.P. Zima, editor, *Parallel Computation*. First International ACPC Conference, 1991.
- [197] C. Moler. Why There isn't a Parallel MATLAB. *MATLAB News and Notes*, April 1995.
- [198] E. Montagne, M. Ruloz, R. Suros, and F. Breant. Modeling Optimal Granularity when Adapting Systolic Algorithms to Transputer Based Supercomputers. *Parallel Computing*, 20 :807–814, 1994.
- [199] J. Moreira, S. Midkiff, and M. Gupta. From Flop to Megaflops : Java for Technical Computing. *ACM Transaction on Programming Languages and Systems*, 22(2) :265–295, March 2000.
- [200] G. Morrow and R. van de Geijn. A Parallel Linear Algebra Server for Matlab-like Environments. In *Supercomputing'98*, Orlando, Florida, November 1998. Available at <http://www.supercomp.org/sc98/>.
- [201] LAM MPI. <http://www.mpi.nd.edu/lam/>.
- [202] MPICH. <http://www-unix.mcs.anl.gov/mpi/mpich/>.
- [203] MPICH-G. <http://www.hpclab.niu.edu/mpi/>.
- [204] Myricom. <http://www.myri.com/>.
- [205] H. Nakada, H. Takagi, S. Matsuoka, U. Nagashima, M. Sato, and S. Sekiguchi. Utilizing the Metaserver Architecture in the Ninf Computing System. In P. Sloot et al., editor, *High Performance Computing and Networking (HPCN)*, number 1401 in Lecture Notes in Computer Science, pages 605–616, 1998.
- [206] R. Namyst. *PM2 : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières*. PhD thesis, Université de Lille 1, 1997.
- [207] R. Namyst and J.-F. Méhaut. PM2 : Parallel multithreaded machine. a computing environment for distributed architectures. In *Parallel Computing (ParCo '95)*, pages 279–285. Elsevier Science Publishers, September 1995.
- [208] NASoftware. <http://www.nasoftware.co.uk/home.html>.
- [209] M. O. Neary, A. Phipps, S. Richman, and P. Cappello. Javelin 2.0 : Java-Based Parallel Computing on the Internet. In A. Bode et al., editor, *Proceedings of EuroPAR'2000*, volume 1900 of *Lecture Notes in Computer Science*, pages 1231–1238. Springer Verlag, 2000.
- [210] NEOS. <http://www-neos.mcs.anl.gov/>.

- [211] T. Nguyen, M. Mills Strout, L. Carter, and J. Ferrante. Asynchronous Dynamic Load Balancing of Tiles. In *Proc. of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, March 1999. <http://www-cse.ucsd.edu/users/mstrout/SIAM99.ps>.
- [212] NIMROD. <http://www.csse.monash.edu.au/~davida/nimrod.html/>.
- [213] NINF. <http://ninf.etl.go.jp/>.
- [214] C.N. Ojeda-Guerra and A. Suárez. Solving Linear Systems of Equations Overlapping Communications and Computations in Torus Networks. In *Fifth EuroMicro Workshop on Parallel and Distributed Processing (PDP'97)*, pages 453–460. IEEE Computer Society Press, January 1997.
- [215] OpenMP. <http://www.openmp.org/>.
- [216] PACXMPI. <http://www.hlrs.de/structure/organisation/par/projects/pacx-mpi/>.
- [217] D. J. Palermo and P. Banerjee. Automatic Selection of Dynamic Data Partitioning Schemes for Distributed-Memory Multicomputers. In *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing*, Columbus, OH., Aug. 1995. <http://www.ece.nwu.edu/cpcd/Paradigm/lcpc95.pb.ps.Z>.
- [218] D.J. Palermo. *Compiler Techniques for Optimizing Communications and Data-distribution for Distributed Memory Multicomputers*. PhD thesis, University of Illinois at Urbana-Champaign, 1996. <ftp://ftp.crhc.uiuc.edu/pub/Paradigm/phd96.p.ps.Z>.
- [219] S. Pawletta, A. Westphal, T. Pawletta, W. Drewelow, and P. Duenow. *Distributed and Parallel Application Toolbox (DP Toolbox) for use with (MATLAB(r))*. Institute of Automatic Control, University of Rostock and Department of Mechanical Engineering, FH Wismar, Germany, version 1.4 edition, March 1999. Available at <http://www-at.E-technik.uni-rostock.de/dp/>.
- [220] J.-L. Pazat. Tools for High Performance Fortran : A Survey. Technical report, IRISA, Rennes, France, 1996. <http://www.irisa.fr/pampa/HPF/survey.html>.
- [221] M. Pourzandi. *Etude de l'impact des recouvrements calcul/communication sur des algorithmes parallèles de calcul matriciel*. PhD thesis, Ecole normale supérieure de Lyon, January 1995.
- [222] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipes in C – The Art of Scientific Computing*. Cambridge University Press, 1988.
- [223] L. Prylli and B. Tourancheau. Fast Runtime Block-Cyclic Data Redistribution on Multiprocessors. *J. Parallel Distributed Computing*, 45 :63–72, 1997.
- [224] L. Prylli, B. Tourancheau, and R. Westrelin. Modeling of a High Speed Network to Maximize Throughput Performance : the Experience of BIP over Myrinet. In H.R. Arabnia, editor, *Parallel and Distributed Processing Techniques and Applications (PDPTA '98)*, volume II, pages 341–349, Las Vegas, USA, 1998. CSREA Press.
- [225] M. J. Quinn and P. J. Hatcher. On the Utility of Communication–Computation Overlap in Data-Parallel Programs. *Journal of Parallel and Distributed Computing*, 33(02) :197, 1996.
- [226] J. Ramanujam and P. Sadayappan. Tiling of Iteration Spaces for Multicomputers. In *Proc. Internal Conference on Parallel Processing*, volume 2, pages 179–186, August 1990.
- [227] S. Ramaswamy, E.W. Hodges IV, and P. Banerjee. Compiling MATLAB Programs to ScaLAPACK : Exploiting Task and Data Parallelism. In *International Parallel Processing Symposium (IPPS'96)*, pages 613–619, April 1996.
- [228] S. Ramaswamy. *Simultaneous Exploitation of Task and Data Parallelism in Regular Scientific Applications*. PhD thesis, Univ. of Illinois, Urbana-Champaign, 1996.
- [229] P. Ramet. Calcul de la suite optimale de taille de paquets pour la factorisation de cholesky. In *ACTES RenPar'9*, pages 111–114, 1997.
- [230] P. Ramet. *Optimisation de la Communication et de la Distribution des Données pour les Solveurs Parallèles Directs en Algèbre Linéaire Dense et Creuse*. PhD thesis, Université de Bordeaux I, Janvier 2000.

- [231] C. Randriamaro. *Optimisation des redistributions et allocation dynamique en parallélisme de données*. PhD thesis, Ecole normale supérieure de Lyon, Janvier 2000.
- [232] T. Rauber and G. Rünger. Scheduling of Data Parallel Modules for Scientific Computing. In *Proceedings of 9th SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Mar 1999.
- [233] K. Remington and R. Pozo. NIST Sparse BLAS User's Guide. Technical report, National Institute of Standards and Technology, 1996. <ftp://gams.nist.gov/pub/karin/spblas/uguide.ps.gz>.
- [234] C. René and T. Priol. MPI Code Encapsulation using Parallel CORBA objets. In *Proceedings of HPDC8*, pages 3–10, 1999.
- [235] Y. Robert. *The Impact of Vector and Parallel Architectures on the Gaussian Elimination Algorithm*. Manchester University Press - Manchester - UK, 1990.
- [236] D.F. Robinson, X.-He Sun, and R.J. Enbody. A Pipelined Parallel Approach to Solving Dense Linear Systems. In *Proc. of the fourth Conf. on Hypercubes, Concurrent Computers and Applications*, pages 441–444, Monterey, CA, 1989.
- [237] L. De Rose and D. Padua. A MATLAB to Fortran 90 Translator and its Effectiveness. In *Proceedings of the 10th ACM International Conference on Supercomputing - ICS'96*, Philadelphia, PA, May 1996. Also Tech. Rep. Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, CSR-1462.
- [238] Y. Saad. SPARSEKIT : A Basic Toolkit for Sparse Matrix Computations. Technical report, NASA Ames Research Center, 1990.
- [239] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS publishing, New York, 1996.
- [240] B. S. Siegell. *Automatic Generation of Parallel Programs with Dynamic Load Balancing for a Network of Workstation*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 1995. Also appears as a tech. report CMU-CS-95-168.
- [241] B.S. Siegell and P. Steenkiste. Automatic Selection of Load-Balancing Parameters using Compile-Time and Run-Time Information. *Concurrency - Practice and Experience*, 9(3) :275–317, 1996. <http://almond.srv.cs.cmu.edu/afs/cs/project/cmcl/archive/GNectar-papers%/96cpe.ps>.
- [242] J.G. Siek and A. Lumsdaine. The Matrix Template Library : A Generic Programming Approach to High Performance Numerical Linear Algebra. In *POOSC '98 (Workshop of ECOOP)*, 1998. <http://www.lsc.nd.edu/research/mtl/>.
- [243] G.-A. Silber and A. Darte. The Nestor library : A tool for implementing Fortran source to source transformations. In *High Performance Computing and Networking (HPCN'99)*, volume 1593 of *Lecture Notes in Computer Science*, pages 653–662. Springer Verlag, April 1999. <http://cri.enscm.fr/~silber/biblio/HPCN99.ps.gz>.
- [244] Simulog. <http://www.simulog.fr/>.
- [245] J. Skeppstedt and M. Dubois. Compiler Controlled Prefetching for Multiprocessors Using Low-Overhead Traps and Prefetch Engines. *Journal of Parallel and Distributed Computing*, 60 :585–615, 2000.
- [246] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI : The Complete Reference*. The MIT Press, 1996. <http://www.netlib.org/utk/papers/mpi-book/mpi-book.html>.
- [247] A. Sohn and R. Biswas. Communication Studies of DMP and SMP Machines. Technical Report NAS-97-004, NAS, 1997. <http://www-sci.nas.nasa.gov/Pubs/TechReports/NASreports/NAS-97-005/NAS-%97-005.ps>.
- [248] A. Sohn, J. Ku, Y. Kodama, M. Sato, H. Sakane, H. Yamana, S. Sakai, and Y. Yamaguchi. Identifying the Capability of Overlapping Computation with Communication. In *Proc. of PACT'96*, pages 133–138, Boston, Mass., Oct 1996. North-Holland Pub. Co. <http://www.cis.njit.edu/~sohn/papers/pact96.ps.gz>.

- [249] J. M. Stichnoth, D. O'Hallaron, and T. R. Gross. Generating Communications for Array Statements : Design, Implementation, and Evaluation. *Journal of Parallel and Distributed Computing*, 21(1) :150–159, 1994.
- [250] V. Strassen. Gaussian Elimination Is Not Optimal. *Numerische Mathematik*, 14(3) :354–356, 1969.
- [251] T. Suzumura, T. Nakagawa, S. Matsuoka, H. Nakada, and S. Sekiguchi. Are Global Systems Useful? Comparison of Client-Server Global Computing Systems Ninf, Netsolve versus CORBA. In *14th International Parallel and Distributed Processing Symposium (IPDPS)*, Cancun, Mexico, May 2001. IEEE Computer Society.
- [252] Y. Tanaka, M. Matsuda, K. Kubota, and M. Sato. Performance Improvement by Overlapping Computation and Communication on SMP Clusters. In *Proc. of Int'l Conf. PDPTA'98*, volume 1, pages 275–282, 1998. <http://www.rwcp.or.jp/lab/pdperf/papers/pdpta98.ps.gz>.
- [253] A. Thirumalai and J. Ramanujam. Fast Address Sequence Generation for Data-Parallel Programs Using Integer Lattices. In C.-H. Huang, P. Sadayappan, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, volume 1033 of *Lectures Notes in Computer Science*, pages 191–208. Springer Verlag, 1995.
- [254] M. Thottethodi, S. Chatterjee, and A. Lebeck. Tuning Strassen's Matrix Multiplication for Memory Efficiency. In *Proceedings of Supercomputing'98*, Orlando, Nov 1998.
- [255] Top500. <http://www.top500.org/>.
- [256] A.E. Trefethen, V.S. Menon, C.C. Chang, G.J. Czajkowski, C. Myers, and L.N. Trefethen. MultiMatlab : Matlab on Multiple Processors. Technical Report 96-239, Cornell Theory Center, 1996. Available at <http://www.cs.cornel.edu/Info/People/Int/multimatlab.html>.
- [257] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Rice University, January 1993.
- [258] A.J. van der Steen. *Aspects of Computational Science - A Text Book on High-Performance Computing*. National Computing Facilities Foundation, 1995. ISBN 900-70608-33-2.
- [259] K. van Reeuwijk, W. Denissen, H. J. Sips, and E. M.R.M. Paalvast. An Implementation Framework for HPF Distributed Arrays on Message-Passing Parallel Computer Systems. *IEEE Trans. Parallel Distributed Systems*, 7(9) :897–914, 1996.
- [260] A. Wakatani and M. Wolfe. A New Approach to Array Redistribution : Strip-Mining Redistribution. In *Proceedings of Parallel Architectures and Languages Europe (PARLE 94)*, pages 323–335, July 1994. <http://www.cse.ogi.edu/Sparse/paper/wakatani.parle.94.ps>.
- [261] A. Wakatani and M. Wolfe. Effectiveness of Message Strip-Mining for Regular and Irregular Communication. In *Intl. Conference on Parallel and Distributed Computing Systems*, Oct. 1994. <http://www.cse.ogi.edu/Sparse/paper/wakatani.pdcs.94.ps>.
- [262] A. Wakatani and M. Wolfe. Redistribution of Block-Cyclic Data Distributions Using MPI. *Parallel Computing*, 21(9) :1485–1490, 1995.
- [263] D. W. Walker and S. W. Otto. Redistribution of Block-Cyclic Data Distributions Using MPI. *Concurrency : Practice and Experience*, 8(9) :707–728, 1996.
- [264] L. Wang, J. M. Stichnoth, and S. Chatterjee. Runtime Performance of Parallel Array Assignment : An Empirical Study. In *1996 ACM/IEEE Supercomputing Conference*, 1996. <http://www.supercomp.org/sc96/proceedings>.
- [265] R. Y. Wang, A. Krishnamurthy, R. P. Martin, T. E. Anderson, and D. E. Culler. Towards a Theory of Optimal Communications Pipelines. Technical Report CSD-98-981, University of California, Berkeley, January 26, 1998.
- [266] WebFlow. <http://www.npac.syr.edu/users/haupt/WebFlow/>.
- [267] R.C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. In *Proceedings of Supercomputing '98*. ACM SIGARCH and IEEE Computer Society, 1998. Also LAPACK Working Note N° 131, <http://www.netlib.org/atlas/>.

- [268] M. Wolfe. Iteration Space Tiling for Memory Hierarchies. In Gary Rodrigue, editor, *Proceedings of the 3rd Conference on Parallel Processing for Scientific Computing*, pages 357–361, Philadelphia, PA, USA, December 1989. SIAM Publishers.
- [269] S. Wolfram. *The Mathematica Book*. Wolfram Median, Inc. and Cambridge University Press, third edition, 1996.
- [270] R. Wolski, N. T. Spring, and J. Hayes. The Network Weather Service : A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computing Systems, Metacomputing Issue*, 15(5–6) :757–768, Oct. 1999.
- [271] XtremWeb. <http://www.lri.fr/~fedak/XtremWeb/introduction.php3>.
- [272] C.-W. Xu and B. He. PCB - A Distributed Computing System in CORBA. *Journal of Parallel and Distributed Computing*, 60 :1293–1310, 2000.
- [273] H. Zima and B. Chapman. Compiling for Distributed Memory Systems. Technical report, Institute for Statistics and Computer Science, Vienna, Austria, May 1994.
- [274] J. Zory. *Contributions à l'optimisation de programmes scientifiques*. PhD thesis, Ecole des Mines de Paris, December 1999.



# Table des figures

1.1	Architecture logicielle de ScaLAPACK. . . . .	8
1.2	Architecture logicielle de PETSc. . . . .	8
2.1	Boucle DOACCROSS séquentielle. . . . .	17
2.2	Boucle DOACCROSS parallèle avec distribution par blocs. . . . .	17
2.3	Diagramme de Gantt d'un programme pipeline entre deux processeurs. . . . .	18
2.4	Programme pipeline entre plusieurs processeurs. . . . .	18
2.5	ADI séquentiel. . . . .	19
2.6	ADI pipeline. . . . .	19
2.7	Algorithme séquentiel pour le SOR. . . . .	20
2.8	Détail de l'exécution du SOR sur une frontière. . . . .	20
2.9	Répartition de la charge de l'image intermédiaire en fonction de l'angle de rotation. . . . .	21
2.10	Algorithme de référence pour les différentes méthodes de recouvrement. . . . .	23
2.11	Algorithme de référence après la première optimisation. . . . .	23
2.12	Pipeline mono-dimensionnel avec une matrice 2D distribuée sur une grille de processeurs. . . . .	26
2.13	Pipeline bi-dimensionnel avec une matrice 3D distribuée sur un tore 3D de processeurs. . . . .	26
2.14	Trois façons de découper un produit matriciel. . . . .	28
2.15	Pipeline de cinq produits matriciels. . . . .	28
2.16	Diagramme de Gantt pour les quatre solutions de découpage pour le pipeline de cinq produits de matrices. . . . .	28
2.17	Schéma pipeline entre 2 processeurs. . . . .	30
2.18	Interfaces OPIUM pour le cas linéaire et pour le cas générique. . . . .	32
2.19	Temps de factorisation par bloc en fonction de la taille des paquets. . . . .	33
2.20	Temps de factorisation par bloc en fonction du nombre de processeurs. . . . .	33
2.21	Temps du SOR en fonction de la taille des paquets. . . . .	33
2.22	Temps du SOR en fonction du nombre de processeurs. . . . .	33
2.23	Parallélisation de l'ADI avec redistribution ou utilisation des LOCCS en HPF. . . . .	34
2.24	Facteur d'accélération de l'ADI avec redistribution ou utilisation des LOCCS en HPF. . . . .	35
3.1	Placement de données à deux niveaux dans HPF. . . . .	39
3.2	Distributions de matrices cycliques, blocs et blocs cycliques. . . . .	40
3.3	Ordonnement des communications pour l'exemple $P = 6, r = 2, Q = 6$ et $s = 3$ (sans entrelacement des étapes de communication). . . . .	49
3.4	Ordonnement des communications pour l'exemple $P = 6, r = 2, Q = 6$ et $s = 3$ (avec entrelacement des étapes de communication). . . . .	49
3.5	Temps d'exécution de la routine PDGEMR2D en fonction de la taille des blocs (grilles $1 \times 4$ à $4 \times 1$ ). . . . .	52
3.6	Temps d'exécution de la routine PDGEMM en fonction de la taille des blocs (4 processeurs, matrices $1000 \times 1000$ ). . . . .	53



3.7	Temps d'exécution de la routine PDTRSM en fonction de la taille des blocs (4 processeurs, matrices $1000 \times 1000$ ). . . . .	53
3.8	Cycle de développement d'une application sur une plate-forme à mémoire distribuée. . . . .	55
3.9	Vue des dépendances dans la première version de TRANSTOOL. . . . .	57
3.10	Deux fenêtres d'HPFize V1 (distribution et templates). . . . .	58
3.11	L'interface d'aLASca intégrée à TransTool montrant un code Fortran séquentiel avec son graphe de succession. . . . .	59
3.12	L'interface d'aLASca intégrée à TransTool montrant le code Fortran parallèle correspondant à la figure 3.11, avec son graphe de succession. . . . .	60
3.13	L'interface d'aLASca intégrée à TransTool montrant le code HPF correspondant à la figure 3.11, avec son graphe de succession. . . . .	60
4.1	Architecture générale de SCILAB//. . . . .	67
4.2	Utilisation de PDGEMM dans SCILAB//. . . . .	70
4.3	Produit de matrice parallèle avec surcharge de l'opérateur * dans SCILAB. . . . .	71
4.4	Performances du produit de matrices en utilisant la surcharge d'opérateur distribué * dans SCILAB//. . . . .	72
4.5	Produit de matrices complexes entre Nancy et Bordeaux en utilisant NETSOLVE. . . . .	73
4.6	Vue générale de SLiM et FAST. . . . .	75
4.7	Comparaison des prédictions de FASTet de NETSOLVE 1.3 par rapport au temps mesuré. . . . .	76
4.8	Vue générale de DIET. . . . .	76
4.9	Initialisation d'un système DIET. . . . .	76

# Liste des tableaux

2.1	Quatre cas pour le problème du recouvrement. . . . .	24
2.2	Complexités des différents recouvrements. . . . .	24
2.3	Gains asymptotiques des différents recouvrements. . . . .	24
2.4	Gain avec $N_d$ dimensions distribuées. . . . .	27
2.5	Solution 4 : codes pour les tâches 1 et 2. . . . .	29
2.6	Solution 4 : codes pour les tâches 3, 4 et 5. . . . .	29
3.1	Grille de communication pour $P = Q = 16, r = 3$ , et $s = 5$ . La taille des messages indiquée est calculée pour la redistribution d'un vecteur de taille $L = 240$ . . . . .	45
3.2	Les étapes de communication pour $P = Q = 16, r = 3$ , et $s = 5$ . . . . .	45
3.3	Les coûts de communication pour $P = Q = 16, r = 3$ , et $s = 5$ . . . . .	45
3.4	Grille de communication pour $P = Q = 16, r = 7$ , et $s = 11$ . La taille des messages indiquée est calculée pour la redistribution d'un vecteur de taille $L = 1232$ . . . . .	46
3.5	Les étapes de communication pour $P = Q = 16, r = 7$ , et $s = 11$ . . . . .	46
3.6	Les coûts de communication pour $P = Q = 16, r = 7$ , et $s = 11$ . . . . .	46
3.7	Grille de communication pour $P = Q = 15, r = 3$ , et $s = 5$ . La taille des messages indiquée est calculée pour la redistribution d'un vecteur de taille $L = 225$ . . . . .	47
3.8	Les étapes de communication pour $P = Q = 15, r = 3$ , et $s = 5$ . . . . .	47
3.9	Les coûts de communication pour $P = Q = 15, r = 3$ , et $s = 5$ . . . . .	47
3.10	Grille de communication pour $P = 12, Q = 8, r = 4$ , et $s = 3$ . La taille des messages indiquée est calculée pour la redistribution d'un vecteur de taille $L = 48$ . . . . .	48
3.11	Les étapes de communication pour $P = 12, Q = 8, r = 4$ , et $s = 3$ . . . . .	48
3.12	Les coûts de communication pour $P = 12, Q = 8, r = 4$ , et $s = 3$ . . . . .	48
3.13	Grille de communication pour $P = Q = 6, r = 2$ , et $s = 3$ . La taille des messages indiquée est calculée pour la redistribution d'un vecteur de taille $L = 36$ . . . . .	48
3.14	Récapitulatif des différents algorithmes de distribution automatique. . . . .	52
4.1	Projets autour de la parallélisation de langages de type MATLAB. . . . .	68
4.2	Projets qui conservent l'interactivité de MATLAB. . . . .	68
4.3	Opérations surchargées pour les matrices distribuées dans SCILAB//. . . . .	71
4.4	Connaissances nécessaires pour l'agent. . . . .	75