Numéro d'ordre : 233
Numéro attribué par la bibliothèque : 02ENSLO 233

École Normale Supérieure de Lyon
Laboratoire de l'Informatique du Parallélisme

THÈSE

pour obtenir le grade de

Docteur de l'École normale supérieure de Lyon

Spécialité : **Informatique**

au titre de l'école doctorale de : Mathématiques et Informatique fondamentale

présentée et soutenue publiquement le 13/11/2002

par Monsieur Frédéric Suter

Parallélisme mixte et prédiction de performances sur réseaux hétérogènes de machines parallèles

Directeur de thèse : DESPREZ Frédéric

Après avis de : Monsieur Michel Daydé, Membre/Rapporteur

Monsieur Jean Roman, Membre/Rapporteur

Devant la Commission d'examen formée de :

Monsieur Michel DAYDÉ, Membre/Rapporteur

Monsieur Frédéric DESPREZ, Membre

Monsieur Hervé GUYENNET, Membre

Madame Brigitte PLATEAU, Membre

Monsieur Jean ROMAN, Membre/Rapporteur

à Caro

Remerciements

L'histoire de cette thèse et donc de mon entrée dans le monde merveilleux de la recherche peut se résumer comme une suite de concours de circonstances, de hasards et de choix guidés par ma fainéantise.

Tout à commencé, je crois, lorsque mes parents ont déménagé et que j'ai du changer de collège. J'ai abandonné dans ce déménagement le peu d'esprit de compétition scolaire que j'avais. Dans ce nouveau collège, j'ai eu la joie d'avoir une formidable professeur de sciences naturelles qui m'a fait aimer sa discipline. Du coup, en arrivant au lycée, ma décision était prise : je voulais faire de la biologie ! Malheureusement, les enseignants se suivent mais ne se ressemblent pas. Mon intérêt pour la biologie, qui entre temps m'avait fait atterrir en terminale D, s'est détourné vers les mathématiques. Ne suivant que mes résultats du bac, je me suis donc inscrit en DEUG MIAS.

Ce DEUG, outre le fait qu'il m'a permis de comprendre que je n'était définitivement pas fait pour les mathématiques d'un niveau supérieur à la terminale, m'a initié à l'informatique. Heureusement pour la suite, cette découverte s'est faite par l'intermédiaire d'un des professeurs les plus pédagogues qu'il m'ait été donné de rencontrer : M. Philippe Moreau. Cela a conditionné mon troisième (et oui déjà !) changement d'orientation, me poussant des mathématiques vers l'informatique.

Surviennent alors deux personnes très importantes pour la suite du débat. Tout d'abord, David, mon binôme du DEUG à la maîtrise. Sans lui (et le 22/20 en projet), pas de DEUG, donc pas licence, ni de maîtrise, DEA et thèse. Ensuite, il y a Christophe, qui voulait faire de la recherche alors que je voulais aller dans le privé. Il a fini développeur et moi en DEA ! !

Ça y est, j'y suis presque. Voyons maintenant ce qui m'a vraiment fait basculer du côté obscur des études. Tout à commencé lors du module « étude et recherche » de la maîtrise. Deux de mes professeurs, Dominique et Gil pour ne pas les nommer, avaient pondu toute une série de sujets attractifs pour avoir le plus d'étudiants possible. Pour ma part, j'avais opté pour « Autour de TCL/Tk », tout un programme ! Mais, à la première réunion, nous avons tous eu droit au discours suivant : « Bon ! On a changé les sujets. La règle est simple, nouveau sujet = module, ancien sujet = pas module ! » Je suis donc retrouvé avec « Compatibles MATLAB » comme nouveau sujet. C'est quoi MATLAB ! ? ! ? L'objectif était de savoir s'il était possible de paralléliser, voire d'adapter au out-of-core, un outil comme MATLAB, mais gratuit. J'ai donc commencé à regarder des choses comme Octave et un certain SCILAB (alors prononcé SkyLab).

Comme cette (petite) étude m'avait bien plus en maîtrise, j'ai demandé s'il ne serait pas possible de continuer en stage de DEA. Dominique, Gil et Eddy (déjà) m'ont alors fait un sujet rien que pour moi et ont alors scellé mon destin. J'avais mis le premier pied dans le monde de SCILAB$_{//}$. Tout s'est alors accéléré. Éric Fleury est venu à Amiens et a annoncé qu'il y avait bientôt une réunion sur SCILAB$_{//}$ à Bordeaux. Dominique pensait que c'était une bonne occasion de présenter ce que l'on voulait faire. Hélas (pour lui) cela tombait le même jour qu'une autre réunion. Me voilà donc envoyé à Bordeaux, tout seul comme un grand, pour défendre les intérêts amiénois. Après une longue journée de débats sur duplication vs. serveurs de calculs, nous avons fini au restaurant. Un conseil : ne laissez jamais une bande de DR INRIA aller dans un restaurant où le vin est bon, ou alors laissez les payer ! A la fin du repas, on m'a demandé ce que je comptais faire

l'année suivante. J'ai répondu que je ferais bien une thèse, mais que je cherchais un financement. Éric a alors eu les mots magiques : « Demande à Fred, c'est lui qui a les sous (déjà) ». Une soutenance de DEA (bien défendue ;-), une femme merveilleuse, et un dossier envoyé par fax et me voilà débarquant à Lyon pour trois années de thèse.

Maintenant que vous savez tous comment tout à commencé, nous allons pouvoir passer aux remerciements d'usage.

Commençons tout d'abord par les membres du jury : Michel Daydé et Jean Roman pour avoir accepté d'être mes rapporteurs et de me donner leur avis sur mes travaux ; Hervé Guyennet et Brigitte Plateau pour avoir fait le déplacement jusqu'à Lyon pour assister à ma soutenance.

Viennent ensuite les différents co-bureaux qui se sont succédés au cours de ces trois années. Merci donc à Nicolas, Fabrice puis Guillaume pour avoir tous rédigé les uns à la suite des autres, me laissant seul dans mon bureau la plupart du temps. Il serait cependant injuste de limiter leur apport à leur absence. Nicolas et ses conseils m'ont été d'une aide précieuse pour la préparation de mon départ aux US. Jamais je n'oublierai la délicate odeur des pâtes au bleu de Fabrice que j'ai savourée une après-midi entière. Enfin merci à Guillaume pour tous les goodies Aubade qui donnaient une certaine chaleur au bureau. Passons ensuite au bureau 302 où j'ai passé ma dernière année. Merci à Vincent d'avoir été si souvent présent alors qu'il avait déjà tout fini et à Olivier d'avoir supporté notre bruyante présence et d'avoir assuré la déco du bureau (quelles belles photos !). Merci enfin à Hélène et Antoine pour le calme nécessaire à la dernière ligne droite.

Je souhaiterais ensuite remercier tous les acharnés de la baston du midi ! Merci donc à Hagrid, DrVince, Pilou, MadCow, Spawn, Schtroumph Killer, et les autres de m'avoir laissé rapporter quelques drapeaux avant de me tirer dans le dos. Mais prenez garde à retour de la clé à molette ! !

Un grand, que dis-je, un très grand merci à celles qui nous simplifie la vie de tous les jours. Je veux bien sûr parler des plus exceptionnelles secrétaires que l'on puisse trouver : Anne-Pascale, Sylvie, Corinne et Isabelle. Sans oublier Marie, qui m'a « forcé » à venir au coin café et la piscine. Je la remercie pour ma socialisation et mes kilos en moins.

Je n'oublie pas non plus toute l'équipe DIET avec qui ce fut un bonheur de travailler. Il me faut donc remercier Martin, Philippe, Christophe, Ludovic et les besac-boys. Par extension, j'associe aussi les membres de ReMap : Yves, pour la finesse de ses blagues, Arnaud et Vince pour les compétences Debian, Abdou pour m'avoir laissé arriver avant lui quelques fois. J'ajoute une mention spéciale aux picards du groupe : Eddy (et par transitivité, Isabelle et Naëlle) pour l'hébergement, les nuits Harry Potter, et la patience de me supporter ; et Fred, le directeur de thèse idéal qui prévoit tout à quelques années d'intervalle et qui n'a jamais hésité à mouiller le maillot (au sens propre comme au figuré) pour qu'on ait tous une vie plus facile. J'espère qu'un jour j'arriverai à le remercier suffisamment de sa confiance aveugle, peut-être en étant content de ce qui m'arrive.

Quelques remerciements vont aussi à la famille. À mes parents et beaux-parents pour leur soutien de tous les instants. À mon frère surtout, pour avoir été l'aîné, ce qui m'a permis de faire ce que je voulais alors que lui essuyait les plâtres pour moi.

Merci aussi à tous les copains de Lyon qui nous ont apporté tant de joie et d'amitié durant ce séjour : Séverine et Éric, Jean-Phi et Lorine, l'Hélène polaire, Violaine et Abdallah, Delphine et les autres.

Je terminerai enfin cette longue liste, par le remerciement le plus important à mes yeux. Il va tout naturellement à ma petite femme chérie sans qui je ne serais pas en train d'écrire tout cela. Merci Caro de m'avoir forcé à partir d'Amiens, de me suivre où que j'aille (sauf au Havre, je sais !) et de me faire le plus beau bébé de la terre. Sans toi, je ne serai pas grand chose.

Table des matières

Introduction

De tous temps, l'homme a voulu exécuter des calculs en parallèle. Après avoir constaté que ça ne marchait pas trop mal, l'homme c'est aperçu qu'en faisant du parallélisme mixte ça irait beaucoup mieux. Première version (refusée).

Avec la généralisation de l'Internet, il est désormais possible pour la plupart des utilisateurs de calcul numérique d'accéder aux machines les plus puissantes disponibles de par le monde, et ce depuis leur station de travail. Ce type d'accès distant peut se concevoir de plusieurs manières et à différentes échelles. Ainsi, au niveau d'un laboratoire ou d'une université, la présence de plus en plus répandue de grappes de PCs, encourage le développement d'interfaces entre des environnements de programmation de haut niveau, tels que Matlab ou SCILAB et ces machines parallèles. À une plus grande échelle, un réseau connectant les plus grands centres de calcul donne accès à une puissance totale cumulée de l'ordre du Teraflops et à des capacités de stockage de l'ordre du Petaflops. On parle alors de *metacomputing* ou de *grid computing*.

Toutefois, l'utilisation efficace de telles plates-formes reste encore une affaire de spécialistes, du fait de la forte hétérogénéité, de l'éloignement géographique et de la multiplication des autorisations d'accès engendré par ce type de plates-formes. Afin de masquer du mieux possible cette complexité aux utilisateurs, il est donc nécessaire de développer des environnements d'exécution adaptés au metacomputing. L'idéal est de fournir une facilité d'utilisation proche de celle des environnements de programmation de haut niveau précédemment cités. Une des approches de développement de tels environnements consiste à utiliser des serveurs de calcul, *i.e.*, des machines, le plus souvent parallèles, sur lesquelles sont installées des bibliothèques numériques performantes. Le système classique client-serveur, est dans ce cas remplacé par le modèle client-agents-serveurs. Ces agents sont des processus déployés de manière stratégique sur la plate-forme d'exécution et sont en charge de faire correspondre à une requête soumise par un client le ou les serveurs les mieux adaptés.

Un des points cruciaux des environnements de ce type concerne la capacité à estimer le temps d'exécution d'une routine sur machine donnée, d'une part, et les coûts de transfert des données depuis un client ou un serveur vers le serveur choisi pour la résolution d'une requête, d'autre

part. Pour être capable d'estimer, voire de prédire, ces différents temps, une technique consiste à déployer sur les machines et le réseau un ensemble de sondes logicielles chargées de surveiller les différentes variations de charge. Cette technique vient en complément d'une bonne connaissance des performances théoriques de la plate-forme d'exécution et ne sert qu'à pondérer les informations issues de cette connaissance selon l'état du système au moment du traitement d'une requête. Il est de plus nécessaire de connaître le comportement précis d'une routine numérique, typiquement une fonction d'une bibliothèque, en fonction de la machine, séquentielle ou parallèle, sur laquelle elle s'exécute. En effet, les optimisations apportées par les développeurs de bibliothèques pour tenir compte des spécificités architecturales des machines rendent les performances d'une routine donnée très variables d'une plate-forme à une autre. Pour gérer ce type de variations, il est possible de baser les estimations sur une phase préalable durant laquelle les différentes routines seront testées de manière extensive, dans le but d'appréhender leur comportement. Si ce genre d'approche est plutôt adapté aux machines séquentielles, il peut être étendu aux architectures parallèles en combinant ce type d'informations à une analyse des codes des routines parallèles. Cette analyse permet par exemple d'identifier les appels internes à des routines séquentielles, que l'on sait estimer par la technique précédente, ainsi que les schémas de communications inhérents aux algorithmes utilisés.

D'un point de vue plus algorithmique, la présence de bibliothèques numériques de plus en plus performantes sur des machines parallèles de plus en plus puissantes, et dont le nombre de processeurs ne cesse de croître, permet d'envisager l'exécution concurrente de calculs parallèles. Cela peut en effet exhiber plus de parallélisme qu'une utilisation simple du parallélisme de tâches ou du parallélisme de données. Pour cela, il est nécessaire de développer des algorithmes d'ordonnancement tenant à la fois compte des performances relatives d'une tâche en fonction du nombre de processeurs qui lui sont alloués et des coûts de communication induits par l'introduction du parallélisme.

Cette thèse se compose de quatre chapitres au cours desquels nous reviendrons sur les différents points évoqués précédemment :

Metacomputing : des PSEs aux ASPs

Cette thèse a démarrée au sein de l'Action de Recherche Coopérative INRIA OURAGAN. Ce premier chapitre présente les travaux réalisés au cours de cette ARC portants sur la parallélisation du logiciel SCILAB. Différentes approches ont été étudiées. Il est tout d'abord possible de compiler des codes vers des langages classiques, tels que C ou Fortran. Une seconde approche consiste à répliquer des processus SCILAB sur les nœuds d'une machine parallèle et à les faire communiquer au travers d'une bibliothèque de communication. Cette approche a été choisie pour une première implantation de SCILAB $_{//}$, par le biais d'une interface avec la bibliothèque de passage de messages PVM. Enfin, la dernière approche suit le modèle client–serveur et se base donc sur des serveurs de calcul sur lesquels sont installées des bibliothèques numériques efficaces. Du fait de l'évolution des architectures et des bibliothèques numériques, nos travaux sont naturellement dirigés vers cette solution.

Modélisation de routines parallèles et prédiction de performances

Dans ce chapitre, nous présenterons la bibliothèque FAST (Fast Agent's System Timer), développée par Martin Quinson au Laboratoire de l'Informatique du parallélisme, qui constitue un outil de prédiction dynamique de performances dans un environnement de metacomputing. Une extension de cet outil pour la gestion des routines parallèles a été réalisée au cours de cette thèse. Cette extension permet de prédire le temps d'exécution de routines parallèles d'algèbre linéaire dense en fonction des données impliquées, des caractéristiques

de la plate-forme d'exécution et notamment de la forme et de la taille de la grille de processeurs utilisée. Pour cela une combinaison entre une analyse du code de ces routines et les estimations fournies par FAST, concernant les équivalents séquentiels des routines et les disponibilités du réseau, est effectuée. Nous avons proposé des modèles construits de cette manière pour différentes routines, telles que le produit de matrices ou la résolution de système triangulaire. Ces modèles ont été validés expérimentalement et se sont avérés précis. De plus, ils permettent d'appréhender l'évolution des performances en fonction de la grille de processeurs utilisée, ce qui constituait l'objectif de départ. Nous montrerons enfin comment cet outil peut être utilisé par ordonnanceur pour décider où placer un calcul en fonction de la localisation des données du problèmes, des capacités relatives des machines de la plate-forme et des capacités du réseau d'interconnexion.

Parallélisme mixte

Ce chapitre constitue la partie la plus importante de mon travail de thèse. Le principe de base du parallélisme mixte consiste à exploiter simultanément les parallélismes de tâches et données présents dans une application. Nous nous sommes plus particulièrement intéressés à des graphes de tâches où chacun des nœuds peut être un calcul séquentiel ou utilisant le parallélisme de données. Comme nous l'avons indiqué précédemment, cette technique permet d'exhiber plus de parallélisme qu'en utilisant qu'une des deux formes classiques.

Afin de valider ce paradigme de programmation, nous avons étudié son application aux algorithmes rapides de produit de matrices de Strassen et Winograd. Ces algorithmes sont bien adaptés au parallélisme mixte, leurs graphes de tâches n'étant composés que de tâches parallélisables et ne comportant pas de boucle. Différentes implantations « mixtes » de ces algorithmes ont été réalisées ciblant aussi bien des plates-formes homogènes qu'hétérogènes. Nous avons, de plus, effectué une étude théorique de nos implantations, étude confirmée par une série d'expérimentations.

Nous avons ensuite cherché à automatiser le développement d'implantations mixtes à partir de graphes de tâches. Pour cela, nous avons proposé un algorithme d'ordonnancement en parallélisme mixte dans le cas où les données ne peuvent pas être répliquées. Cet algorithme effectue simultanément le placement et l'ordonnancement des tâches d'un graphe en se basant sur les modèles de coûts fournis par notre extension de FAST et sur un ensemble de distributions possibles.

DIET : une approche hiérarchique des serveurs de calcul

Les différents travaux menés au cours de l'ARC OURAGAN nous ont permis de détecter les lacunes des environnements à base de serveurs de calcul existants. L'un des inconvénients majeurs de ces environnements est sans nul doute l'agent réalisant la jonction entre clients et serveurs qui peut devenir un goulot d'étranglement. Afin de pallier ce problème, et donc de proposer un environnement extensible, nous avons suivi une approche hiérarchique pour développer le logiciel DIET (*Distributed Interactive Engineering Toolbox*). En effet, l'utilisation d'une hiérarchie d'agents permet non seulement d'éviter le phénomène de goulot d'étranglement, mais aussi de réduire le temps de propagation des requêtes depuis un client vers les serveurs. DIET utilise l'outil FAST pour estimer les temps de résolution de problèmes sur les différents serveurs, ainsi que pour évaluer les coûts de transfert de données au sein de l'architecture. Une série d'expérimentations visant à dégager des règles de base pour le déploiement d'une hiérarchie efficace seront également présentées.

Nous conclurons enfin la présentation de ces travaux par diverses perspectives de recherche et de développement.

Chapitre 1

Metacomputing : des PSEs aux ASPs

> Les Shadoks avaient de grosses pompes pour les gros problèmes et des petites pompes pour les petits problèmes. Ils avaient mis au point, aussi, des pompes spéciales pour les cas où il n'y avait pas de problème du tout.
> Devise Shadok.

1.1 Introduction

Depuis le début des années 90, il est devenu courant dans la communauté scientifique d'utiliser des outils interactifs de haut niveau pour développer et traiter des applications. Ces outils, le plus souvent spécifiques à un domaine de compétences particulier, sont couramment appelés *environnements de résolution de problèmes* ou *Problem Solving Environments* (PSE), en anglais. Matlab [77] et Scilab [59] font partie de ces PSEs et appartiennent au domaine des applications numériques. Cependant, des applications de grandes tailles, issues de la simulation numérique ne peuvent être exécutées avec de tels logiciels séquentiels. En effet, les limites matérielles, en termes d'espace mémoire et de puissance de processeur, ne permettent de traiter que des problèmes de tailles relativement réduites. De plus, les performances d'un langage interprété, peuvent s'avérer de une à dix fois plus faibles que celles obtenues en utilisant un langage compilé.

En revanche, ces problèmes numériques de très grande taille peuvent désormais être résolus via Internet grâce aux environnements de metacomputing [56]. En effet, l'évolution des architectures combinée à la généralisation des bibliothèques numériques performantes permettent aujourd'hui de disposer de la puissance de calcul des machines parallèles des grands centres de calcul internationaux depuis sa station de travail.

De plus, un nouveau type d'applications, basé sur la fourniture de services via Internet, commence à émerger dans la communauté de la recherche informatique. Le principe de base du

modèle ASP (*Application Service Provider*) [113] est de concevoir des applications de type client–serveur où le client serait réduit à son expression la plus simple tout en adaptant l'offre logicielle en fonction de l'utilisateur. Par exemple, la majorité des sites web permettent désormais de cibler leur contenu en fonction du profil de l'internaute qui le visite.

Au cours de ce chapitre, nous allons tout d'abord rappeler l'évolution des architectures parallèles qui nous a conduit à évoluer des PSEs aux ASPs. Puis nous donnerons un bref état de l'art des bibliothèques de communication et d'algèbre linéaire. Nous étudierons enfin les différents travaux effectués dans le cadre du développement de la version parallèle de l'outil SCILAB et donnerons les raisons qui nous ont conduits, à partir de ces travaux, à concevoir un ensemble d'outils utiles au développement d'applications conformes au modèle ASP.

1.2 Des super-calculateurs aux plates-formes de metacomputing

Les caractéristiques des processeurs évoluant sans cesse, il est aujourd'hui courant de posséder une station de travail dont les performances sont dignes des super-calculateurs des années 80. Les processeurs actuels possèdent des unités fonctionnelles multiples et donc un parallélisme interne au processeur, des pipelines super-scalaires et des hiérarchies mémoire importantes. De même que pour les processeurs, les stations de travail actuelles possèdent des caches internes dont la taille correspond à la mémoire vive des stations de travail du passé.

En ce qui concerne les architectures parallèles, il est possible de dégager, sans faire un historique complet, quelques grandes classes d'architectures. Après les super-calculateurs de style CRAY (mono-processeurs très puissants) et quelques architectures à mémoire partagée de taille modeste, les efforts de développement des constructeurs se sont dirigés vers les machines à mémoire distribuée. Ce fut l'époque du massivement parallèle avec les machines SIMD dont les limitations à des classes d'applications restreintes sont vite apparues.

L'évolution suivante a été celle des architectures MIMD à mémoire distribuée telle que la Paragon d'Intel. Cette machine était composée de nœuds constitués de deux processeurs i860, dont un spécialement dédié aux communications, et d'un PRMC (Paragon Mesh Routing Chip) reliant un nœud à ses quatre voisins par un lien bidirectionnel à 200 méga-octets par seconde.

La tendance actuelle évolue vers les grappes de machines SMP[1]. Un bon exemple de ce type d'architecture est la machine SP3 d'IBM qui est constituée de nœuds puissants comprenant 4 ou 16 processeurs (et même 32 pour les SP4), partageant une même mémoire reliée par un réseau rapide. Dans un même temps, les processeurs spécialisés développés par les constructeurs ont disparus au profit de composants dits « on the shelf » davantage utilisés dans des PCs et des stations de travail puissants que dans des machines parallèles. Les dernières versions de telles cartes peuvent abriter jusqu'à 8 processeurs pour atteindre des puissances allant jusqu'au Gigaflops.

Du point de vue des réseaux, l'amélioration de la technologie est allée de paire avec une baisse des prix. Les réseaux Myrinet, SCI, Fast Ethernet ou Giga Ethernet offrent ainsi de très bonnes performances pour des prix tout à fait compétitifs.

La combinaison de ces deux évolutions permet désormais à quiconque de construire sa propre machine parallèle à partir de composants du marché. Ces assemblages, nommés grappes, on d'abord été de taille modeste, mais il devient désormais possible d'agréger plusieurs centaines, voire plusieurs milliers de processeurs (jusqu'à 9000 Pentiums pour la machine du programme américain ASCI). Ainsi, au cours de cette thèse, nous avons effectué la majorité de nos expériences sur la grappe de PC *i-cluster* du laboratoire ID de l'IMAG, située à Grenoble. Cette grappe est com-

[1] Symmetric Multi-Processing

posée de 225 nœuds HP e-vectra connectés par un réseau Fast Ethernet. Chaque nœud possède un processeur Pentium III cadencé à 733 MHz, 256 Mo de mémoire et 15 Go d'espace disque. La grappe est organisée de la manière suivante : cinq commutateurs HP Procurve 4000 sont reliés par un réseau complet au Gigabit ; chacun de ces commutateurs permet la connexion de 45 machines sur un bus à 100 mégabits par secondes. La figure 1.1 présente l'architecture de cette grappe.

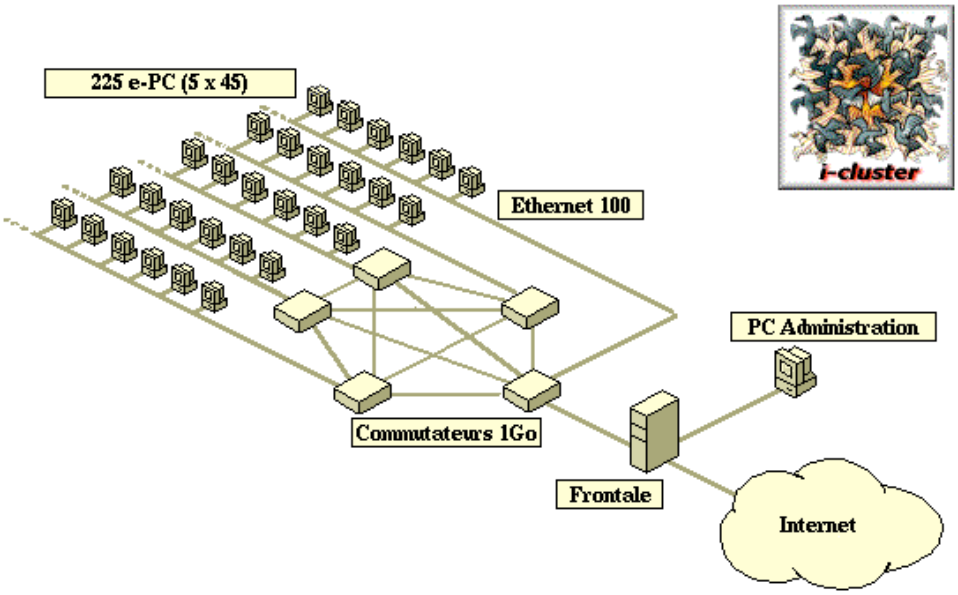


FIG. 1.1 – Architecture matérielle de la grappe *i-cluster* du laboratoire ID.

Une autre tendance consiste à tenter d'utiliser des machines disponibles dans le monde entier, et ce en conservant une relative transparence. Les plates-formes ainsi mises en place ont un potentiel immense car la plupart des machines utilisées de manière interactive sont le plus souvent sous-employées. De plus cela permet de déporter les calculs vers la machine la plus adaptée pour les résoudre. Malheureusement, de telles solutions sont, pour l'instant, 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 (agréger des machines accessibles sur la toile) et le logiciel (utiliser la puissance de machines inconnues comme on utiliserait l'électricité), le metacomputing [56] ou *grid computing* constitue certainement l'une des évolutions majeures de l'informatique.

Du point de vue de l'architecture matérielle, le metacomputing peut prendre plusieurs formes. Une caractéristique commune est cependant son importante hétérogénéité et sa hiérarchisation. L'idéal du metacomputing serait d'agréger des milliers de machines connectées à l'Internet pour résoudre des applications à grande échelle. Il s'agit d'ores et déjà d'une réalité puisque des projets tels que SETI@home [108] permettent d'effectuer des calculs sur des PCs de par le monde. Toutefois, ce type de résolution se limite, du fait de la faible bande passante du réseau, aux applications à parallélisme massif qui ne communiquent qu'épisodiquement avec une machine maître.

La grille GUSTO[2] constitue une autre vision des plates-formes de metacomputing. Cette grille interconnecte 330 super-calculateurs (soit plus de 3600 processeurs) répartis sur 17 sites – aux

[2]Globus Ubiquitous Supercomputing Testbed.

États-Unis, au Japon, en Australie et en Europe – pour fournir une puissance globale cumulée supérieure à 2 TeraFlops par seconde en crête. En France, le projet VTHD[3] du Réseau National de Recherche en Télécommunications (RNRT) vise à établir une plate-forme de metacomputing où les performances en communication sont beaucoup plus importantes que dans GUSTO. VTHD relie les centres de recherche de France Telecom, les Unités de Recherche INRIA et d'autres laboratoires par un réseau à 2,5 Gigabits par seconde. Sur une telle plate-forme, le réseau inter-centres s'avère même plus rapide que les réseaux internes des grappes qu'il relie.

Enfin, au niveau d'un laboratoire, il est également possible d'élaborer une plate-forme hétérogène de taille réduite en connectant plusieurs grappes entre elles par des réseaux plus ou moins rapides. Ainsi, nous avons utilisé durant cette thèse ce type de méta-ordinateur au sein du Laboratoire de l'Informatique du Parallélisme. Notre plate-forme hétérogène a été réalisée en connectant deux grappes de PC homogènes par un lien Fast Ethernet. La première grappes est composée de 12 nœuds Pentium Pro 200 MHz avec 64 Mo de mémoire connectés par un réseau Ethernet à 10 mégabits par secondes. La seconde, quant à elle, possède 7 nœuds Bi-Pentium II cadencés à 450 MHz, possédant 256 Mo de mémoire chacuns, et reliés par un réseau Fast Ethernet. La figure 1.2 présente l'architecture de notre plate-forme hétérogène.

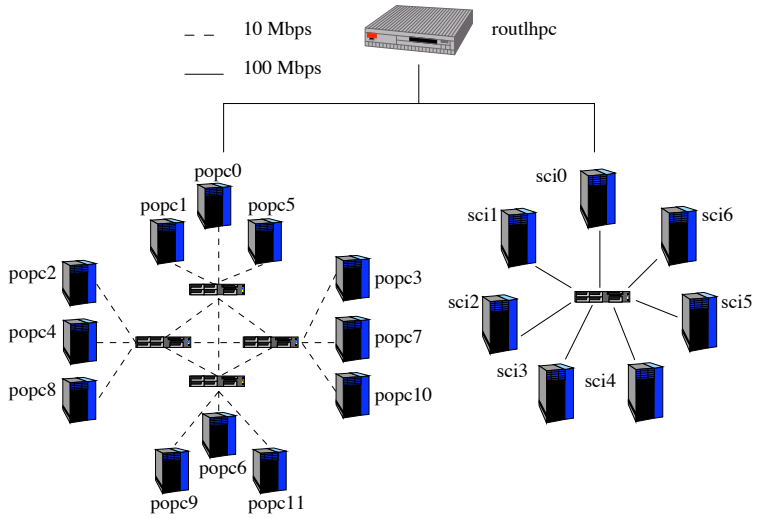


FIG. 1.2 – Architecture matérielle de la plate-forme hétérogène réalisée au sein du LIP.

Le problème commun à toutes ces visions du metacomputing est sans aucun doute l'hétérogénéité. Celle-ci apparaît à tous les niveaux, du matériel au logiciel en passant par le réseau et le système d'exploitation. Dans le meilleur des cas, des machines SMP de même type sont connectées par un réseau rapide, et dans le pire, c'est l'Internet dans son ensemble qui est utilisé comme machine parallèle. Un des challenges principaux du metacomputing est donc de fournir aux utilisateurs une vision « transparente » de la puissance de calcul disponible. Une des solutions consiste à développer des bibliothèques, de calcul et de communication, dont les interfaces sont standardisées.

[3] Réseau à Vraiment Très Haut Débit.

1.3 Bibliothèques de communication et d'algèbre linéaire

Malgré diverses tentatives de développement de compilateurs paralléliseurs efficaces, liés à des langages tels que HPF (*High Performance Fortran*) [72] ou OpenMP [90], le passage de messages reste le paradigme de programmation le plus adapté pour le développement d'applications ciblant des machines parallèles. Des bibliothèques telles que PVM [58], puis l'interface standard MPI [109], ont, de plus, propulsé cette technique au premier rang. Cependant, la programmation par passage de messages est également resté une affaire de spécialistes du fait de sa complexité et de difficultés concernant la recherche d'erreurs.

Dans le but de masquer au mieux cette complexité à l'utilisateur non-spécialiste, nombre de bibliothèques numériques ont été développées. Leur intérêt est triple. Tout d'abord, elles offrent une interface simple pour la résolution de divers problèmes courants 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 super-scalaires, unités parallèles, etc.) et le logiciel (utilisation de bibliothèques de plus bas niveau, options de compilation, choix du langage d'implantation, etc.). Enfin, et comme nous le verrons par la suite, ces bibliothèques peuvent être utilisées pour développer des environnements de plus haut niveau (versions parallèles des bibliothèques, logiciels mathématiques, support d'exécution pour des langages, serveurs de calculs, etc.).

Au cours de cette thèse, nous nous sommes plus particulièrement intéressés aux bibliothèques d'algèbre linéaire dense. Dans la plupart des algorithmes de ces bibliothèques, le nombre d'opérations flottantes est proportionnel au cube de la taille des données en entrée. L'écriture de bibliothèques parallèles efficaces doit donc tout d'abord passer par l'optimisation des noyaux séquentiels d'algèbre linéaire, notamment ceux des BLAS [46, 47, 73] et de LAPACK [51]. De plus, lorsque la taille des données est trop importante pour permettre un stockage en mémoire principale des algorithmes spécifiques doivent être mis en œuvre pour tenir compte du stockage de ces données sur supports de mémoire secondaire, typiquement le disque dur. On parlera alors de calcul *out-of-core* [15].

L'optimisation principale des noyaux séquentiels est basée sur l'utilisation d'algorithmes par blocs qui permettent une meilleure utilisation des hiérarchies mémoire. En effet, les performances obtenues sur un ordinateur moderne par les routines matrice–matrice, qui correspondent aux BLAS de niveau 3, sont bien meilleures que celles obtenues par des routines de niveaux inférieurs : opérations scalaire–vecteur et vecteur–matrice [48]. Afin d'obtenir les meilleures performances d'une machine donnée, une approche intéressante consiste à utiliser des générateurs de bibliothèques. L'idée de base est relativement simple mais cependant 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, les projets ATLAS [49] de l'Université du Tennessee, Knoxville et PhiPac [10] de l'Université de Berkeley développent des outils qui analysent de manière extensive l'architecture cible (niveaux et tailles des caches, opérations duales) et son compilateur (effets des options de compilation) et génèrent ensuite le code source de la bibliothèque. Il suffit alors de compiler la bibliothèque résultante pour obtenir une librairie que l'on pourra lier à d'autres applications. Ces outils permettent d'obtenir des performances proches des performances de crêtes et surtout équivalentes à celles des bibliothèques classiques en assembleur.

SCALAPACK est l'une des bibliothèques parallèles d'algèbre linéaire dense les plus connues. Cette bibliothèque est en fait la version parallèle d'un sous-ensemble des routines de la bi-

bliothèque LAPACK. ScaLAPACK se base principalement sur les noyaux séquentiels BLAS et sur leur version parallèle, les PBLAS [29]. Cette bibliothèque possède également sa propre bibliothèque de communication, les BLACS (*Basic Linear Algebra Communication Subroutines*.) [52] dont il existe plusieurs implantations au dessus de diverses bibliothèques de passage de messages, telles que PVM ou MPI. Un prototype d'extension de ScaLAPACK pour la gestion des données *out-of-core* est également disponible [35]. La figure 1.3 donne l'architecture logicielle de cette bibliothèque.

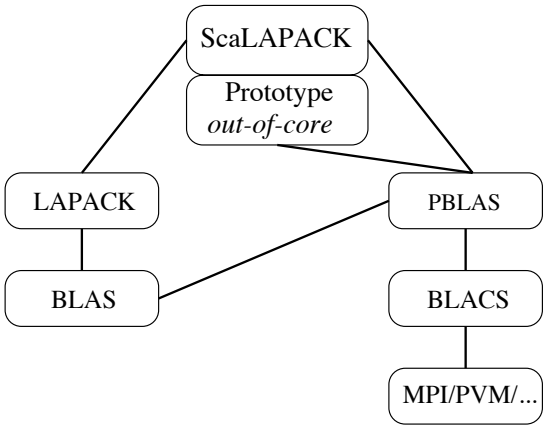


FIG. 1.3 – Architecture logicielle de ScaLAPACK.

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.

1.4 Scilab$_{//}$

Dans le cadre de l'Action de Recherche Coopérative OURAGAN [16] de l'INRIA, nous avons cherché à développer une version parallèle de SCILAB [59] en conservant l'interactivité de l'outil d'origine. SCILAB est un logiciel développé au sein du projet Métalau de L'INRIA Rocquencourt. SCILAB dispose d'une syntaxe simple qui permet de développer tout en restant très proche de la formulation mathématique du problème à résoudre, puisque basée sur des opérateurs matriciels de haut niveau. SCILAB inclut plus d'une centaine de fonctions mathématiques ainsi que des bibliothèques d'algèbre linéaire, de traitement du signal, d'analyse et optimisation de réseau, d'optimisation de systèmes linéaires, etc. Il est également possible d'interfacer facilement SCILAB avec des programmes écrits dans d'autres langages tels que C, Fortran ou issus d'autres programmes tels que Maple ou Mupad. Enfin, SCILAB est un logiciel du domaine public, contrairement à Matlab.

Comme nous l'avons indiqué précédemment, les environnements de type Matlab ou SCILAB sont limités par les capacités mémoire des machines sur lesquelles ils s'exécutent. Un utilisateur doit donc effectuer un compromis entre facilité de développement et performances. Dans la plupart des cas, la solution retenue est de diviser le développement d'une application en deux phases : un prototypage réalisé au moyen d'outils de haut niveau et validé sur des problèmes de petites tailles, d'une part et une phase de réécriture de l'application dans un langage compilé, d'autre part. Cette réécriture vise le plus souvent des plates-formes parallèles offrant de plus grandes capacités de mémoire et de calcul que de simples stations de travail. Le problème de cette solution est qu'elle impose à l'utilisateur d'acquérir de solides connaissances en programmation parallèle, ou de sous-traiter la parallélisation de l'application.

Pour ne plus être confronté à un développement en deux phases, l'idée de paralléliser des outils tels que Matlab et SCILAB est apparue. Cependant, en 1995, Cleve Moler, alors président de Mathworks, donne les raisons de l'inexistence d'un Matlab parallèle [81]. Selon lui, il y a trois raisons majeures pour ne pas développer de versions parallèles de Matlab. Tout d'abord, le modèle mémoire des machines parallèles, *i.e.*, la mémoire est distribuée, conduit à une multitude de transfert des opérandes entre la station où s'exécute Matlab et la machine parallèle. Or, les mouvements de données sont souvent plus coûteux que les opérations de calcul. La seconde raison est le grain de calcul de Matlab. Cleve Moler affirme que la plus grande partie des temps d'exécution d'applications Matlab est passée dans des portions de codes qui ne peuvent pas être parallélisées, telles que l'analyseur de code, l'interpréteur, ou les routines graphiques. Enfin, l'auteur soulève un argument économique supposant le manque d'utilisateurs de Matlab possédant une machine parallèle.

En dépit de cette note, plusieurs projets ont été démarrés dès le début des années 90 afin d'obtenir des environnement de type Matlab à hautes performances. Même si la distribution actuelle de Matlab inclue la bibliothèque d'algèbre linéaire LAPACK [51], les performances sont toujours limitées pour des calculs de haut niveau. Nombre d'arguments peuvent être énoncés pour contrer ceux de Moler, mais le meilleur d'entre eux reste le développement de ces projets. Deux types d'approches existent : soit compiler des codes Matlab – avec ou sans extensions parallèles – pour les exécuter ensuite sur une machine parallèle ; soit utiliser le parallélisme directement depuis la console Matlab à travers des extensions du langage ou une surcharge des opérateurs.

Dans la suite de ce chapitre, après avoir brièvement présenté des versions compilant des codes Matlab, nous allons détailler les deux approches que nous avons suivies. La figure 1.4 présente ces deux approches. La première consiste à répliquer des processus SCILAB sur chacun des nœuds de la machine parallèle. Ces processus communiquent ensuite grâce aux bibliothèques de communication PVM [58] ou MPI [109] (Fig. 1.4-A). Le but du projet SCILAB$_{//}$ étant d'obtenir des hautes performances de SCILAB dans le cadre d'applications numériques, des interfaces à la bibliothèque d'algèbre linéaire SCALAPACK [12] et son prototype *out-of-core* [35] ont également été développées (Fig. 1.4-B). La seconde approche est basée sur des serveurs de calcul. Nous détaillerons l'interface entre le SCILAB et NETSOLVE (Fig. 1.4-C). Nous conclurons alors ce chapitre en expliquant comment les développements menés dans le cadre d'OURAGAN peuvent aboutir à l'élaboration d'un environnement conforme au modèle ASP.

1.4.1 Compilation–parallélisation de scripts Matlab

Le tableau 1.1 recense les différents projets visant à obtenir un Matlab parallèle à hautes performances en suivant l'approche « compilation ». Des scripts Matlab sont compilés vers un autre langage (typiquement Fortran [36, 102] ou C [53, 61, 76, 80, 107]). Des optimisations issues des

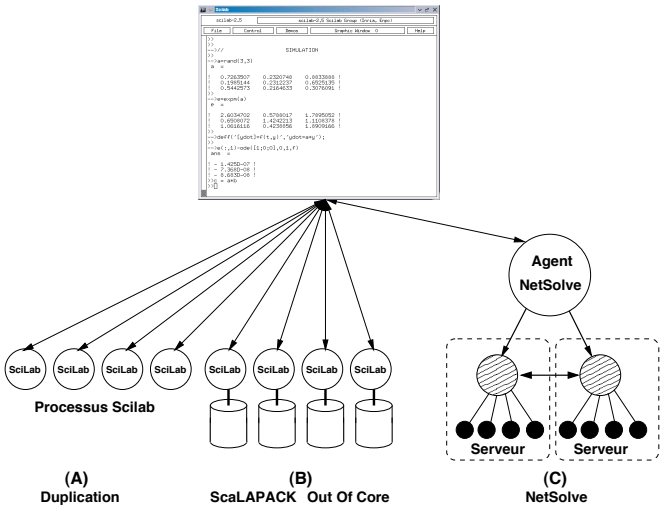


FIG. 1.4 – Architecture générale de SCILAB $_{//}$.

domaines de la parallélisation automatique et de la vectorisation sont alors appliquées aux codes générés [36, 61, 80, 107]. Des appels à des bibliothèques numériques à haute performance peuvent également utilisés dans ces codes [28, 53, 76, 102].

Nom	Langage généré	Bibliothèques	Référence
Menhir	FORTRAN, C	LAPACK et SCALAPACK	[28]
FALCON	FORTRAN 90 et pC++	-	[36]
CONLAB	C	PICL (comm.), BLAS, LAPACK, Motif	[53]
MATCH	C, RTL, VHDL	spécifique	[61]
Otter	C	MPI (comm.), SCALAPACK, FFT parallèle	[76]
ParAL	C	PVM (comm.), spécifique	[80]
Paradigm	FORTRAN	SCALAPACK	[102]
RTExpress	C	MPI (comm.)	[107]

TAB. 1.1 – Projets de compilation–parallélisation de scripts Matlab.

Cette approche permet d'obtenir de bonnes performances du fait de l'utilisation de bibliothèques efficaces et de techniques de compilation sophistiquées. En revanche, la compilation de scripts ne conserve pas l'interactivité d'un outil tel que Matlab. Toute modification du programme impliquera donc une recompilation. De plus, des problèmes compliqués tels que l'inférence de type et la gestion de tailles de données dynamiques apparaissent. Enfin, si l'application utilise des fonctionnalités propres à Matlab, le compilateur devra déterminer quelle bibliothèque parallèle sera à même de réaliser ces opérations. L'ensemble de ces raisons ont fait que l'approche « compilation » n'a pas été retenue pour le développement de SCILAB $_{//}$.

1.4.2 Parallélisation par duplication de processus

Cette seconde approche dans la parallélisation d'outils de type Matlab conserve l'interactivité du logiciel séquentiel. Le principe de base consiste à répliquer l'outil, intégralement ou seulement une partie, sur chacun des nœuds de la machine parallèle cible. L'avantage de cette approche est que le processus maître, situé sur la station de travail de l'utilisateur, n'envoie que des commandes classiques aux processus de travail qui les interpréterons dès leur réception. Dans la plupart des logiciels développés selon ce principe, le langage de base est étendu par des primitives de communications, elles-mêmes basées sur des bibliothèques standards telles que PVM [64, 95] ou MPI [96]. Le tableau 1.2 résume les différents projets qui parallélisent Matlab en dupliquant des processus.

Nom	Bibliothèque de communication	Référence
Parallel Toolbox	PVM	[64]
Paramat	-	[94]
DP Toolbox	PVM	[95]
PMI	MPI	[96]
Multimatlab	-	[115]

TAB. 1.2 – Projets utilisant la duplication de processus Matlab.

Dans le cadre du développement de SCILAB//, nous avons souhaité ajouter dans SCILAB la capacité d'exécuter des commandes standards sur des machines distantes. Pour cela, des processus SCILAB « classiques » sont lancés sur ces machines. Afin de pouvoir faire communiquer ces processus, une interface à la bibliothèque de passage de messages PVM a été développée par Éric Fleury et Frédéric Desprez [39].

1.4.2.1 Interfaces pour le passage de messages

L'interface entre SCILAB et PVM permet à un utilisateur de développer des applications parallèles tout en bénéficiant de l'ensemble des fonctionnalités de SCILAB. La bibliothèque PVM a été retenue parce qu'il est possible, avec cet environnement, d'ajouter dynamiquement de nouveaux processus après le démarrage du programme. Cette fonctionnalité n'est en revanche pas disponible dans MPI. Une interface entre SCILAB et MPI a néanmoins été développée, les implantations de ce standard étant généralement plus performantes. Mais dans ce cas, l'utilisateur sera contraint de décider du nombre maximum de processus qu'il souhaite démarrer au cours de sa session SCILAB//.

Afin d'obtenir des performances les plus proches possible de celles de la bibliothèque d'origine, ces interfaces ne donnent accès qu'à des fonctions de bas niveau. Leur utilisation est par conséquent plutôt réservée à des utilisateurs familiers de la programmation par passage de messages et de ces bibliothèques. Plusieurs instances de SCILAB peuvent ainsi communiquer et interagir, et l'utilisateur peut envoyer et recevoir tous types de données (*e.g.*, matrices, listes, fonctions, etc.) en utilisant des appels à des fonctions PVM ou MPI.

Pour l'envoi de matrices complètes, les performances de ces interfaces sont équivalentes à celles obtenues en utilisant les mêmes bibliothèques de communication dans un code C ou Fortran et le surcoût d'interprétation est négligeable [38]. En revanche, lorsque l'on souhaite envoyer des

sous-matrices, *e.g.*, $A(1:2:100, 2:2:100)$, ce qui correspond à une sous-matrice de taille 50×50, la recopie des données dans un tampon induit un surcoût naturel.

Ces interfaces pour le passage de messages permettent désormais d'écrire des programmes parallèles sans perdre la simplicité de la syntaxe et les fonctionnalités de SCILAB. Mais comme nous l'avons indiqué précédemment, l'utilisation de ces interfaces reste une affaire d'experts. L'objectif suivant du développement de SCILAB$_{//}$ a été d'offrir un niveau d'abstraction supplémentaire en réalisant des interfaces entre SCILAB et des bibliothèques d'algèbre linéaire telles que les PBLAS, SCALAPACK, son prototype *out-of-core* et sa bibliothèque de communication associée.

1.4.2.2 Interfaces avec des bibliothèques parallèles d'algèbre linéaire

Grâce à ces interfaces, un utilisateur de SCILAB$_{//}$ peut distribuer des matrices sur l'ensemble des processus SCILAB démarrés et exécuter des routines parallèles sur la grille virtuelle ainsi créée. Néanmoins, cette méthode nécessite toujours une certaine compétence du point de vue de l'utilisation des bibliothèques interfacées ainsi que de l'algèbre linéaire parallèle, puisque l'utilisateur est responsable de la distribution initiale des données et des éventuelles redistributions. En revanche, sa tâche est simplifiée puisque, si l'API utilisée dans SCILAB$_{//}$ reste très proche de celles des bibliothèques interfacées, certains paramètres peuvent être « omis » et remplacés par des valeurs par défaut, *e.g.*, transposition des matrices, indices de ligne et/ou de colonne, etc. De plus, le choix de la routine à exécuter est adapté en fonction du type des données, *e.g.*, réels simple ou double précision, complexes, etc.

Comme nous l'avons évoqué précédemment, l'objectif de ces interfaces est d'offrir un niveau d'abstraction supplémentaire. En effet, la plupart des utilisateurs de SCILAB sont des numériciens qui souhaitent obtenir de bonnes performances pour des tailles de problèmes raisonnables tout en exprimant leurs équations dans une syntaxe la plus proche possible du langage mathématique. Or, ce type d'utilisateur n'est pas nécessairement spécialiste du parallélisme. Afin de leur masquer la gestion des échanges de messages et les appels de bas niveau aux bibliothèques d'algèbre linéaire, un nouveau type distribué a été ajouté dans SCILAB$_{//}$. Du point de vue de l'utilisateur, le fait qu'une matrice soit distribuée ou non ne modifie que très peu la manière d'écrire des programmes SCILAB. En effet, seules trois commandes ont été ajoutées qui permettent respectivement d'initialiser la grille de processeurs, de spécifier une distribution et de distribuer effectivement une matrice sur les processus de travail SCILAB. Les deux premières routines acceptent des valeurs par défaut qui peuvent éventuellement être stockées dans un fichier.

Puisque grâce à ces nouvelles fonctions, SCILAB$_{//}$ peut désormais gérer aussi bien des matrices locales que distribuées, les fonctions et opérateurs les plus courants de SCILAB ont été surchargés pour gérer le type distribué. Ainsi, les opérations effectuées sur des matrices distribuées seront réalisées en parallèle. Pour l'instant, toutes les opérations qui possèdent leur équivalent dans les deux bibliothèques parallèles PBLAS et SCALAPACK ont été surchargées. Il est important de noter que l'utilisateur peut toujours utiliser les notations matricielles « classiques » de SCILAB pour écrire des programmes parallèles.

Bien entendu, toutes les fonctions SCILAB n'ont pas été surchargées. Lorsqu'une opération est effectuée sur une donnée distribuée et qu'il n'y a pas de fonction parallèle correspondante, l'utilisateur peut choisir entre plusieurs modes : le premier – et le plus simple – consiste à générer une erreur ; le second ramène systématiquement la matrice distribuée dans la console SCILAB à condition que celle-ci puisse être stockée en mémoire, et exécute ensuite 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 locale. 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 données.

Le tableau 1.3 liste les fonctions SCILAB qui supportent la surcharge d'opérateur pour les matrices distribuées. La récupération et la modification de sections de tableaux fonctionnent également de manière transparente sur les matrices distribuées.

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

TAB. 1.3 – Opérations surchargées pour les matrices distribuées dans SCILAB $_{//}$.

La figure 1.5 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, respectivement sur un, quatre et seize processeurs d'une SGI Origin 2000, deux multiplications de matrices `Res=A*B*C`, où A, B et C sont des matrices carrées. Nous pouvons constater qu'à partir de matrices de taille 200×200, il devient plus intéressant d'utiliser un produit de matrices parallèle. En effet, le léger surcoût de traitement induit par SCILAB n'est plus un problème lorsque la dimension du problème devient assez importante.

Une première définition de l'interface entre SCILAB et le prototype *out-of-core* de SCALAPACK a été donnée dans [19]. Celle-ci est basée sur une version améliorée du prototype *out-of-core*, développée au LaRIA à Amiens qui se compose de quelques routines telles qu'une version optimisée de la factorisation LU, un générateur de matrice identité *out-of-core*, un comparateur de matrices *out-of-core* ainsi que d'une routine d'inversion de matrices *out-of-core* [20, 23].

Si de bonnes performances peuvent être obtenues pour des opérations globales telles qu'une factorisation, le surcoût introduit par les entrées/sorties ne peut être masqué pour des opérations SCILAB où le calcul est effectué élément par élément. Ainsi, dans l'expression $A = \sin(A) + \cos(B) + sqrt(A)$, où A et B sont des matrices *out-of-core*, la matrice A est lue deux fois durant l'évaluation. De plus, des données temporaires de grande taille sont générées puis lues. Tout cela induit une importante consommation d'espace disque et rend le coût des entrées/sorties supérieur au coût de calcul. Une solution pour diminuer ces coûts d'entrées/sorties, tout en supprimant la génération de variables temporaires, consisterait à développer des algorithmes adaptés dans

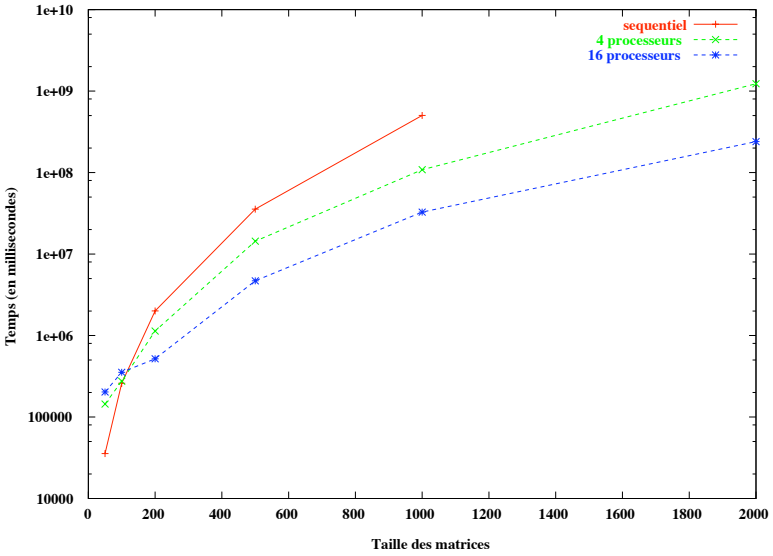


FIG. 1.5 – Performances du produit de matrices en utilisant la surcharge d'opérateur distribué $*$ dans SCILAB$_{//}$.

lesquels les matrices A et B serait découpées en blocs de manière à ce que toute l'expression puisse être évaluée en mémoire bloc après bloc. Hélas, cette gestion optimisée des données *out-of-core* impliquerait une réécriture quasi totale de l'interpréteur SCILAB et n'a donc pas été inclue dans les travaux de l'ARC OURAGAN.

1.4.3 Approche serveurs de calcul

Afin de fournir une interface la plus « transparente » possible à l'utilisateur, une seconde approche a été étudié dans le cadre du développement de SCILAB$_{//}$. Celle-ci consiste à utiliser le modèle RPC[4] [78, 79] en ciblant des serveurs de bibliothèques numériques. Le tableau 1.4 donne la liste des projets qui parallélisent Matlab au moyen de serveurs de calcul. Matpar [110] est un projet du Jet Propulsion Laboratory de la NASA qui propose un ensemble de nouvelles fonctions à inclure au langage Matlab pour pouvoir lancer des calculs sur une machine parallèle de type HP Convex SPP2000 ou sur une grappe Beowulf. Ces fonctions sont écrites sous forme de scripts compilés. Les commandes et les données sont transférées de la console vers la machine parallèle en utilisant la bibliothèque de communication PVM. PPserver [66] est à la base un projet d'étudiants développé au MIT. Le parallélisme est introduit par la surcharge de certaines fonctions ayant leurs équivalents dans les bibliothèques disponibles et par l'ajout d'un nouvel opérateur $*p$. Cet opérateur permet à l'utilisateur de signifier qu'une matrice doit être distribuée selon une certaine dimension. Le projet PSI (PLAPACK Server Interface [82]) défini également un ensemble de routines C appelables depuis Matlab qui permettent de communiquer avec une machine parallèle via MPI. Un serveur PLAPACK, un concurrent de SCALAPACK, tourne sur la machine parallèle dans l'attente de commandes provenant de la console. Enfin, NETSOLVE [24, 87] est un environnement de type NES[5] à part entière qui dispose d'une interface Matlab en plus de ses autres interfaces (C,

[4] Remote Procedure Call
[5] Network Enabled Servers

Fortran et Mathematica). NETSOLVE permet de démarrer, au moyen de requêtes bloquantes ou non-bloquantes, des calculs à distance sur une plate-forme de metacomputing. Il utilise le modèle client–agent–serveurs. Une politique simple d'équilibrage des charges 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.

Nom	Bibliothèques de calcul	Référence
NETSOLVE	LAPACK, SCALAPACK, PETSc	[24]
PPServer	SCALAPACK, S3L, PARPACK, PETSc	[66]
PSI	PLAPACK	[82]
MatPar	BLAS, PBLAS, SCALAPACK	[110]

TAB. 1.4 – Projets utilisant des serveurs de calcul.

Dans le cadre d'OURAGAN, une interface entre SCILAB et NETSOLVE a été développée par Éric Fleury et Emmanuel Jeannot. Ce développement, basé sur l'interface Matlab de NETSOLVE, permet à un utilisateur de soumettre des problèmes à NETSOLVE depuis la console.

Cette approche RPC utilisée par NETSOLVE permet de résoudre des problèmes de très grande taille via Internet. D'autres environnements qui emploient ce paradigme existent. Nous pouvons notamment citer NINF [85, 89], NEOS [34, 86], RCS [3], GridRPC [84] ou même OVM [13] pour une utilisation sur des grappes. Le nombre et l'activité de ces projets montrent l'intérêt porté par la communauté du metacomputing à ce type d'environnements. C'est dans cette optique qu'en 2000, à la fin de l'ARC OURAGAN, nous avons orienté nos recherches sur le développement d'un tel environnement nommé DIET (*Distributed Interactive Engineering Toolbox*) [44]. Ces travaux se sont tout d'abord basés sur NETSOLVE, mais nous nous sommes rapidement aperçus que des améliorations étaient possibles.

De plus, il nous a semblé nécessaire de développer un ensemble d'outils utiles au développement d'applications conformes au modèle ASP, présenté au début de ce chapitre. Ce modèle permet en effet, plutôt que de confier la gestion de tous les services à un prestataire externe, de ne fournir que des offres spécifiques dépendant fortement du prix payé par le client. Le bénéfice majeur de telles applications, outre la simplicité d'utilisation, est sans nul doute la multiplication de l'offre, grâce à Internet, combinée à une disponibilité quasi continue. Les fournisseurs de services peuvent quant à eux garder un total contrôle de leur code, ce qui améliore la maintenance et l'évolutivité. Enfin, le fait que les codes applicatifs demeurent au sein des entreprises ou laboratoires de recherche qui les développent limite les risques de piratage ou d'usage abusif. La réalisation d'environnements conformes à ce modèle associe différents niveaux de compétences et de fonctionnalités allant de la gestion des services réseaux et des serveurs de calculs aux interfaces client en passant par l'infrastructure en elle-même qui est en charge d'effectuer une transition efficace et transparente entre les deux composants précédents. Nous présenterons plus en détail tous les développements liés à un tel environnement dans le chapitre 4.

Chapitre 2

Modélisation de routines parallèles et prédiction de performances

Plus de détails lundi, tant il est vrai que les prédictions sont difficiles, surtout quand elles concernent l'avenir.
Pierre Eckert, collaborateur de Météo Suisse.

2.1 Introduction

Dans le cadre du metacomputing et plus particulièrement dans celui des ASP, que nous avons décrit dans le chapitre précédent, l'objectif est de déterminer quel serveur est le plus adapté à la résolution d'un problème. Un des critères importants dans cette détermination du « meilleur » serveur est la connaissance des performances relatives des différentes machines, en fonction du problème soumis et des données transmises. Cette connaissance s'exprime sous la forme d'informations statiques concernant les besoins en termes d'espace mémoire et de puissance de calcul d'une routine donnée, ainsi que les performances théoriques d'un réseau d'interconnexion. Dans le cas de routines parallèles, les informations statiques nécessaires portent également sur la décomposition en appels à des routines séquentielles, sur le schéma de communication inhérent à la routine, ainsi que sur l'architecture de la machine parallèle utilisée.

À ces informations statiques viennent s'ajouter des informations dynamiques relatives au moment où est exécutée la routine à évaluer. En effet, dans le cas d'une plate-forme d'exécution non dédiée, il est indispensable de tenir compte de la charge processeur et de la disponibilité réelle du réseau au moment de l'exécution.

Plusieurs critères de performance permettent de déterminer quelle solution adopter parmi plusieurs possibles. Une « meilleure » solution peut, en effet, être celle ayant le plus petit temps

d'exécution, celle minimisant les communications, où encore celle utilisant le moins de processeurs tout en obtenant un temps d'exécution acceptable. Le critère de choix et la notion d'acceptabilité peuvent être définis par l'utilisateur, mais le processus d'acquisition des différents temps de calcul et communication correspondant à chacune des solutions doit rester transparent pour cet utilisateur. Ces temps de calcul et de communication doivent non seulement tenir compte des capacités théoriques de la plate-forme d'exécution, mais aussi des disponibilités réelles au moment de la résolution de problème. Enfin, le temps de réponse doit être le plus faible possible. En effet, la somme du temps d'estimation, de la déportation du calcul sur le serveur choisi et du transfert des données et du résultat doit être inférieure au temps d'exécution local du calcul de l'utilisateur.

2.2 Exemple motivant

Supposons, par exemple, que l'on souhaite résoudre un problème impliquant deux données A et B. Nous disposons pour cela de trois serveurs de puissances différentes. La donnée A est distribuée sur le premier serveur, S_1, tandis que la donnée B est distribuée sur le second serveur, S_2. La figure 2.1 résume la configuration dans laquelle nous nous plaçons. Sur cette figure, nous trouvons également un quatrième serveur, S_4, qui est en réalité l'agrégation de S_1 et S_2. En effet, il est possible que plusieurs serveurs de calcul s'exécutent concurremment sur plusieurs sous-ensembles d'une même machine parallèle et puissent être agrégés pour obtenir plus de puissance tout en conservant l'homogénéité du serveur, en termes de processeurs et de réseau.

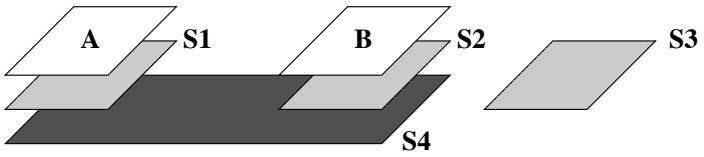


FIG. 2.1 – Exemple de configuration dans le cadre d'une application à base de serveurs de calcul.

Dans une configuration de ce type, et en supposant que les données doivent être alignées pour pouvoir exécuter le calcul, plusieurs solutions s'offrent à nous. En effet, il est possible de :

1. Redistribuer B sur S_1, et d'y effectuer le calcul ;
2. Redistribuer A sur S_2, et d'y effectuer le calcul ;
3. Redistribuer A et B sur S_3, et d'y effectuer le calcul ;
4. Redistribuer A et B sur S_4, et d'y effectuer le calcul.

Dans ce chapitre, nous allons présenter la bibliothèque FAST [40, 98, 99], développée par Martin Quinson au Laboratoire de l'Informatique du parallélisme, qui constitue un outil de prédiction dynamique de performances dans un environnement de metacomputing. Cet outil n'étant, dans sa version actuelle, capable de ne traiter que des routines séquentielles, une extension pour la gestion des routines parallèles a été développée au cours de cette thèse [21, 22].

L'objectif de FAST est de fournir à une application cliente, typiquement un ordonnanceur, des informations pertinentes et précises sur les différentes composantes de la plate-forme d'exécution. Ce type d'informations permettra de répondre aux problèmes de choix soulevés par l'exemple que nous venons de présenter. FAST est ainsi capable de prédire le temps d'exécution et l'espace mémoire nécessaires aux tâches à ordonnancer sur la plate-forme d'exécution grâce à une modélisation des besoins des routines. Des outils de surveillance adéquats permettent en outre

de mesurer les disponibilités dynamiques des différentes ressources de calcul et de communication. FAST est également capable de mettre ces valeurs en correspondance pour prédire le temps nécessaire à l'exécution d'une tâche donnée sur une machine donnée à un instant donné.

2.3 FAST : Fast Agent's System Timer

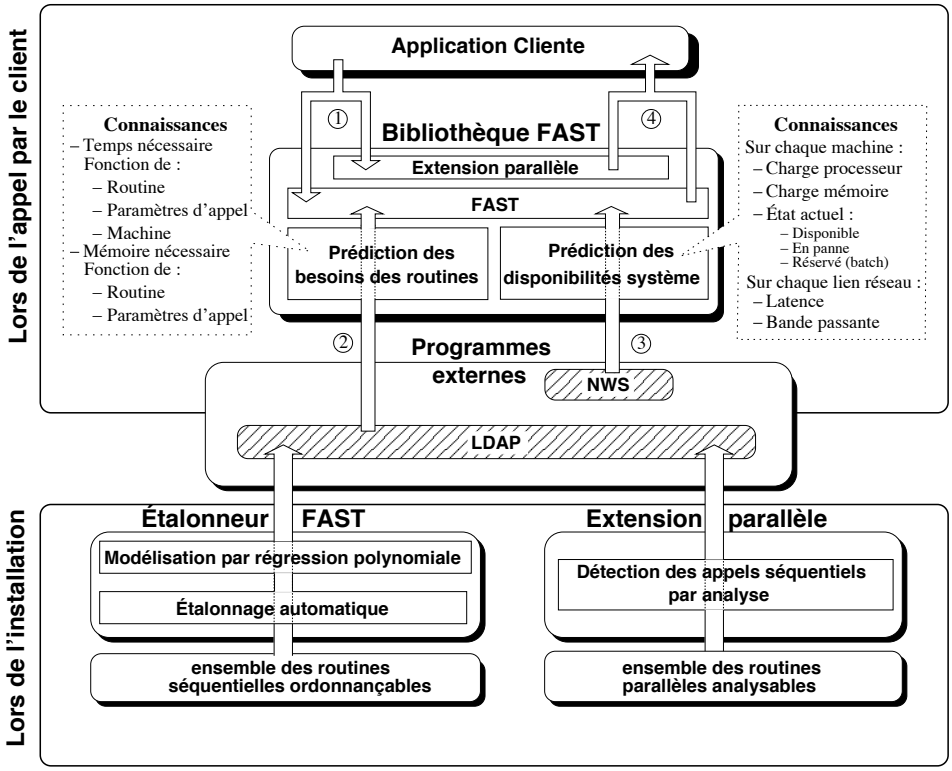


FIG. 2.2 – Architecture de FAST.

Comme nous pouvons le voir sur la figure 2.2, FAST se compose de plusieurs modules. Le premier, lancé lors de l'installation de FAST sur un serveur de calcul, est un outil chargé de modéliser les routines séquentielles exécutables sur ce serveur, d'une part, et d'analyser les routines parallèles pour détecter les appels aux routines séquentielles contenus dans ces routines, d'autre part. Le module intermédiaire se compose des différents outils externes utilisés par FAST. Nous avons enfin la bibliothèque à proprement parler, appelée à chaque estimation, et ses différents composants. Nous allons à présent détailler le contenu et le fonctionnement de chacun de ces modules ainsi que leurs interactions. Pour chacun d'entre eux, nous justifierons les choix effectués en les comparant aux autres systèmes ou méthodes existants.

2.3.1 Acquisition de données dynamiques et prédiction de performances

Plusieurs projets permettent de surveiller une plate-forme d'exécution distribuée et d'extraire de cette surveillance des informations susceptibles d'être utilisées par un ordonnanceur [1, 69]. Mais ces informations sont souvent de trop bas niveau pour être utilisées telles quelles.

NWS (Network Weather Service) [120] est un projet développé par l'équipe du Professeur Wolski à l'Université de Californie, Santa Barbara. Il s'agit d'un système distribué de sondes logicielles et d'estimateurs statistiques permettant de centraliser l'état actuel de la plate-forme sur laquelle il s'exécute, ainsi que d'en prédire les évolutions. Il est ainsi possible de surveiller la latence et la bande passante de n'importe quel lien de communication TCP, la charge processeur, la mémoire disponible ou encore l'espace libre sur les disques. En ce qui concerne le processeur, NWS n'est pas seulement capable de décrire la quantité utilisée, mais également la quote-part dont disposerait un nouveau processus [121]. [120] et [99] donnent une présentation plus détaillée du fonctionnement de NWS.

NWS permettant non seulement d'acquérir des informations dynamiques sur une plate-forme de metacomputing mais également de les prédire, ce système a été retenu pour constituer la base du développement de FAST. Ce choix n'est cependant nullement limitatif, et il est parfaitement possible d'utiliser une autre bibliothèque offrant le même type de fonctionnalités.

En plus de NWS, FAST utilise un autre logiciel : LDAP (Lightweight Directory Access Protocol) [65], un système d'annuaire de données distribué et hiérarchique largement répandu dans la communauté du metacomputing. LDAP est explicitement optimisé pour les opérations de lecture et de recherche, au détriment des opérations d'écriture. Ce système n'est donc pas prévu pour stocker des données fortement dynamiques, mais convient en revanche parfaitement pour l'usage qu'il en fait dans FAST, à savoir stocker des données statiques telles que le modèle de coût d'une routine séquentielle ou parallèle pour une machine donnée.

2.3.2 Détermination des disponibilités du système

L'objectif du module chargé de la détermination des disponibilités du système est de découvrir la charge induite par l'utilisation des ressources afin de tenir compte du caractère partagé de la plate-forme de metacomputing, en se basant sur NWS. Dans sa version actuelle, FAST est capable de surveiller la charge processeur et la mémoire disponible sur chaque hôte, ainsi que la latence et la bande passante de chaque lien TCP. La surveillance de nouvelles ressources, telles que l'espace disque ou les performances de liens non-TCP, peut aisément être intégrée à FAST.

L'intégration faite par Martin Quinson de NWS dans FAST lui a permis d'apporter une solution à l'une des faiblesses de NWS. En effet, un administrateur peut spécifier les tests réseaux à mener, afin de ne faire que les tests nécessaires et de ne pas saturer le réseau inutilement, mais NWS ne permet pas de combiner ensuite les résultats de ces tests automatiquement. Si l'on dispose, par exemple, de trois machines A, B et C, et que l'on a demandé des tests entre A et B d'une part, et B et C d'autre part, NWS est incapable de donner une estimation des capacités de communication entre A et C, car il n'y a pas de test direct entre ces machines. Dans ce cas, FAST agrège les résultats donnés par NWS pour offrir à l'application cliente une estimation des paramètres demandés. FAST effectue pour cela un parcours du graphe des tests directs pour trouver un chemin entre les machines concernées. La bande passante renvoyée est le minimum de celles rencontrées sur ce chemin tandis que la latence est la somme de celles retournées par les tests directs. Les prédictions ainsi réalisées risquent certes d'être d'une qualité moindre que celles basées sur des tests directs, mais constituent toutefois des informations intéressantes et non négligeables.

2.3.3 Estimation de routines

L'un des apports les plus fondamentaux du travail de Martin Quinson dans FAST est sa capacité à prédire les besoins théoriques en termes de temps d'exécution et d'espace mémoire d'une

routine de calcul. Ce problème est complexe, et selon le type de routine à évaluer, différentes approches sont possibles.

Certaines routines sont relativement régulières et facilement chronométrables. Il s'agit typiquement de routines séquentielles telles que celles de la bibliothèque BLAS [46]. L'approche classique pour estimer le temps d'exécution de telles routines est de réaliser une modélisation basée sur une étude manuelle ou automatique des codes sources. Ce type d'approche n'est pas toujours réalisable car il nécessite d'avoir accès au code source, ce qui n'est pas toujours possible. Par exemple, pour la plupart des machines, une implantation de la bibliothèque BLAS fortement optimisée est fournie par le constructeur ou peut être obtenue par génération automatique [49]. Même lorsque le code source est disponible, de très nombreux paramètres entrent en jeu lorsqu'il s'agit de prévoir les performances d'un code séquentiel sur une machine donnée. Il faut en effet tenir compte de l'implantation, mais également des optimisations réalisées par le compilateur ou le processeur, des caractéristiques du système d'exploitation ou encore du matériel comme la gestion de la mémoire et des caches. La prise en compte de l'ensemble de ces paramètres impose une étude approfondie de chaque couple {code ; matériel}, ce qui représenterait un travail colossal dans un environnement de metacomputing.

Pour pallier ce problème, il est relativement classique d'utiliser un étalonnage générique de la machine pour déterminer le nombre d'opérations élémentaires effectuées par unité de temps, puis de dénombrer le nombre de ces opérations pour chaque routine à évaluer. Cependant, cette approche ne tient pas compte des effets de cache dont l'influence est pourtant capitale en calcul intensif comme nous l'avons expliqué dans le chapitre précédent.

L'approche utilisée par FAST consiste à étalonner les performances d'une routine par une série de tests réalisée lors de l'installation de FAST sur un nouveau serveur de calcul. Les résultats obtenus lors de ces tests sont ensuite modélisés par régression polynômiale puis stockés dans un arbre LDAP. Cette phase de tests peut s'avérer être assez coûteuse en temps, mais elle n'intervient qu'une seule fois à l'installation de FAST. De plus, il est possible de ne tester qu'un seul élément d'un ensemble homogène de machines, et d'utiliser les valeurs obtenues pour toutes les autres machines de cet ensemble.

D'autres routines sont toutefois plus difficiles à chronométrer, et donc à étalonner, mais cependant plus simples à analyser. C'est par exemple le cas des routines parallèles régulières qui sont le plus souvent décomposables en une alternance de phases de calcul et de phases de communication. Les phases de calcul sont composées d'appels à une ou plusieurs routines séquentielles. Les phases de communication sont quant à elles constituées de communications point–à–point ou globales. Le chronométrage de telles routines est rendu d'autant plus difficile par la gestion de la distribution des données et le choix de la grille de processeurs utilisée. Il nous a donc semblé possible et intéressant de combiner analyse de code et informations fournies par FAST sur les temps d'exécutions séquentiels et les disponibilités réseau pour estimer les temps d'exécution de routines parallèles. La gestion de telles routines parallèles dans FAST sera détaillée un peu plus tard dans ce chapitre.

Il existe enfin une troisième classe de routines qu'il n'est pour l'instant pas possible d'estimer avec FAST. Il s'agit typiquement de routines impliquant des matrices creuses. En effet, dans le cas d'algorithmes itératifs, le temps d'exécution total dépend fortement des données. Nous ne pouvons donc pas inclure le modèle générique d'une routine portant sur des matrices creuses comme nous l'avons fait pour les routines denses de type SCALAPACK. Cependant, il est parfois possible d'utiliser la phase de prétraitement pour prédire le temps de résolution. Ainsi, dans [63], les auteurs utilisent une simulation basée sur la factorisation symbolique pour obtenir une estimation précise du temps de résolution.

2.3.4 Interface utilisateur

L'interface utilisateur de FAST comporte plusieurs catégories de fonctions. Certaines permettent d'obtenir le détail des prédictions sur les besoins des routines et les disponibilités du système. D'autres combinent automatiquement ces résultats pour produire des valeurs directement utilisables par l'application cliente. Cette interface a été conçue de façon à masquer les complexités sous-jacentes à l'application appelante. Nous ne présenterons ici que les fonctions utilisées pour le développement de l'extension pour la gestion des routines parallèles, l'interface utilisateur complète étant détaillée dans [99].

```
fast_comp_time(hôte, fonction, data_desc, &valeur)
```

permet d'obtenir le temps de calcul prévu sur la machine `hôte` pour `fonction` selon la valeur des paramètres contenus dans `data_desc` (dont le format sera détaillé à la fin de ce paragraphe) en tenant compte de la charge actuelle de la machine. La fonction renvoie un code d'erreur si la machine choisie est incapable de faire le calcul en question, soit parce qu'elle ne dispose pas d'assez de mémoire, soit parce que la routine correspondante n'est pas installée.

Une interface de plus bas niveau existe également pour, par exemple, acquérir directement des informations sur les disponibilités du système.

```
fast_avail(ressource, hôte1, hôte2, &valeur)
```

permet ainsi d'obtenir la `valeur` actuellement disponible de `ressource`. Il peut s'agir de la charge CPU, de la quote-part de CPU dont disposerait un nouveau processus, de la quantité de mémoire vive disponible, de l'espace disque libre, ou bien de la latence ou de la bande passante du réseau entre deux machines. Le paramètre `hôte2` est destiné aux ressources réseaux, et est ignoré pour les ressources ne concernant qu'une seule machine.

FAST regroupe la description des arguments passés en paramètres aux routines au sein d'un tableau nommé `data_desc`. Chacun de ces arguments peut être un scalaire, un vecteur ou une matrice dont les éléments de base sont de types entier, double ou caractère. `data_desc` contient des informations sur chaque argument qui doivent permettre de prédire le temps d'exécution des routines. Dans le cas d'un scalaire, on stocke sa valeur. En revanche, dans le cas d'un vecteur ou d'une matrice, on ne garde que les caractéristiques structurelles, telles que les dimensions et la forme (*i.e.,* si la matrice est triangulaire inférieure, supérieure, bande, etc.). Il a été choisi de ne pas stocker le contenu d'une matrice dans `data_desc` car le temps de calcul d'une routine dépend des informations structurelles et non de ce contenu.

Les paramètres décrits dans `data_desc` représentent les paramètres logiques du problème et non les paramètres réels de l'appel à la routine. Ainsi, dans le cas du produit matriciel, la routine `dgemm`, de la bibliothèque BLAS, nécessite treize paramètres, qui sont des pointeurs vers les zones mémoires occupées par les matrices à multiplier ainsi que certaines grandeurs telles que les dimensions de ces matrices. Étant donné que `dgemm` calcule le produit $C = \alpha op(A) \times op(B) + \beta C$, où $op(A)$ (resp. $op(B)$) peut être A ou A^t (resp. B ou B^t), FAST ne considère que les cinq paramètres α, β, A, B et C et est capable de faire la conversion entre ces deux visions.

2.4 Extension pour la gestion de routines parallèles

2.4.1 Travaux relatifs

Pour obtenir un ordonnancement satisfaisant pour une application parallèle, il est nécessaire de déterminer à priori le temps de calcul de chacune des tâches qui la composent ainsi que les temps de communication induits par le parallélisme. La technique la plus courante pour

déterminer ces temps est de décrire l'application et de modéliser la machine parallèle qui l'exécute.

La modélisation de la machine parallèle et notamment des communications peut être considérée de différents points de vue. Ainsi, si l'on ignore les coûts de communications, modéliser une routine parallèle revient à utiliser la loi d'Amdahl. Ce type de modèle n'est pas réaliste dans le cas de grappes de PCs où la mémoire est physiquement distribuée. En effet, sur de telles plates-formes, les temps de communications représentent une fraction importante du temps d'exécution total d'une application et ne peut être négligé. De plus, cela ne permet pas de prendre en considération l'impact de la forme de la grille de processeurs sur les performances de la routine. Dans ces conditions, il devient en effet important de tenir compte du schéma de communication de la routine étudié afin de déterminer la forme et la taille de grille la plus adaptée. Par exemple, dans le cas de la routine de produit de matrices de la bibliothèque SCALAPACK, des grilles compactes permettent d'obtenir des performances bien meilleures qu'en utilisant des grilles de processeurs plus allongées. Ainsi, nous pouvons voir sur la figure 2.3 qu'une utilisation trop simpliste de la loi d'Amdahl, consistant à ajouter un processeur à la plate-forme d'exécution tant que le temps d'exécution obtenu est meilleur que celui précédemment estimé, a une validité limitée et peut ne pas conduire à l'utilisation de la plate-forme optimale.

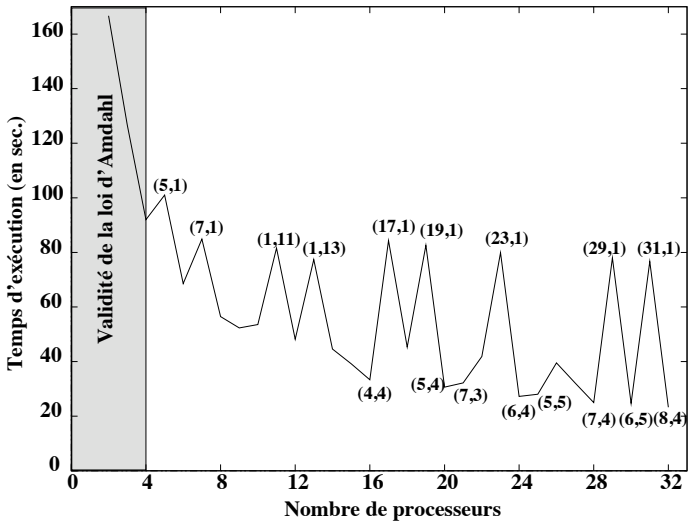


FIG. 2.3 – Temps d'exécutions obtenus pour la multiplication de matrices 4096×4096 sur la meilleure grille utilisant un nombre donné de processeurs.

Les modèles *délai* [106] et *LogP* [33] tiennent quant à eux compte des communications. Le premier sous forme d'un délai d constant, alors que le second considère quatre paramètres théoriques : le temps de transmission d'un processeur à un autre (L), le surcoût en calcul d'une communication (o), le débit du réseau (g) et le nombre de processeurs (P). Cependant, il nous a semblé que le modèle délai n'était pas assez précis et le modèle LogP trop compliqué pour modéliser d'une manière simple mais suffisamment réaliste une plate-forme de metacomputing.

En ce qui concerne la modélisation des algorithmes parallèles utilisées dans des bibliothèques telles que SCALAPACK, plusieurs approches sont possibles. Dans [50], les auteurs cherchent à utiliser des routines parallèles sur des grappes de PCs lorsqu'il est possible de faire mieux que le temps séquentiel. Pour cela, la modélisation utilisée consiste à identifier les appels séquentiels

et à remplacer ceux-ci par des fonctions dépendant de la taille des données et des performances relatives de la routine séquentielle sur un des nœuds de la grappe. Cette modélisation ne tient cependant pas compte des variations de charge de la plate-forme d'exécution, ni de l'impact des pipelines présents dans ces routines parallèles. Dans sa thèse [45], Stéphane Domas propose également un modèle de routines parallèles basé sur l'estimation des routines séquentielles. Dans son cas, cette estimation prend la forme de polynômes dont les variables sont les différentes tailles des matrices et où les coefficients des différents termes du polynôme dépendant de l'algorithme et de la machine utilisée. Ces paramètres sont déterminés en interpolant, dimension par dimension, un jeu de courbes obtenus par exécution de la routine sur des données de petites tailles. Le modèle proposé par Domas est fortement centré sur la routine de factorisation LU étudiée durant sa thèse. La technique employée est de plus difficilement extensible, l'estimation nécessitant la recompilation d'un code C généré en fonction du problème traité. L'environnement *ChronosMix* [14] utilise quant à lui le *micro-benchmarcking* pour estimer le temps d'exécution d'applications parallèles. Cette technique consiste effectuer des tests extensifs sur un jeu d'instructions C et MPI. Des codes sources, écrits dans ce langage et utilisant cette bibliothèque de communications, sont ensuite interprétés et leurs temps d'exécution déterminés. L'utilisation de cet environnement reste donc pour l'instant limitée aux codes C/MPI. Or, la plupart des bibliothèques numériques sont encore écrites en Fortran. De plus, dans le cas de SCALAPACK, les communications sont gérées par les BLACS, qui peuvent certes être implantées au dessus de MPI, mais aussi de PVM.

2.4.2 Choix de modélisation

Dans sa version actuelle, l'extension parallèle de FAST ne permet d'obtenir des estimations que pour certaines routines parallèles d'algèbre linéaire dense de la bibliothèque SCALAPACK. Dans le cas de telles routines, l'étape de description consiste à déterminer quels sont les équivalents BLAS appelés, leurs paramètres d'appels (taille des données, coefficients multiplicateurs, transpositions, etc.), les schémas de communication ainsi que les volumes de données échangés. FAST pouvant estimer tous les équivalents séquentiels, des appels aux fonctions de la bibliothèque suffisent alors pour prédire leurs temps d'exécution.

De plus, pour obtenir de bonnes performances, les processeurs exécutant un code SCALAPACK doivent être homogènes. Il en va de même pour le réseau reliant ces processeurs. Cela nous autorise deux simplifications notables. D'une part, les processeurs étant homogènes, l'étalonnage de la partie séquentielle de FAST peut n'être exécuté que sur un seul processeur. D'autre part, en ce qui concerne les communications, il suffit de ne surveiller que quelques liens représentatifs de la machine pour avoir un aperçu du comportement global de la plate-forme. Ce choix, même s'il peut paraître pénalisant du point de vue de la dynamicité, nous a paru simple et suffisant pour une première implantation.

Pour l'estimation des communications point–à–point, nous avons opté pour le modèle classique $\lambda + L\tau$ où λ est la latence du réseau, L la taille du message et τ le temps de transfert par élément, *i.e.*, l'inverse de la bande passante. L peut être déterminée lors de l'analyse, λ et τ pouvant quant à eux être estimés via des appels à FAST. Dans les opérations de diffusion, λ et τ sont remplacés par des fonctions dépendant de la forme de la grille de processeurs [20]. Si l'on considère une grille de processeurs à p lignes et q colonnes, et en supposant que les données soient distribuées uniformément sur les processeurs, λ_p^q sera la latence pour qu'un processeur d'une colonne donnée diffuse ses données aux processeurs qui sont sur la même ligne que lui et $1/\tau_p^q$ sera la bande passante. De la même manière, λ_q^p et $1/\tau_q^p$ dénoteront ces quantités pour une ligne de processeurs. Ces fonctions dépendent directement de l'implantation de la diffusion. Par exemple,

sur une grappe de stations de travail connectée par un commutateur, la diffusion pourra être implantée en utilisant un arbre. Dans ce cas, λ_p^q sera égal à $\lceil log_2\, q \rceil \times \lambda$ et τ_p^q égal à $(\lceil log_2\, q \rceil / p) \times \tau$. Dans ce cas, λ doit être interprétée comme la latence d'un nœud et $1/\tau$ comme la bande passante moyenne.

Ce travail peut donc être considéré à la fois comme un client de FAST et comme une extension pour la gestion des routines parallèles. En effet, les estimations de la partie séquentielle de FAST sont injectées dans un modèle obtenu par analyse de code. Une fois ce couplage effectué, le temps d'exécution de la routine parallèle modélisée peut être prédit par FAST et est donc accessible par l'interface utilisateur standard.

2.4.3 Exemples de modèles de routines parallèles

2.4.3.1 Produit de matrices denses

La routine `pdgemm` de la bibliothèque SCALAPACK est la version parallèle de la routine `dgemm`. Elle calcule donc également le produit $C = \alpha op(A) \times op(B) + \beta C$ où $op(A)$ (resp. $op(B)$) peut être A ou A^t (resp. B ou B^t). Par souci de simplification, nous ne nous intéresserons ici qu'au cas $C = AB$, les autres cas étant similaires. A est une matrice $M \times K$, B une matrice $K \times N$, et la matrice résultat C est de dimension $M \times N$. Étant donné que ces matrices sont distribuées de manière cyclique par blocs sur une grille $p \times q$ de processeurs, la taille d'un bloc étant R, le temps de calcul s'exprime de la manière suivante :

$$\left\lceil \frac{K}{R} \right\rceil \times \texttt{temps\_dgemm}, \tag{2.1}$$

où `temps_dgemm` est obtenu par l'appel FAST suivant :

```
fast_comp_time (hôte, dgemm_desc, &temps_dgemm),
```

où `hôte` est une des machines impliquées dans l'exécution d'un appel à `pdgemm`. Les matrices passées en paramètres dans `dgemm_desc` pour cet appel FAST sont de tailles $\lceil M/p \rceil \times R$ pour le premier opérande et $R \times \lceil N/q \rceil$ pour le second.

Pour estimer le temps de communication, il est important de considérer le schéma de communication de la routine `pdgemm`. À chaque étape, les pivots, *i.e.*, une colonne de blocs et une ligne de blocs, sont diffusés à l'ensemble des processeurs pour permettre l'exécution des multiplications de manière indépendante. Les quantités de données communiquées sont donc $M \times K$ pour la diffusion des lignes et $K \times N$ pour la diffusion des colonnes. Chacune de ces diffusions est effectuée bloc par bloc. On obtient donc :

$$(M \times K)\tau_p^q + (K \times N)\tau_q^p + \left(\lambda_p^q + \lambda_q^p\right) \left\lceil \frac{K}{R} \right\rceil. \tag{2.2}$$

Cela nous donne donc l'estimation suivante pour la routine `pdgemm` :

$$\left\lceil \frac{K}{R} \right\rceil \times \texttt{temps\_dgemm} + (M \times K)\tau_p^q + (K \times N)\tau_q^p + \left(\lambda_p^q + \lambda_q^p\right) \left\lceil \frac{K}{R} \right\rceil. \tag{2.3}$$

Lorsque l'on dispose d'une diffusion par arbre, on peut remplacer τ_p^q, τ_q^p, λ_p^q et λ_q^p par leurs valeurs en fonction de τ et λ. Ces deux quantités sont estimées par les appels suivants à FAST :

```
fast_avail (bande_passante, source, dest, &tau)
```

et

```
fast_avail (latence, source, dest, &lambda),
```

où le lien {`source ; dest`} est un des liens surveillés par FAST. L'équation 2.2 devient alors :

$$\frac{\left(\frac{\lceil log_2\, q\rceil \times M \times K}{p} + \frac{\lceil log_2\, p\rceil \times K \times N}{q}\right)}{\tau} + \left\lceil \frac{K}{R} \right\rceil (\lceil log_2\, q\rceil + \lceil log_2\, p\rceil) \times \lambda. \tag{2.4}$$

2.4.3.2 Résolution triangulaire

La routine `pdtrsm` de la bibliothèque SCALAPACK, dans sa version 1.6, permet de résoudre les systèmes linéaires suivants : $op(A) \times X = \alpha B$ et $X \times op(A) = \alpha B$, où $op(A)$ peut être A ou A^t. A est une matrice $N \times N$ triangulaire inférieure ou supérieure qui peut être unitaire ou non. X et B sont deux matrices $M \times N$.

Inférieure Supérieure	Gauche Droite	Transposition	Opération	Cas
S	G	N	B = A \ α B	1
S	D	T		
S	G	T	B = α B / A	2
S	D	N		
I	G	N	B = A \ α B	3
I	D	T		
I	G	T	B = α B / A	4
I	D	N		

FIG. 2.4 – Correspondance entre paramètres d'appels et calculs effectifs pour la routine `pdtrsm`.

La figure 2.4 présente les équivalents graphiques d'un appel à la routine `pdtrsm` en fonction de la valeur des trois paramètres `UPLO`, `SIDE` et `TRANSA` qui déterminent respectivement si A est triangulaire inférieure ou supérieure, si X est un opérande gauche ou droite et enfin si A est transposée ou non. Les huit combinaisons possibles peuvent être regroupées en quatre cas comme nous pouvons le voir sur la figure 2.4. Ces cas peuvent à leur tour être regroupés en deux familles dont les performances peuvent être diamétralement opposées en fonction de la forme de la grille de processeurs. La première famille comprend les cas 1 et 3 qui s'exécuteront plus rapidement sur des grilles où le nombre de lignes est nettement supérieur au nombre de colonnes. À l'inverse, les cas 2 et 4, appartenant à la seconde famille, s'exécuteront plus rapidement sur des grilles où le nombre de colonnes est nettement supérieur au nombre de lignes. Enfin, un appel à `pdtrsm` est en fait composé d'appels à la sous-routine `pbdtrsm` qui résout le même système mais lorsque le nombre de lignes de B est inférieur où égal à la taille d'un bloc, R. Ainsi, pour obtenir le coût d'un appel à `pdtrsm`, il faudra multiplier le coût d'un appel à `pbdtrsm` par $\lceil M/R \rceil$.

Pour comprendre pourquoi certaines formes de grilles sont clairement plus efficaces que d'autres, nous nous sommes restreints au cas de la résolution du système $XA = B$ où A est une matrice triangulaire supérieure non unitaire et non transposée. La figure 2.5 montre les temps obtenus pour différentes formes de grilles impliquant 8 processeurs. Dans ce cas particulier, (cas 2 dans la figure 2.4), nous pouvons voir qu'une ligne de 8 processeurs (*i.e.*, 1×8) permet d'obtenir des performances jusqu'à 5 fois meilleures qu'une colonne de 8 processeurs (*i.e.*, 8×1). Nous avons donc tout d'abord concentré notre analyse sur le cas où la grille de processeurs est une ligne. Nous avons ensuite étendu notre modèle au cas plus général d'une grille rectangulaire.

Pour résoudre le système d'équations présenté par la figure 2.6, où A est une matrice $N \times N$,

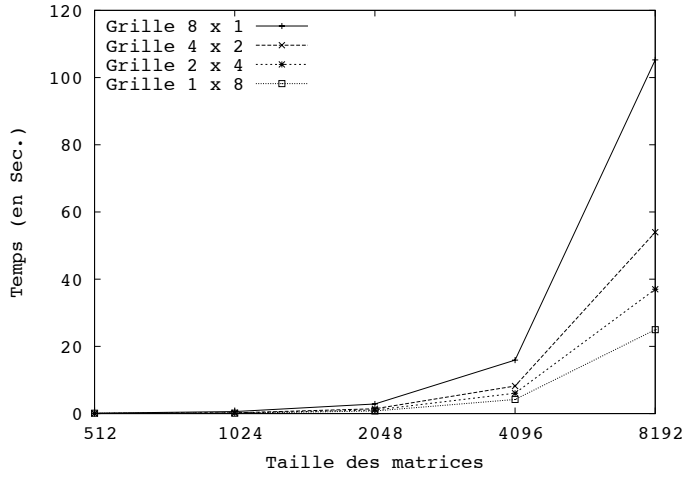


FIG. 2.5 – Temps d'exécution d'un `pbdtrsm` pour différentes formes de grilles.

$$
\begin{aligned}
B_{11} &= B_{11}/A_{11} \\
B_{12} &= (B_{12} - B_{11}A_{12})/A_{22} \\
&\vdots \quad\quad \vdots \\
B_{1j} &= (B_{1j} - B_{11}A_{1j} - \cdots - B_{1(j-1)}A_{(j-1)j})/A_{jj} \\
&\vdots \quad\quad \vdots \\
B_{1j} &= (B_{1n} - B_{11}A_{1N} - \cdots - \cdots - B_{1(N-1)}A_{(N-1)N})/A_{NN}
\end{aligned}
$$

FIG. 2.6 – Calcul effectué lors d'un appel à `pbdtrsm`, où A est une matrice $N \times N$.

le principe est le suivant. Le processeur qui détient le bloc diagonal courant A_{ii} effectue une résolution triangulaire séquentielle pour calculer B_{1i}. Le bloc résultant de ce calcul est diffusé aux processeurs se trouvant sur la même ligne. Les processeurs recevant ce bloc peuvent alors mettre à jour les blocs non encore résolus qu'ils détiennent, *i.e.*, calculer $B_{1j} - B_{1i}A_{ij}$, pour $i < j \leq n$. Cette séquence est ensuite répétée pour chaque bloc diagonal, comme le montre la figure 2.7.

Le temps d'exécution d'une résolution triangulaire séquentielle, `temps_trsm`, sera estimé par :

```
fast_comp_time (hôte, trsm_desc, &temps_trsm),
```

où `hôte` est l'une des machines impliquées dans le calcul du `pdtrsm`. Les matrices passées en argument dans `trsm_desc` sont de taille $R \times R$.

Les blocs résolus sont diffusé le long de l'anneau formé par la ligne de processeurs, mais le chemin critique du `pbdtrsm` suit également cet anneau. Nous n'avons donc qu'à considérer la communication entre le processeur qui calcule la résolution triangulaire séquentielle et son voisin de droite. La quantité de données communiquée lors de chaque diffusion étant R^2, nous pouvons estimer une opération de diffusion par la formule suivante :

$$T_{diffusion} = R^2\tau + \lambda, \tag{2.5}$$

où les valeurs de λ et τ sont estimées par les appels FAST présentés dans le modèle de la

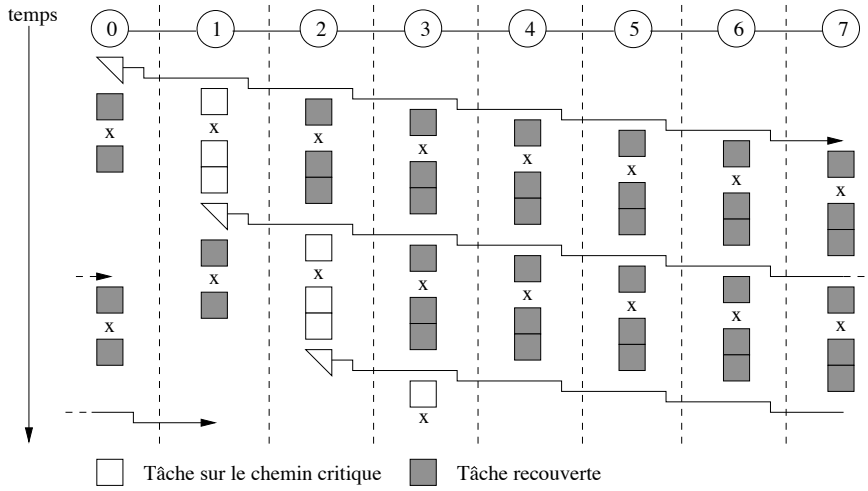


FIG. 2.7 – Exécution d'un `pbdtrsm` sur une ligne de 8 processeurs.

multiplication de matrices.

À chaque étape, la phase de mise à jour est effectuée en appelant la routine `dgemm`. Le premier opérande de ces appels est une copie du bloc résolu à cette étape, et est par conséquent toujours de taille $R \times R$. Le nombre de colonnes du second opérande dépend quant à lui du nombre de blocs qui ont déjà été résolus eut être exprimé de la manière suivante : $R\lceil (N - iR)/(qR) \rceil$, où i est le nombre de blocs résolus. L'appel FAST correspondant à l'estimation à un tel appel à `dgemm` est alors :

```
fast_comp_time (hôte, dgemm_desc_ligne, &temps_dgemm_ligne).
```

Des intervalles d'inactivité peuvent apparaître si l'un des récepteurs de la diffusion est toujours en train d'effectuer la mise à jour correspondant à l'étape précédente lorsqu'il est contacté par son voisin de gauche. Notre modèle estimant le temps d'exécution de la routine en suivant son chemin critique, ces intervalles d'inactivité sont gérés en appliquant une correction, notée C_b au temps de diffusion. Cette correction est calculée en fonction du maximum des temps précédemment estimés pour l'émetteur et le récepteur d'une communication. Cela correspond en effet au temps que ces processeurs doivent attendre avant de pouvoir réaliser la communication. L'équation 2.6 donne le modèle de coût de la routine `pbdtrsm` exécutée sur une ligne de processeurs.

$$\sum_{i=1}^{\lceil N/R \rceil} (\texttt{temps\_trsm} + T_{diffusion} + C_b + \texttt{temps\_dgemm\_ligne}) . \tag{2.6}$$

La principale différence entre le cas « ligne » et le cas général se situe au niveau de la phase de mise à jour. En effet, un mécanisme de pipeline est utilisé dans le cas d'une grille rectangulaire. Ce pipeline s'effectue en découpant la phase de mise à jour en deux étapes. Durant la première, les $p-1$ premiers blocs sont mis à jour, où p est le nombre de lignes de la grille de processeurs, les blocs restants étant traités durant la deuxième étape. Une fois la première partie des blocs mise à jour, elle est communiquée au processeur situé sur la même colonne et sur la ligne suivante, modulo p. Le processeur qui reçoit ces blocs effectue alors une accumulation pour terminer la mise à jour de ses propres blocs, comme le montre la figure 2.8.

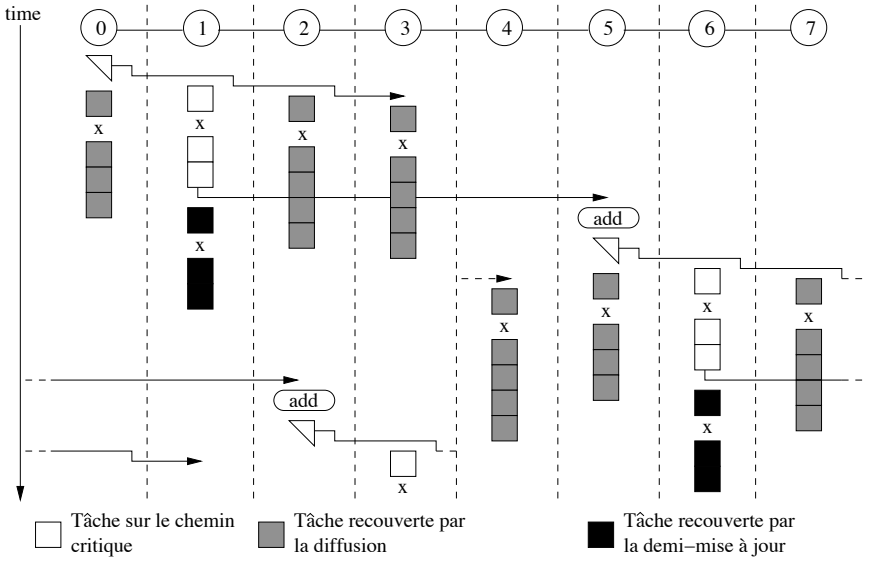


FIG. 2.8 – Exécution d'un `pbdtrsm` sur une grille 2×4 de processeurs.

Cette optimisation implique deux valeurs distinctes pour le nombre de colonnes du second opérande des appels à `dgemm`. La première peut s'exprimer comme le minimum entre $R\lceil (N-iR)/(QR) \rceil$ et $R(P-1)$, alors que la seconde sera $R(\lceil (N-iR)/(QR) \rceil - (P-1))$ si cette quantité est positive. Nous avons donc besoin des deux appels FAST suivants pour estimer les temps d'exécution des deux appels à la routine `dgemm` effectués au cours d'une mise à jour :

```
fast_comp_time (hôte, dgemm_desc_1, &temps_dgemm_1)
```

et

```
fast_comp_time (hôte, dgemm_desc_2, &temps_dgemm_2).
```

Restent deux opérations à estimer pour compléter le modèle de coût dans le cas général : les opérations d'*envoi* et d'*accumulation*. Pour ces deux opérations, le nombre de colonnes du second opérande est calculé de la même manière que pour le premier produit de matrices. Cela nous conduit donc aux expressions suivantes :

$$T_{envoi} = \left(R \times min \left((P-1)R, R \left\lceil \frac{N-iR}{QR} \right\rceil \right) \right) \tau + \lambda, \tag{2.7}$$

et

```
fast_comp_time (hôte, add_desc, &temps_addition).
```

Les intervalles d'inactivité pouvant apparaître lors de l'opération d'envoi sont gérés de la même manière que pour la diffusion. La correction appliquée à cette opération est notée C_u. L'équation 2.8 donne le modèle de coût de la routine `pbdtrsm` exécutée sur une grille rectangulaire de processeurs.

$$\sum_{i=1}^{\lceil N/R \rceil} (\texttt{temps\_trsm} + T_{diffusion} + C_b + \texttt{temps\_dgemm\_1} + T_{envoi} + C_u + \texttt{temps\_addition}). \tag{2.8}$$

2.5 Validation expérimentale

Après avoir étudié le fonctionnement et l'interface de FAST, nous allons maintenant présenter quelques résultats expérimentaux validant ces travaux. Nous allons tout d'abord étudier la qualité des prédictions de FAST, aussi bien en ce qui concerne les routines séquentielles que les routines parallèles. Ensuite, nous montrerons combien FAST peut être utile à un ordonnanceur pour décider du meilleur scénario d'exécution possible.

2.5.1 Étude la précision des prédictions

Afin de tester la validité de l'approche choisie, nous avons mis au point une série d'expériences pour étudier la qualité des estimations produites par FAST. La première expérience ne porte que sur la modélisation des besoins (en temps et en espace) des routines sans tenir compte de la charge. La seconde traite de la prédiction du temps de calcul d'opérations séquentielles en tenant compte de la charge. La troisième porte sur la prédiction du temps de calcul d'une séquence d'opérations séquentielles sur une plate-forme hétérogène. La quatrième teste la précision de l'estimation d'une routine parallèle en fonction de la taille et de la forme de la grille de processeurs utilisée. Enfin, la dernière expérience valide la prédiction du temps d'exécution d'une routine parallèle pour une grille de processeurs donnée.

2.5.1.1 Modélisation des besoins

Avant de mesurer la qualité des prédictions de FAST, il nous a semblé important de mesurer la qualité de la modélisation utilisée pour les besoins des routines sans tenir compte de la charge. Pour cela, nous avons comparé la valeur modélisée à la valeur mesurée pour le temps et l'espace mémoire nécessaires à l'exécution de trois routines séquentielles de la bibliothèque BLAS : l'addition de matrices ; le produit de matrices ; et la résolution triangulaire. Cette comparaison a été effectuée sur deux machines différentes : un Pentium III 733 MHz disposant de 256 Mo de mémoire ; et d'un Bi-Pentium II 450 MHz avec également 256 Mo de mémoire.

	Addition		Produit		Résolution	
	PIII	*Bi-PII*	*PIII*	*Bi-PII*	*PIII*	*Bi-PII*
Erreur Maximale	0.02s (6 %)	0.02s (35 %)	0.21s (0.3 %)	5.8s (4 %)	0.13s (10 %)	0.31s (16 %)
Erreur Moyenne	0.006s (4 %)	0.007s (6.5 %)	0.025s (0.1 %)	0.03s (0.1 %)	0.02s (5 %)	0.08s (7 %)

TAB. 2.1 – Qualité de la modélisation temporelle de routines séquentielles.

Le tableau 2.1 présente la qualité de la modélisation temporelle des trois fonctions étudiées sur ces deux machines pour des tailles de matrices variant entre 128 et 1152. Sur la première ligne figure la valeur absolue de l'erreur maximale constatée lors de l'expérience. L'erreur relative est indiquée entre parenthèses. La seconde ligne donne la moyenne de l'erreur, à la fois en valeur absolue et en valeur relative. Nous pouvons constater que l'erreur maximale commise est généralement inférieure à 0,2 secondes. L'erreur moyenne est quant à elle de l'ordre du centième

de seconde. Cette précision est donc tout à fait acceptable dans le contexte dans lequel nous nous plaçons, puisque le temps de réponse de NWS est de l'ordre du dixième de seconde. L'erreur relative est quant à elle d'environ 5 % en moyenne, et d'un peu plus de 15 % dans le pire des cas.

Dans le cas de l'addition, les erreurs relatives sont plus importantes. Ceci s'explique par le fait que nous avons choisi d'effectuer les mesures dans FAST avec la fonction `rusage`, qui donne le temps système et utilisateur du processus en cours. Cette méthode est insensible à la charge externe, mais sa précision n'est que de 0,01 secondes. L'addition de matrice durant moins d'une demi seconde, l'erreur notée sur ce tableau n'est donc pas seulement l'erreur de modélisation, mais regroupe également des erreurs de mesure.

La modélisation spatiale n'est pas représentée car FAST parvient à modéliser parfaitement l'espace nécessaire à l'exécution des routines en fonction des paramètres. Cette qualité est possible car la taille d'un programme effectuant une opération matricielle correspond à la taille des données, plus une constante pour le code du programme. Le tout est donc facilement exprimable sous forme d'une fonction polynomiale dépendant de la taille des matrices.

2.5.1.2 Prédiction du temps d'exécution d'une routine séquentielle

Le but de cette expérience est de mesurer la qualité des prédictions de FAST en tenant compte de la charge. Cette expérience ne porte que sur le temps d'exécution, car la charge extérieure n'a aucune influence sur la quantité de mémoire nécessaire à l'exécution d'une routine. Nous avons comparé la valeur mesurée à la celle prédite par FAST pour le temps d'exécution de la routine `dgemm`. La charge extérieure est simulée par l'exécution d'un programme de calcul pendant l'expérience. La plate–forme de test utilisée est la même que pour l'expérience précédente. Les résultats présentés dans la figure 2.9 sont des moyennes sur cinq exécutions. Malgré la charge externe, FAST parvient à prédire le temps de calcul avec une erreur maximale de 22 %, et une erreur moyenne inférieure à 10 %.

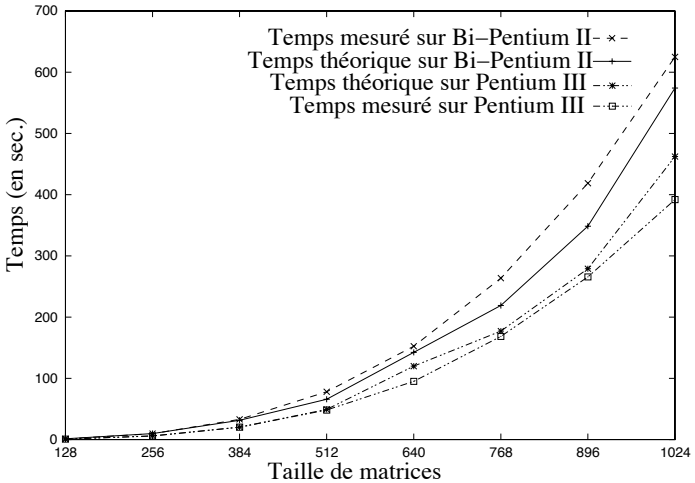


FIG. 2.9 – Comparaison du temps réel et de la prédiction pour une exécution de la routine `dgemm`.

2.5.1.3 Prédiction du temps d'exécution d'une séquence de routines séquentielles

Pour juger de la qualité des prédictions de FAST sur une séquence d'opérations séquentielles, nous avons choisi de réaliser un produit de matrices complexes. Les données sont deux matrices complexes A et B, séparées en parties réelles et imaginaires. Le problème à résoudre est :

$$C = \begin{cases} C_r = A_r \times B_r - A_i \times B_i \\ C_i = A_r \times B_i + A_i \times B_r \end{cases}$$

L'objectif étant de mesurer la précision de la prédiction et non son impact sur les décisions d'ordonnancement, nous avons mis en place une plate-forme d'évaluation hétérogène composée de seulement deux machines. Le processus client s'exécute sur une station de travail dotée d'un Pentium II et de 128 Mo de mémoire tandis que deux serveurs de calcul sont lancés sur une machine SMP dotée de quatre processeurs Pentium III et de 256 Mo de mémoire. Le client interroge FAST pour obtenir une prédiction de temps de calcul de la séquence, puis initie le calcul et mesure le temps réel. La répartition (statique) des sous–calculs entre les machines est la suivante :

- Sur le client :
  Envoi de A_r, A_i et B_r au serveur 1 ;
  Envoi de A_r, A_i et B_i au serveur 2.

- Sur le serveur 1 :
  $C_{r_1} = A_r \times B_r$;
  $C_{i_2} = A_i \times B_r$;
  Envoi de C_{i_2} au serveur 2 ;
  $C_r = C_{r_1} - C_{r_2}$;
  Envoi de C_r au client.

- Sur le serveur 2 :
  $C_{r_2} = A_i \times B_i$;
  $C_{i_1} = A_r \times B_i$;
  Envoi de C_{r_2} au serveur 1 ;
  $C_i = C_{i_1} + C_{i_2}$;
  Envoi de C_i au client.

La figure 2.10 montre la comparaison entre les temps de calcul prédit et mesuré pour le produit de matrices complexes en fonction de la taille des matrices à multiplier. Nous pouvons constater une évolution similaire des deux courbes. Pour une séquence composée de six calculs de deux types différents et de six échanges de matrices, FAST parvient à prédire le temps nécessaire avec un taux d'erreur inférieur à 25 % dans le pire des cas, et à 12 % en moyenne.

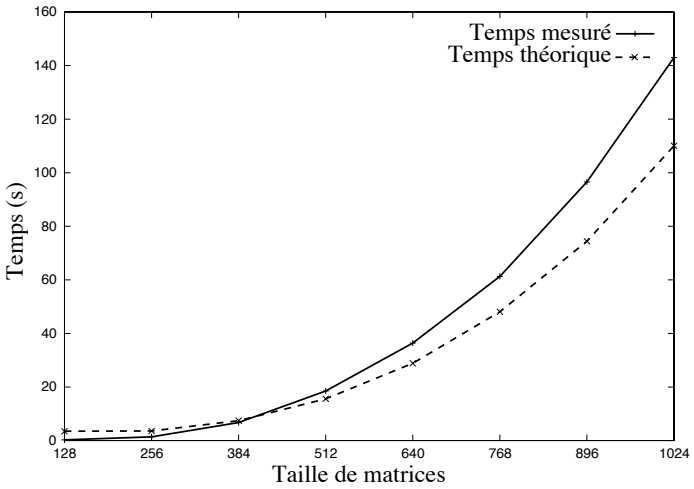


FIG. 2.10 – Comparaison des temps mesurés et prédits pour une séquence d'opérations.

2.5.1.4 Prédiction du temps d'exécution d'une routine parallèle

Afin de valider notre gestion des routines parallèles dans FAST, nous avons effectué divers tests basés sur le produit de matrices ScaLAPACK. Ces tests ont été exécutés sur l'*i-cluster*.

Dans cette expérience, nous avons cherché à valider la précision de notre extension pour une grille de processeurs donnée. La figure 2.11 montre le taux d'erreur de la prédiction par rapport au temps d'exécution mesuré pour des produits de matrices effectués sur une grille 8×4 de processeurs. Les tailles de matrices employées vont de 1024 à 10240. Notre extension s'avère très précise puisque nous obtenons un taux d'erreur inférieur à 3%.

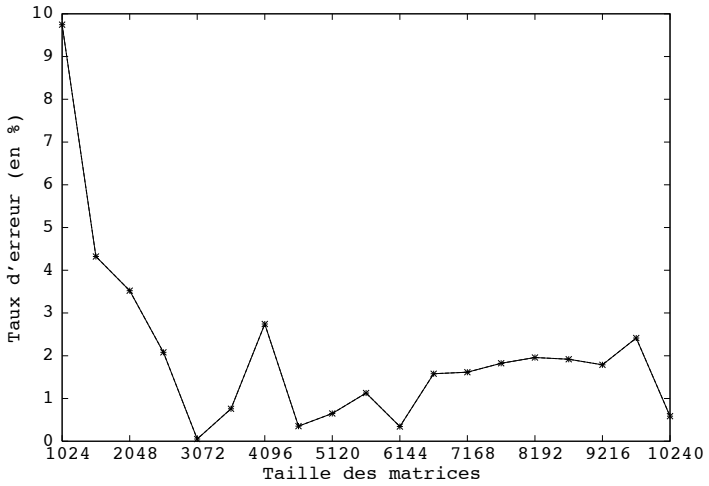


FIG. 2.11 – Taux d'erreur entre prédiction et temps d'exécution sur une grille 8×4 de processeurs.

2.5.1.5 Temps d'exécution d'une routine parallèle en fonction de la forme de la grille

La figure 2.12 présente une comparaison entre le temps prédit par notre extension de FAST pour la gestion des routines parallèles (haut) et le temps mesuré (bas) pour la routine `pdgemm` pour toutes les grilles possibles comprenant entre 1 et 32 processeurs de l'*i-cluster*. Les matrices sont toutes de taille 2048×2048. L'axe des x représente le nombre de lignes de la grille de processeurs, l'axe des y le nombre de colonnes et l'axe des z le temps en secondes. Sur cette courbe nous pouvons voir que FAST permet d'estimer avec précision le temps d'exécution d'un produit de matrices parallèle, et ce pour n'importe quelle forme de grille de processeurs. L'erreur maximale est de moins de 15% et l'erreur moyenne inférieure à 4%. Cette expérience nous permet également de constater l'impact de la forme de la grille sur les performances. Les grilles compactes induisent de meilleurs résultats que les grilles allongées du fait du schéma de communication symétrique de la routine. Les paliers observés pour les grilles « ligne » et « colonne » proviennent du terme en *log* introduit par l'arbre de diffusion.

Nous avons également mené ce type d'expérience pour la résolution triangulaire. La figure 2.13 présente les estimations fournies par notre modèle. En comparant ces résultats avec ceux de la figure 2.5, nous pouvons voir que notre modèle permet de prédire l'évolution des performances en fonction de la forme de la grille de processeurs utilisée. L'erreur moyenne obtenue est inférieure à 12%. Il est à noter que si ce taux d'erreur est bien supérieur à celui obtenu avec le

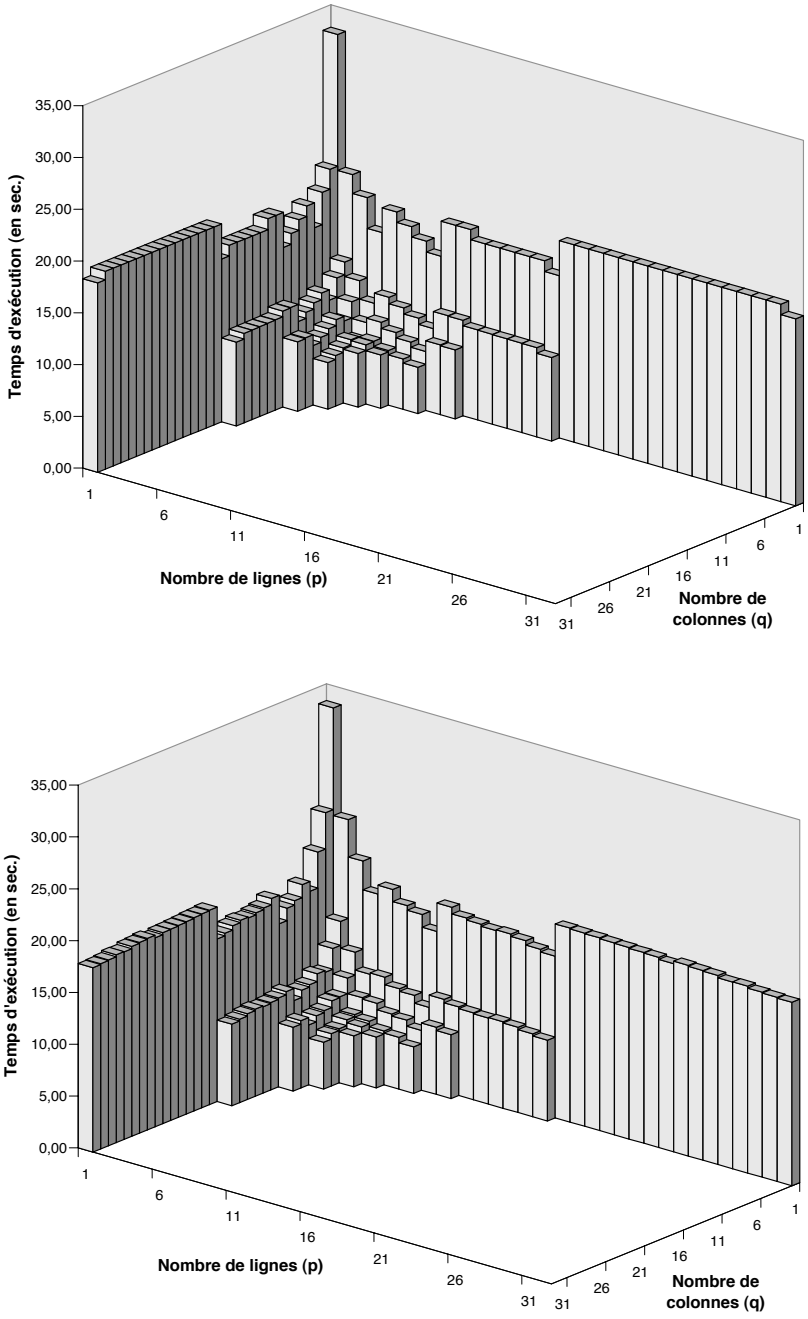


FIG. 2.12 – Comparaison entre temps estimés (haut) et temps mesurés (bas) pour l'exécution de la routine `pdgemm` sur toutes les grilles possibles comprenant de 1 et 32 processeurs de l' *i-cluster*.

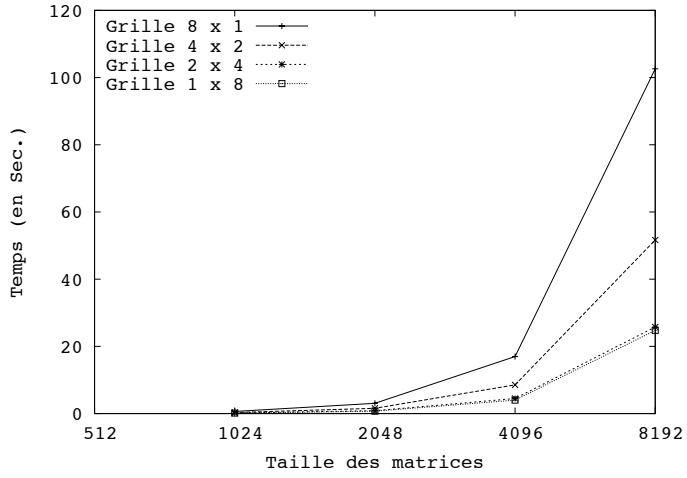


FIG. 2.13 – Estimations du temps d'exécution d'un `pbdtrsm` pour différentes formes de grilles.

modèle du produit de matrices, cela provient essentiellement des difficultés à modéliser les optimisations apportées à cette routine par l'utilisation de pipelines. Le modèle s'avère notamment très imprécis sur le cas d'une grille 2×4, pour laquelle le temps d'exécution est fortement sous estimé. En revanche, si l'on ne considère que l'exécution sur une ligne de processeurs, ce qui, rappelons le, s'avère être la meilleure plate-forme pour le cas considéré, le taux d'erreur est alors inférieur à 5%.

2.5.2 Utilité dans un contexte d'ordonnancement

L'objectif de FAST est de fournir des informations précises permettant à l'application cliente, typiquement un ordonnanceur, de déterminer quelle est la meilleure solution entre différentes possibilités, comme dans l'exemple présenté par la figure 2.1. La figure 2.14 (gauche) présente une configuration de ce type où la matrice A est distribuée sur la grille G_a et la matrice B sur la grille G_b. Ces deux grilles de processeurs peuvent être agrégées de différentes manières afin de former une grille virtuelle plus puissante. Nous avons retenu deux grilles pour cette expérience : une compacte, G_{v1} ; et une plus allongée, G_{v2}. Ces grilles sont en réalité des ensembles de processeurs de l'*i-cluster*. Les processeurs sont donc homogènes et les coûts de communication inter- et intra–grilles peuvent être considérés comme similaires.

Malheureusement, la version actuelle de FAST n'est pas capable d'estimer le coût d'une redistribution entre deux ensembles de processeurs. Ce problème est en effet très difficile dans le cas général [37]. Pour les besoins de cette expérience, nous avons donc déterminé les volumes de données transmis entre chaque couple de processeurs ainsi que le schéma de communication engendré par la routine de redistribution de ScaLAPACK [97]. Nous avons ensuite utilisé FAST pour estimer les coûts des différentes communications point–à–point générées. La figure 2.14 (droit) compare les temps estimés et mesurés pour chacune des grilles présentées en figure 2.14 (gauche).

Nous pouvons constater que l'utilisation de FAST permet de déterminer avec précision quelle est la solution la plus rapide, à savoir celle utilisant une grille 4×3 de processeurs. Si cette solution est la plus intéressante en ce qui concerne le calcul, elle est en revanche la moins efficace pour ce qui est de la redistribution. L'utilisation de FAST peut donc permettre d'effectuer une présélection

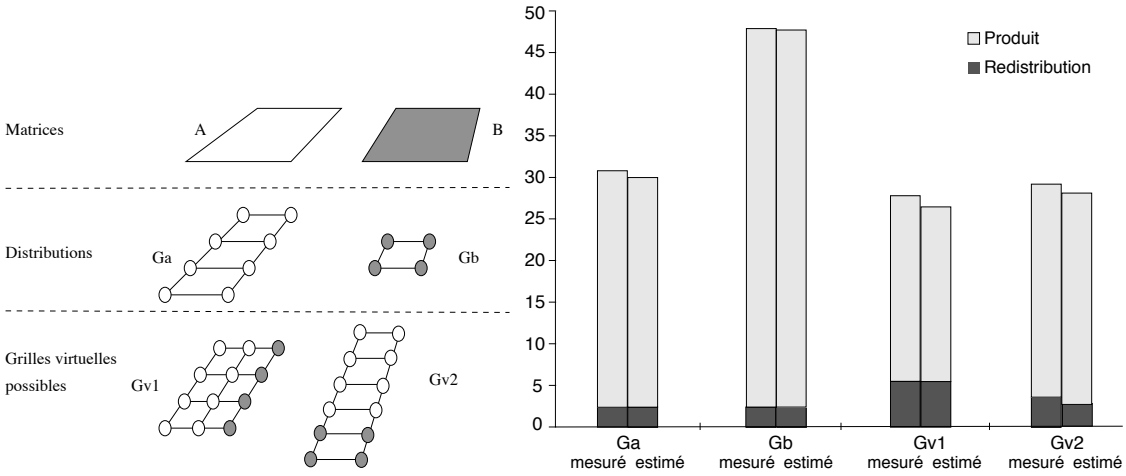


FIG. 2.14 – Validation de la gestion des routines parallèles dans le cas d'un alignement de matrices suivi d'un produit pour 4 grilles virtuelles de processeurs (gauche). Les temps estimés sont comparés aux temps mesurés (droit) en distinguant la redistribution et le calcul.

en fonction du ratio entre puissance des processeurs et débit réseau. De plus, il est intéressant de constater que si la solution consistant à effectuer le produit sur G_a est un peu plus coûteuse en temps, elle génère moins de communications et libère 4 processeurs pour d'éventuelles tâches en attente.

2.6 Conclusion

Dans ce chapitre, nous avons présenté FAST, un outil de modélisation et de prédiction de performances dans un contexte de metacomputing. Nous avons étudié le contenu et le fonctionnement des différents modules de cet outil ainsi que l'interface offerte à ses utilisateurs. Nous avons ensuite présenté une extension de FAST qui permet d'estimer également le temps d'exécution de routines parallèles de la bibliothèque SCALAPACK. Les estimations obtenues ont été validées expérimentalement, aussi bien pour la partie séquentielle que pour la partie parallèle. Nous avons également montré l'utilité d'un outil comme FAST pour un ordonnanceur d'environnement de type client–agent(s)–serveurs.

Nous envisageons d'étendre notre extension pour gérer plus de routines parallèles et notamment les fonctions de factorisation contenues dans SCALAPACK. En effet, la routine de factorisation LU de SCALAPACK n'est composée que d'appels aux fonctions de résolution triangulaire et de produit de matrices. Il semble donc aisé d'extraire un modèle pour cette routine qui sera basé sur ceux présenté dans ce chapitre.

Nous souhaitons également améliorer l'estimation des coûts de redistribution. Si le cas général est un problème difficile, il nous semble possible de baser nos estimations sur un ensemble de classes de redistribution. Ces classes de redistributions sont construites en fonction des modifications apportées à la grille de départ pour obtenir la grille destination. Par exemple, les redistributions de G_a et G_b vers G_{v2} effectuées en section 3.3.8 appartiendraient à la classe de redistributions utilisant uniquement une augmentation proportionnelle du nombre de lignes de la grille.

Chapitre 3

Parallélisme mixte

Mais bien entendu qu'on peut faire plus vite ! Mais on est limité par le temps !
Yves Christen, homme politique suisse.

3.1 Introduction

Les applications scientifiques parallèles peuvent être divisées en deux grandes classes : les applications qui utilisent le paradigme du *parallélisme de données*, d'une part, et celles qui emploient celui du *parallélisme de tâches*, d'autre part. La première méthode consiste à appliquer la même opération en parallèle sur différents éléments d'un ensemble de données, alors que la seconde est définie comme étant l'exécution concurrente de calculs distincts sur des ensembles de données différents. Ces deux classes peuvent être combinées pour obtenir une exploitation simultanée des parallélismes de données et de tâches, appelée *parallélisme mixte*. Dans les applications utilisant le paradigme du parallélisme mixte, plusieurs tâches utilisant le parallélisme de données peuvent être exécutées de façon concurrente à la manière du parallélisme de tâches. L'exploitation du parallélisme mixte a plusieurs avantages dont celui d'augmenter l'extensibilité d'un programme en exhibant plus de parallélisme.

Il est également possible de diviser les applications scientifiques en deux classes, selon un autre critère. Celui-ci concerne la régularité des applications. Nous avons d'une part les applications régulières dans lesquelles les structures de données employées sont des tableaux denses et où les accès à ces structures peuvent aisément être déterminés à la compilation. Nous avons d'autre part les applications irrégulières où certaines structures de données peuvent être des matrices creuses dont la structure ne peut être déterminée qu'à l'exécution. Dans cette thèse, nous ne nous sommes intéressés qu'au domaine des applications régulières.

Après avoir présenté les différents travaux existant dans le domaine du parallélisme mixte, nous détaillerons l'application de ce paradigme de programmation aux algorithmes rapides de produit de matrices de Strassen et Winograd. Cela nous a permis de constater si l'exploitation simultanée des parallélismes de tâches et de données a un réel impact sur les performances d'algorithmes numériques. Nous présenterons alors diverses implantations, développées selon différentes stratégies de placement issues de l'analyse des algorithmes de base, en milieux homogène et hétérogène. Nous donnerons également une analyse théorique des implantations présentées en les comparant à des implantations utilisant exclusivement le parallélisme de données. Les résultats obtenus par cette analyse théorique ont été validés expérimentalement.

Nous avons ensuite cherché à automatiser le processus d'ordonnancement en proposant un algorithme dans lequel l'allocation et l'ordonnancement des tâches d'un graphe sont effectués simultanément. Cet algorithme est principalement basé sur les estimations des temps d'exécution et de communication fournies par FAST et son extension parallèle. Notre avons limité notre étude au cas où les données initiales de l'application sont déjà distribuées, mais pas nécessairement alignées, sur la plate-forme d'exécution. De plus, nous avons ajouté une contrainte de non réplication des données pour diminuer l'espace mémoire nécessaire à l'exécution de l'application à ordonnancer. Cet algorithme et sa complexité seront présentés au paragraphe 3.4

3.2 Travaux précédents

La plupart des recherches concernant la parallélisme mixte ont été effectuées dans le domaine des langages de programmation. Bal et Haines [6] présentent ainsi une étude de plusieurs langages visant à intégrer à la fois le parallélisme de tâches et le parallélisme de données. Pour réaliser une telle intégration, deux approches peuvent être suivies : il est tout d'abord possible d'ajouter du contrôle dans un langage à parallélisme de données, tel que HPF par exemple ; à l'inverse, certains projets introduisent de nouvelles structures pour la gestion du parallélisme de données dans des langages utilisant le parallélisme de tâches. Selon les auteurs, les avantages d'un tel langage combinant les deux formes de parallélisme sont multiples. Tout d'abord, cela permettrait de coder un très large éventail d'applications, la plupart étant des applications numériques qui exhibent les deux types de parallélisme. De plus, l'extensibilité des codes serait accrue par rapport aux implantations qui, du fait des langages de programmation disponibles, n'utilisent qu'une des deux formes de parallélisme. Enfin, le paradigme du parallélisme mixte semble être une manière simple et efficace de coupler des applications existantes utilisant le parallélisme de données. Cependant, l'intégration des parallélismes de tâches et de données au sein d'un même langage soulève quelques problèmes dus aux spécificités de chacun des deux modes de programmation. En effet, dans les langages utilisant le parallélisme de données, tous les efforts d'optimisation sont fournis par le compilateur alors que dans les langages à parallélisme de tâches tout est effectué à l'exécution, avec notamment la création dynamique de processus et leur allocation « à la volée » aux processeurs. Un autre aspect divergeant se situe au niveau de l'espace d'adressage utilisé par chacune des deux classes de langages. Avec le parallélisme de données, l'adressage est global alors qu'il est séparé pour le parallélisme de tâches.

Les projets Fx [112], Fortran-M [55] et Opus [26] ajoutent du contrôle et des communications dans un langage à parallélisme de données. Les deux premiers utilisent des directives explicites alors que le dernier est basé sur un système relativement coûteux d'appels de procédures à distance. Data$_{//}$-Orca [9] et Braid [118] ajoutent quant à eux des objets distribués sur la plate-forme d'exécution dans des langages à parallélisme de tâches. Le premier utilise des mécanismes se rapprochant de ceux employés par les Mémoires Virtuellement Partagées (MVP) alors que second

crée des classes C++ adaptées au parallélisme de données. D'autres langages existent qui utilisent des bibliothèques de passage de messages pour intégrer le parallélisme de tâches à des langages à parallélisme de données. COLT$_{HPF}$ [91] permet ainsi à des tâches HPF de communiquer et de se synchroniser au moyen d'appels à PVM, alors que HPF/MPI [57] utilise MPI.

Si aucun des langages présentés ne combine toutes les fonctionnalités de langages n'utilisant qu'une des deux formes de parallélisme, ils sont toutefois plus génériques, et ce sans complication inutile du modèle de programmation. Cependant, la définition d'un langage « idéal » intégrant les parallélismes de tâches et de données reste un challenge. C'est pourquoi une autre approche est utilisée. Celle-ci s'emploie à trouver le parallélisme de données dans des graphes de tâches. Une fois ces informations extraites, des bibliothèques numériques parallèles sont appelées concurremment, réalisant ainsi la combinaison recherchée.

C'est dans ce cadre que la notion de parallélisme mixte a été introduite par Chakrabarti, Demmel et Yelick dans [25]. Dans cet article, les auteurs présentent une méthode d'ordonnancement appelée « parallélisme commuté » (*switched parallelism*) consistant à placer des tâches soit sur un seul processeur, soit sur l'ensemble des processeurs de la plate-forme d'exécution. Cette méthode ne s'applique cependant qu'à certaines classes d'applications, notamment celles de type « diviser pour régner ». Ce type d'applications a été choisi car il exhibe les deux sortes de parallélisme (de tâches et de données) dans des proportions similaires. L'algorithme d'ordonnancement proposé classe tout d'abord l'ensemble des tâches sous la forme d'une liste et détermine ensuite un seuil pour lequel il est le plus intéressant d'utiliser le parallélisme de données avant, et le parallélisme de tâches après. Ceci s'explique par le fait que, dans les applications de type « diviser pour régner », la taille des tâches, et par conséquent leur potentialité à exhiber du parallélisme de données, décroît au cours l'avancée dans l'arbre de division. À l'inverse, le nombre de tâches, et donc la possibilité d'employer le parallélisme de tâches, augmente.

Au cours de cette thèse, nous nous sommes plus particulièrement intéressés aux travaux de trois groupes de chercheurs autour du parallélisme mixte et des techniques d'ordonnancement associées. Les premiers travaux que nous avons étudiés sont ceux de Ramaswamy *et al.* qui visent d'une part à intégrer le parallélisme mixte dans le compilateur HPF Paradigm [104] et d'autre part à extraire du parallélisme mixte de codes Matlab pour ensuite appeler la bibliothèque ScaLAPACK [102]. Ces travaux sont synthétisés dans la thèse de Ramaswamy [103]. Dans les deux cas, une première étape consiste à extraire de l'information des codes sources afin de déterminer où le parallélisme mixte est susceptible d'être le mieux utilisé. Pour cela, les auteurs introduisent la notion de *Macro Dataflow Graph* (MDG). Un MDG est un graphe acyclique direct où les nœuds représentent des calculs séquentiels ou parallèles et les arcs représentent les relations de précédence. Les nœuds sont pondérés par le temps d'exécution de la tâche, estimé par la loi d'Amdahl, avec les inconvénients que nous avons évoqués au chapitre précédent. Ces temps d'exécutions sont augmentés des temps de latences inhérentes au transfert des données en entrée et en sortie de la tâche. Ces latences sont déterminées en fonction des distributions source et destination des données, au moyen d'un tableau de correspondance entre diverses distributions unidimensionnelles issues de HPF (`ALL`, `BLOCK`, `CYCLIC` et `BLOCKCYCLIC(X)`). Le calcul de ces latences dépend alors du nombre de messages envoyés et de la quantité de données correspondante. Les distributions n-dimensionnelles sont quant à elles gérées par décomposition en une séquence de redistributions unidimensionnelles. Les arcs sont quand à eux pondérés par le temps de communication entre deux nœuds. Les latences étant intégrées aux temps de calculs, ces temps de communication sont estimés en divisant la quantité de données à transmettre par la bande passante du réseau. Deux nœuds de ce graphe sont distingués, l'un précédant et l'autre suivant tous les autres nœuds. Enfin, le MDG est une structure hiérarchique, chacun des nœuds pouvant être

également un MDG. Il existe quatre types de nœuds : simples, boucles, branchements conditionnels, et définis par l'utilisateur. Cependant, dans notre étude des travaux de Ramaswamy, nous nous sommes concentrés sur les MDG uniquement composés de nœuds simples.

L'algorithme d'ordonnancement proposé par Ramaswamy utilise la programmation convexe, rendue possible la propriété de posynomialité des modèles de coût choisis, ainsi que certaines propriétés du MDG. Ainsi, le chemin critique est défini comme étant le plus long chemin dans le MDG et le temps de complétion associé est donc minimal. L'autre métrique utilisée est l'aire moyenne, définie comme le produit *temps* $\times$ *nombre de processeurs* moyen du MDG. L'algorithme TSAS (*Two Step Allocation and Scheduling*) se décompose en deux étapes. La première cherche à minimiser le temps de complétion selon les deux métriques présentées ci-dessus. Cette étape permet de déterminer le placement de chacune des tâches du graphe. La seconde étape est basée sur un algorithme par liste pour ordonnancer les tâches ainsi placées.

Rădulescu *et al.* ont également proposé deux algorithmes d'ordonnancement mixte en deux étapes : CPR (*Critical Path Reduction*) [101] et CPA (*Critical Path and Area-based scheduling*) [100]. Tous deux sont basés sur la réduction du chemin critique de l'application. La principale différence entre CPR et CPA est que le processus d'allocation est complètement découplé de l'ordonnancement dans CPA. Dans l'étape d'allocation, ces deux algorithmes cherchent à déterminer le nombre de processeurs le plus approprié à l'exécution de chacune des tâches. Pour cela, ils allouent tout d'abord un seul processeur à chacune des tâches. Puis, pour chaque tâche, des processeurs supplémentaires sont ajoutés un par un, tant que le temps d'exécution de cette tâche décroît, conformément à la loi d'Amdahl, et que le nombre total de processeurs disponibles n'est pas atteint. Ici encore, le problème d'une telle utilisation de la loi d'Amdahl, que nous avons souligné au chapitre précédent, ne permet pas toujours de déterminer la configuration optimale pour l'exécution d'une tâche. La seconde étape ordonne les tâches ainsi placées selon un algorithme par liste, comme pour TSAS.

Rauber et Rünger [105] limitent quant à eux leur étude à des graphes construits par compositions séries et/ou parallèles. Dans le premier cas, une séquence d'opérations présentant des dépendances de données est placée sur l'ensemble des processeurs. Les tâches de cette séquence sont alors exécutées séquentiellement. Dans le second cas, l'ensemble des processeurs est divisé en un nombre optimal de sous-ensembles, déterminé par un algorithme glouton. Le critère d'optimalité de cet algorithme est la minimisation du temps de complétion de l'ensemble de tâches considéré. Les temps d'exécution des routines parallèles sont estimés par des formules dépendant des coûts de communication et des temps correspondants à des exécutions séquentielles.

3.3 Application aux algorithmes rapides de produit de matrices

3.3.1 Algorithmes de Strassen et Winograd

En 1969, Strassen a proposé dans [111] un algorithme pour le produit de matrices 2×2 ne nécessitant que 7 multiplications, la huitième étant remplacée par une série de dix-huit additions et soustractions. L'algorithme de décomposition de Strassen et le graphe de tâches correspondant sont présentés par la figure 3.1. Cet algorithme a donc une complexité asymptotique meilleure que celle de l'algorithme standard de produit de matrices. Il est également possible de calculer le produit $C = AB$, où les matrices A, B et C sont carrées et de taille M, en utilisant l'algorithme de Strassen. Il suffit pour cela de découper ces matrices en blocs de taille $M/2$ de la manière suivante :

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

Données : $Matrices\ A, B$

Phase 1

$T_1 = A_{11} + A_{22}$ $\quad T_6 = B_{11} + B_{22}$
$T_2 = A_{21} + A_{22}$ $\quad T_7 = B_{12} - B_{22}$
$T_3 = A_{11} + A_{12}$ $\quad T_8 = B_{21} - B_{11}$
$T_4 = A_{21} - A_{11}$ $\quad T_9 = B_{11} + B_{12}$
$T_5 = A_{12} - A_{22}$ $\quad T_{10} = B_{21} + B_{22}$

Phase 2

$Q_1 = T_1 * T_6$ $\quad Q_5 = T_3 * B_{22}$
$Q_2 = T_2 * B_{11}$ $\quad Q_6 = T_4 * T_9$
$Q_3 = A_{11} * T_7$ $\quad Q_7 = T_5 * T_{10}$
$Q_4 = A_{22} * T_8$

Phase 3

$U_1 = Q_1 + Q_4$ $\quad U_2 = Q_5 - Q_7$
$U_3 = Q_3 + Q_1$ $\quad U_4 = Q_2 - Q_6$
$C_{11} = U_1 - U_2$ $\quad C_{12} = Q_3 + Q_5$
$C_{21} = Q_2 + Q_4$ $\quad C_{22} = U_3 - U_4$

Résultat : $C = (C_{ij})$

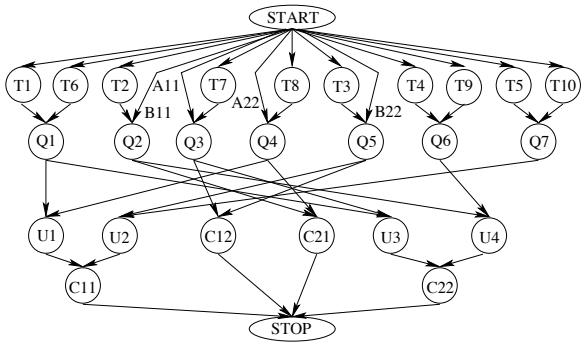


FIG. 3.1 – Décomposition de Strassen (gauche) et graphe de tâches associé (droite).

Si les sept produits internes de la décomposition de Strassen sont effectués en utilisant l'algorithme standard, le nombre total d'opérations arithmétiques calculées peut alors s'exprimer comme suit : $7(2(M/2)^3 - (M/2)^2) + 18(M/2)^2 = (7/4)M^3 + (11/4)M^2$. Si l'on considère maintenant le ratio entre cette complexité et celle de l'algorithme standard, nous obtenons :

$$Ratio = \frac{7M^3 + 11M^2}{8M^3 - 4M^2}.$$

Ce ratio tend vers $7/8$ lorsque M devient grand. Ceci implique que pour des matrices d'une taille suffisamment importante, il existe un gain théorique possible de $12,5\%$.

La décomposition de Strassen peut aussi être appliquée récursivement à des matrices de dimension $M = 2^k$ pour obtenir la complexité asymptotique de $O(M^{\log(7)}) = O(M^{2.807})$. Lorsque les matrices sont de taille quelconque, il faut se ramener au cas de matrices carrées dont la taille est une puissance de deux. Pour cela, il suffit de déterminer le maximum des dimensions de chacune des matrices, puis d'amener la taille des matrices à la première puissance de deux supérieure à ce maximum. Il ne reste plus alors qu'à compléter les matrices avec des éléments nuls.

La variante de Winograd de l'algorithme de Strassen, introduite dans [119], nécessite le même nombre de multiplications mais permet de réduire le nombre d'additions et soustractions effectuées de dix-huit à quinze. Cet algorithme et le graphe des tâches correspondant sont présentés dans la figure 3.2.

Il a été prouvé dans [93] que la décomposition de Winograd est minimale pour les algorithmes d'ordre sept. Il n'existe donc pas d'autre algorithme pour le produit de matrices 2×2 utilisant sept produits, au lieu des huit de l'algorithme standard, qui nécessite moins de quinze additions et/ou soustractions.

Ces deux algorithmes on été très étudiés sur des machines mono-processeur pour améliorer les performances d'applications numériques [5, 27, 62, 67, 114]. Ainsi dans [62], la décomposition

Données : $Matrices\, A, B$

Phase 1

$T_1 = A_{21} + A_{22}$ $\quad T_5 = B_{12} - B_{11}$
$T_2 = T_1 - A_{11}$ $\quad T_6 = B_{22} - T_5$
$T_3 = A_{11} - A_{21}$ $\quad T_7 = B_{22} - B_{12}$
$T_4 = A_{12} - T_2$ $\quad T_8 = B_{21} + T_6$

Phase 2

$Q_1 = A_{11} * B_{11}$ $\quad Q_5 = T_3 * T_7$
$Q_2 = A_{12} * B_{21}$ $\quad Q_6 = T_4 * B_{22}$
$Q_3 = T_1 * T_5$ $\quad Q_7 = A_{22} * T_8$
$Q_4 = T_2 * T_6$

Phase 3

$U_1 = Q_1 + Q_4$ $\quad U_2 = U_1 + Q_5$
$U_3 = U_1 + Q_3$
$C_{11} = Q_1 + Q_2$ $\quad C_{12} = U_3 + Q_6$
$C_{21} = U_2 + Q_7$ $\quad C_{22} = U_2 + Q_3$

Résultat : $C = (C_{ij})$

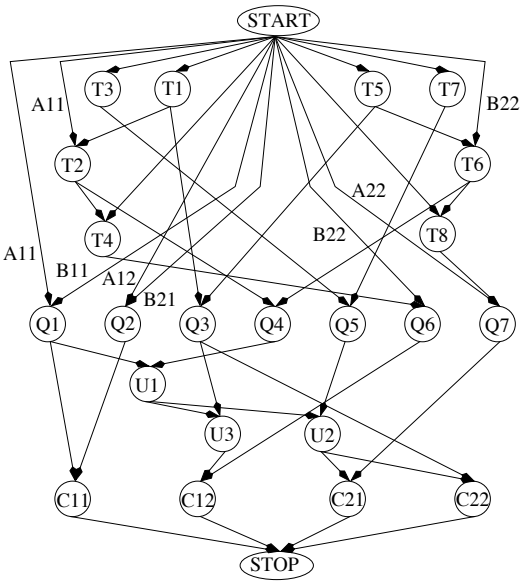


FIG. 3.2 – Décomposition de Winograd (gauche) et graphe de tâches associé (droite).

de Strassen est utilisée dans les opérations des BLAS de niveau 3. La stabilité numérique de l'algorithme est également étudiée dans cet article. En effet, si les méthodes de Strassen et Winograd réduisent le nombre d'opérations arithmétiques effectuées, l'introduction des additions augmente le risque de perte de stabilité. Dans [67], une étude précise des coûts et gains induits par l'utilisation de versions récursives ainsi que des implantations dans le cas de matrices de taille impaire sont présentées.

Plusieurs implantations parallèles des algorithmes de Strassen et Winograd ont également été proposées. [11] présente des implantations utilisant le parallélisme de données sur des machines SIMD. Dans [30], les auteurs présentent un algorithme basé sur une dérécursification de la décomposition visant à augmenter le nombre de produits à ordonnancer. Le placement de ces tâches sur un nombre de processeurs puissance de sept est combiné à une réplication des données lors de la distribution initiale pour diminuer le nombre des communications générées. L'implantation obtenue obtient de meilleures performances que l'algorithme classique par anneau. [27] discute de l'utilisation de différentes distributions de données pour le développement de versions en parallélisme de données d'algorithmes de produits de matrices standards et rapides. Dans tous ces articles, les implantations présentées utilisent exclusivement les algorithmes de Strassen ou de Winograd. Une autre approche consiste à combiner l'utilisation de ces algorithmes rapides et d'une implantation de l'algorithme standard. Deux choix orthogonaux sont alors possibles : développer un algorithme utilisant le parallélisme de données où les produits séquentiels sont effectués selon les décompositions de Strassen ou de Winograd ; il est également possible de développer une version parallèle des algorithmes rapides et d'utiliser un algorithme standard pour les produits de matrices internes. Dans [60], les auteurs justifient le choix de la seconde solution. L'argument avancé est que le gain apporté par les algorithmes rapides n'est intéressant que pour des matrices de grandes tailles. Par conséquent, l'utilisation de ces algorithmes au niveau le plus bas d'un algorithme parallèle limiterait de fait le gain potentiel. Les auteurs proposent donc un algorithme

basé sur des matrices de permutation pour obtenir un placement efficace des données sur des grilles carrées de processeurs. L'algorithme séquentiel standard utilisé dans cette implantation est SUMMA [116]. Une telle combinaison entre une implantation de l'algorithme standard et une variante de la décomposition de Winograd est utilisée dans [54] pour développer un algorithme ciblant des hyper-grilles de 7^k processeurs. Enfin, Ramaswamy [103] utilise la décomposition de Strassen comme code de test pour l'outil de parallélisation automatique Paradigm.

3.3.2 Cadre de travail

La motivation première de l'application du paradigme du parallélisme mixte aux algorithmes de Strassen et Winograd est de développer des algorithmes parallèles efficaces dans le cadre d'applications client–serveurs. Dans cette optique, nous avons réduit notre cadre de travail aux cas où les données à traiter ont déjà une distribution du fait de calculs précédents. Cela se justifie parfaitement puisque le produit de matrices est toujours utilisé comme noyau de calcul d'applications complexes. Plus particulièrement, nous nous intéressons ici au produit $C = AB$ de deux matrices distribuées sur deux grilles disjointes de processeurs. Ces deux grilles de processeurs – que nous dénommerons par la suite contextes – sont carrées, de dimension p et peuvent être considérées comme deux sous-ensembles d'une seule grille virtuelle rectangulaire de taille $p \times 2p$ (soit un total de $2p^2$ processeurs). Dans un but pédagogique, nous ne considérerons dans ce chapitre que des cas où p est une puissance de 2. Les implantations présentées ne sont bien entendu pas limitées à de telles tailles de grilles. Les processeurs de la grille globale sont numérotés par ligne de $[0, 0]$, dans le coin supérieur gauche, à $[p - 1, 2p - 1]$, dans le coin inférieur droit. Nous conserverons cette numérotation globale pour identifier les processeurs jusqu'à la fin de ce chapitre. En ce qui concerne les matrices impliquées dans ce calcul, elles sont carrées et de dimension $M = 2kp$, où $k \geq 1$. Enfin, nous imposons une contrainte à l'exécution du calcul, en l'occurrence l'obligation pour la matrice résultat C d'être alignée avec la matrice A à la fin de la multiplication.

Dans les deux paragraphes suivants, nous allons étudier des implantations des algorithmes de Strassen et Winograd développées suivant ces contraintes. Nous nous intéresserons tout d'abord à deux implantations utilisant le parallélisme de données, puis nous étudierons les différents choix qui nous conduits au développement de deux types d'implantations utilisant le parallélisme mixte. La première suit les phases des algorithmes originaux alors que la seconde peut être considérée comme le résultat d'un algorithme d'ordonnancement par liste.

3.3.3 Implantations utilisant le parallélisme de données

Étant donné que les matrices A et B sont distribuées sur des contextes disjoints, elles doivent être alignées avant de pouvoir appliquer la décomposition de Strassen (ou celle de Winograd). Cet alignement implique que deux redistributions ont lieu avant même le début de l'algorithme. De plus, étant donnée la contrainte de distribution imposée au résultat C, une troisième redistribution est nécessaire à la fin du calcul.

Les figures 3.3 et 3.4 présentent deux implantations utilisant le parallélisme de données qui sont axées sur la réduction du nombre de variables temporaires allouées. Pour chaque calcul, la variable de l'algorithme orignal correspondante est indiquée. Cette réduction s'opère en ordonnant l'exécution des tâches de manière à maximiser la réutilisation des variables. Nous avons développé l'implantation de l'algorithme de Strassen alors que l'implantation de l'algorithme de Winograd peut être trouvée dans [67].

Données : Matrices A et B
Variables temporaires : X, Y, Z, R_1, R_2, R_3

Stocker dans :		**Calcul**	**Variable Algorithmique**
		Redistribuer A dans X Redistribuer B dans Y	
R_1	$\leftarrow$	$X_{21} - X_{11}$	T_4
R_2	$\leftarrow$	$Y_{11} + Y_{12}$	T_9
R_3	$\leftarrow$	$R_1 * R_2$	Q_6
R_1	$\leftarrow$	$X_{21} + X_{22}$	T_2
R_2	$\leftarrow$	$R_1 * Y_{11}$	Q_2
Z_{22}	$\leftarrow$	$R_3 - R_2$	$-U_4$
R_1	$\leftarrow$	$Y_{21} - Y_{11}$	T_8
R_3	$\leftarrow$	$X_{22} * R_1$	Q_4
Z_{21}	$\leftarrow$	$R_2 + R_3$	Z_{21}
R_1	$\leftarrow$	$X_{11} + X_{22}$	T_1
R_2	$\leftarrow$	$Y_{11} + Y_{22}$	T_6
Z_{11}	$\leftarrow$	$R_1 * R_2$	Q_1
Z_{22}	$\leftarrow$	$Z_{22} + Z_{11}$	
Z_{11}	$\leftarrow$	$Z_{11} + R_3$	U_1
R_1	$\leftarrow$	$Y_{12} - Y_{22}$	T_7
R_2	$\leftarrow$	$X_{11} * R_1$	Q_3
Z_{22}	$\leftarrow$	$Z_{22} + R_2$	Z_{22}
R_1	$\leftarrow$	$X_{11} + X_{12}$	T_3
R_3	$\leftarrow$	$R_1 * Y_{22}$	Q_5
Z_{12}	$\leftarrow$	$R_2 + S_3$	Z_{12}
Z_{11}	$\leftarrow$	$Z_{11} - R_3$	
R_1	$\leftarrow$	$X_{12} - X_{22}$	T_5
R_2	$\leftarrow$	$Y_{21} + Y_{22}$	T_{10}
R_3	$\leftarrow$	$R_1 * R_2$	Q_7
Z_{11}	$\leftarrow$	$Z_{11} + R_3$	Z_{11}
		Redistribuer Z dans C	

Résultat : Matrice C

FIG. 3.3 – Implantation de l'algorithme de produit de matrices de Strassen utilisant le parallélisme de données.

Données : Matrices A et B
Variables temporaires : X, Y, Z, R_1, R_2

Stocker dans :		**Calcul**	**Variable Algorithmique**
		Redistribuer A dans X Redistribuer B dans Y	
R_1	$\leftarrow$	$X_{11} - X_{21}$	T_3
R_2	$\leftarrow$	$Y_{22} - Y_{12}$	T_7
Z_{11}	$\leftarrow$	$R_1 * R_2$	Q_5
R_1	$\leftarrow$	$X_{21} + X_{22}$	T_1
R_2	$\leftarrow$	$Y_{12} - Y_{11}$	T_5
Z_{22}	$\leftarrow$	$R_1 * R_2$	Q_3
R_1	$\leftarrow$	$R_1 - X_{11}$	T_2
R_2	$\leftarrow$	$Y_{22} - R_2$	T_6
Z_{21}	$\leftarrow$	$R_1 * R_2$	Q_4
R_1	$\leftarrow$	$X_{12} - R_1$	T_4
R_2	$\leftarrow$	$Y_{21} - R_2$	T_8
Z_{12}	$\leftarrow$	$R_1 * Y_{22}$	Q_6
Z_{12}	$\leftarrow$	$Z_{12} + Z_{22}$	
R_1	$\leftarrow$	$X_{11} * Y_{11}$	Q_1
Z_{21}	$\leftarrow$	$Z_{21} + R_1$	U_2
Z_{12}	$\leftarrow$	$Z_{21} * Z_{12}$	Z_{12}
Z_{21}	$\leftarrow$	$Z_{21} + Z_{11}$	U_3
Z_{11}	$\leftarrow$	$X_{22} * R_2$	Q_2
Z_{11}	$\leftarrow$	$Z_{11} + R_1$	Z_{11}
R_1	$\leftarrow$	$X_{21} * R_2$	Q_7
Z_{21}	$\leftarrow$	$Z_{21} + R_1$	Z_{21}
		Redistribuer Z dans C	

Résultat : Matrice C

FIG. 3.4 – Implantation de l'algorithme de produit de matrices de Winograd utilisant le parallélisme de données.

3.3.4 Implantations utilisant le parallélisme mixte

La stratégie choisie pour bénéficier de l'utilisation du parallélisme mixte dans les algorithmes de Strassen et Winograd consiste à conserver la distribution d'origine des matrices et à distribuer les tâches de calcul aux différents processeurs. Cette stratégie a été préférée à celle utilisée par les algorithmes utilisant le parallélisme de données consistant à aligner les matrices avant de les multiplier. Nous espérons ainsi réduire le volume de communication tout en équilibrant au mieux la charge entre les processeurs.

Nous avons souhaité utiliser des routines parallèles de la bibliothèque SCALAPACK pour effectuer les différentes tâches de calcul générées par la décomposition de Strassen. Nous avons choisi de baser nos travaux sur cette bibliothèque numérique pour son extensibilité et sa portabilité. Pour exploiter toute l'efficacité de telles routines, il est indispensable d'utiliser des distributions de données adaptées à ces noyaux d'algèbre linéaire. Les distributions candidates sont la distribution *par blocs* et la distribution *cyclique par blocs bidimensionnelle*. Afin de déterminer quelle distribution choisir, il est important de noter que les implantations parallèles des algorithmes de Strassen et Winograd ne peuvent être meilleures que des implantations de l'algorithme standard que si les additions peuvent être calculées sans introduire de communications. Or, si A (ou B) est distribuée selon une distribution par blocs, chacune des additions de la phase 1 de l'algorithme de Strassen (ou de Winograd) va générer des communications, comme le montre la figure 3.5 pour le calcul de $T_1 = A_{11} + A_{22}$. En effet, A_{11} étant possédée par le processeur $(0, 0)$ et A_{22} par le processeur $(1, 1)$, cette addition induit nécessairement une communication entre ces deux processeurs.

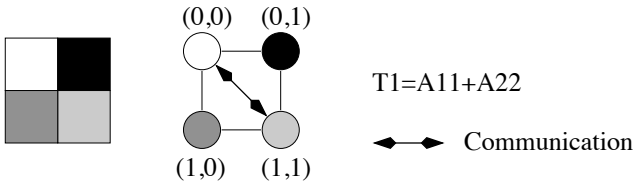


FIG. 3.5 – Exemple de communication induite par une distribution par blocs.

Ces communications ne peuvent être éliminées que si chaque processeur possède une partie de chacun des quatre quarts de la matrice. Une distribution cyclique par blocs bidimensionnelle permet d'obtenir une telle configuration, mais la taille des blocs de distribution doit être soigneusement choisie. En effet, une taille de bloc qui ne diviserait pas la taille des matrices ne permettrait pas d'obtenir l'alignement nécessaire au calcul des additions sans communications. De plus, une taille de bloc trop petite engendrerait un accroissement des accès mémoire. $R = M/2p$ est la plus grande taille de bloc permettant le calcul local de toutes les additions de l'algorithme. La figure 3.6 montre un exemple de la distribution choisie lorsque p est égal à 2. Pour chaque bloc, l'indice indique à quel quartier de matrice ce bloc appartient alors que l'exposant donne la position de ce bloc selon la distribution cyclique par blocs.

Ce choix de distribution nous assurant de l'exécution sans communications de toutes les additions des décompositions de Strassen et de Winograd, il nous reste à répartir les différentes tâches de calcul aussi équitablement que possible entre les deux contextes.

3.3.4.1 Implantations par phases

Les premières implantations des algorithmes de Strassen et Winograd que nous avons développées ont été tout d'abord présentées dans [41, 42] et améliorées dans [43]. L'idée prin-

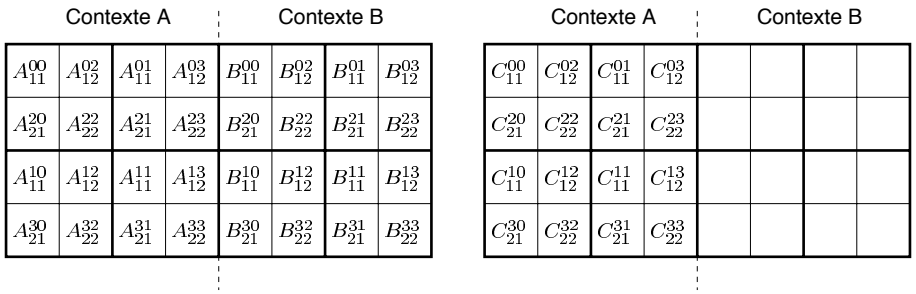


FIG. 3.6 – Distribution des matrices A, B et C sur une grille 2×4 de processeurs.

cipale des implantations de [42] était de conserver les différentes phases des algorithmes originaux, à savoir une phase d'addition, une phase de multiplication et enfin une autre phase d'addition. Ces tâches étant réparties sur des contextes disjoints, lorsque une donnée distribuée sur l'un des contextes est nécessaire à l'exécution d'une tâche sur l'autre contexte, cette donnée doit être déplacée. Pour cela, il suffit d'insérer, entre les phases de calcul, des phases de communications où seront effectués tous les déplacements de données indispensables à la poursuite de l'algorithme.

Afin de déterminer une bonne répartition des tâches entre les contextes, nous avons cherché à savoir quelles données étaient impliquées dans quelles tâches. Il est aisé de remarquer qu'aussi bien dans l'algorithme de Strassen que dans celui de Winograd, toutes les additions de la première phase impliquent exclusivement soit des données issues de A, soit des données provenant de B. Il semble judicieux d'effectuer ces tâches là où sont distribuées les données sur lesquelles elles portent, afin de préserver le principe de localité. À l'inverse, chacune des multiplications de la deuxième phase implique une donnée issue de A et une provenant de B. Un mouvement de donnée entre les contextes est donc nécessaire pour chaque multiplication. Mais le choix de répartition entre les contextes est par conséquent plus libre que pour les additions. Dans nos implantations, nous avons choisi d'effectuer quatre produits sur le contexte où doit être distribué le résultat et les trois produits restants sur l'autre contexte. Enfin, nous avons choisi d'exécuter les additions de la troisième phase là où doit être distribuée la matrice résultat à la fin du calcul. Ce choix, s'il diminue la quantité de parallélisme exploité, permet cependant de réduire le volume de communication lié à cette dernière phase. Or, pour une taille de matrice donnée, le coût d'une addition est bien moindre que celui d'une communication. Ces choix de répartition nous ont conduits à l'implantation mixte par phases de l'algorithme de Strassen présenté par la figure 3.7.

Dans l'implantation mixte par phases de l'algorithme de Winograd, nous avons cherché à utiliser les particularités de l'algorithme de base pour réduire le volume de communication généré par rapport à l'implantation de l'algorithme de Strassen. Il est possible d'obtenir cette réduction en effectuant du calcul redondant. En effet, si l'on exécute les calculs de Q_3 et Q_5 sur le contexte A, cela force le déplacement T_5 et B_{22} du contexte B vers le contexte A. Or, ces deux données sont les opérandes du calcul de T_6. Ce calcul peut donc être effectué sur les deux contextes. Grâce à ce calcul redondant, il est alors possible de calculer trois des sept produits internes en ne transférant que deux données au lieu de trois. Nous obtenons ainsi un gain, le coût d'une addition étant inférieur à celui d'une communication, pour une taille de matrice donnée. L'implantation résultante est donnée en figure 3.8.

Contexte A				Contexte B			
Donnée : Matrice A				**Donnée :** Matrice B			
Variables temporaires : R_1, R_2, R_3, R_4, R_5, R_6, R_7				Variables temporaires : S_1, S_2, S_3, S_4, S_5, S_6, S_7			
Stocker dans :		**Calcul**	**Variable Algorithmique**	**Stocker dans :**		**Calcul**	**Variable Algorithmique**
R_1	$\leftarrow$	$A_{11} + A_{22}$	T_1	S_1	$\leftarrow$	$B_{11} + B_{22}$	T_6
R_2	$\leftarrow$	$A_{21} + A_{22}$	T_2	S_2	$\leftarrow$	$B_{12} - B_{22}$	T_7
R_3	$\leftarrow$	$A_{11} + A_{12}$	T_3	S_3	$\leftarrow$	$B_{21} - B_{11}$	T_8
R_4	$\leftarrow$	$A_{21} - A_{11}$	T_4	S_4	$\leftarrow$	$B_{11} + B_{12}$	T_9
R_5	$\leftarrow$	$A_{12} - A_{22}$	T_5	S_5	$\leftarrow$	$B_{21} + B_{22}$	T_{10}
			Redistribuer R_6 dans S_2 Redistribuer R_1 dans S_2 Redistribuer R_1 dans S_3 Redistribuer R_2 dans S_3 Redistribuer R_2 dans S_4 Redistribuer R_3 dans S_4 Redistribuer R_3 dans S_5				
R_7	$\leftarrow$	$A_{11} * R_6$	Q_3	S_5	$\leftarrow$	$S_2 * S_1$	Q_1
R_6	$\leftarrow$	$A_{22} * R_1$	Q_4	S_6	$\leftarrow$	$S_3 * B_{11}$	Q_2
R_1	$\leftarrow$	$R_4 * R_2$	Q_6	S_7	$\leftarrow$	$S_4 * B_{22}$	Q_5
R_2	$\leftarrow$	$R_5 * R_3$	Q_7				
			Redistribuer R_3 dans S_5 Redistribuer R_4 dans S_6 Redistribuer R_5 dans S_7				
C_{11}	$\leftarrow$	$R_3 + R_6$					
C_{11}	$\leftarrow$	$C_{11} - R_5$					
C_{11}	$\leftarrow$	$C_{11} + R_2$	C_{11}				
C_{12}	$\leftarrow$	$R_7 + R_5$	C_{12}				
C_{21}	$\leftarrow$	$R_4 + R_6$	C_{21}				
C_{22}	$\leftarrow$	$R_3 + R_7$					
C_{22}	$\leftarrow$	$C_{22} - R_4$					
C_{22}	$\leftarrow$	$C_{22} + R_1$	C_{22}				
Résultat : $C = (C_{ij})$							

FIG. 3.7 – Implantation par phases de l'algorithme de produit de matrices de Strassen utilisant le parallélisme mixte.

Contexte A				Contexte B			
Donnée : Matrice A Variables temporaires : R_1, R_2, R_3, R_4, R_5, R_6, R_7				**Donnée :** Matrice B Variables temporaires : S_1, S_2, S_3, S_4, S_5, S_6			
Stocker dans :		**Calcul**	**Variable Algorithmique**	**Stocker dans :**		**Calcul**	**Variable Algorithmique**
R_1	$\leftarrow$	$A_{21} + A_{22}$	T_1	S_1	$\leftarrow$	$B_{12} - B_{11}$	T_5
R_2	$\leftarrow$	$R_1 - A_{11}$	T_2	S_2	$\leftarrow$	$B_{22} - S_1$	T_6
R_3	$\leftarrow$	$A_{11} - A_{21}$	T_3	S_3	$\leftarrow$	$B_{22} - B_{12}$	T_7
R_4	$\leftarrow$	$A_{12} - R_2$	T_4	S_4	$\leftarrow$	$B_{21} + S_2$	T_8
Redistribuer R_3 dans S_5 Redistribuer R_5 dans S_1 Redistribuer A_{11} dans S_6 Redistribuer R_6 dans B_{22} Redistribuer A_{22} dans S_1 Redistribuer R_7 dans B_{21}							
R_3	$\leftarrow$	$R_1 * R_5$	Q_3	S_2	$\leftarrow$	$S_6 * B_{11}$	Q_1
R_1	$\leftarrow$	$R_4 * R_6$	Q_6	S_6	$\leftarrow$	$S_5 * S_3$	Q_5
R_4	$\leftarrow$	$A_{12} * R_7$	Q_2	S_5	$\leftarrow$	$S_1 * S_4$	Q_7
R_6	$\leftarrow$	$R_6 - R_5$	T_6				
R_5	$\leftarrow$	$R_2 * R_6$	Q_4				
Redistribuer R_6 dans S_2							
C_{11}	$\leftarrow$	$R_6 + R_4$	C_{11}				
R_4	$\leftarrow$	$R_6 + R_5$	T_1				
Redistribuer R_2 dans S_6 Redistribuer R_5 dans S_5							
R_6	$\leftarrow$	$R_4 + R_2$	T_2				
C_{12}	$\leftarrow$	$R_4 + R_3$					
C_{12}	$\leftarrow$	$C_{12} + R_1$	C_{12}				
C_{21}	$\leftarrow$	$R_6 + R_5$	C_{21}				
C_{22}	$\leftarrow$	$R_6 + R_3$	C_{22}				
Résultat : $C = (C_{ij})$							

FIG. 3.8 – Implantation par phases de l'algorithme de produit de matrices de Winograd utilisant le parallélisme mixte.

3.3.4.2 Implantations par liste

Après avoir implanté ces versions mixtes par phases des algorithmes de Strassen et Winograd, nous avons cherché à développer des versions de manière plus automatique et en utilisant moins de variables temporaires. Pour cela, nous avons adopté le principe de réutilisation des variables employé par les algorithmes utilisant le parallélisme de données. Les implantations ainsi définies peuvent être considérées comme étant le résultat d'un algorithme d'ordonnancement par liste. En effet, l'ordre dans lequel les tâches sont exécutées suit certaines règles que nous allons détailler.

Une hypothèse majeure a été faite dans la conception des ces implantations. En effet, nous supposons que la répartition des multiplications entre les contextes est connue avant l'exécution de l'algorithme d'ordonnancement par liste. Pour déterminer cette répartition, nous avons étudié le schéma reliant les multiplications et les additions de la troisième phase. Ce schéma est construit de la manière suivante : il existe une relation de dépendance entre deux tâches si leurs résultats sont opérandes d'une même tâche. Dans le cas de l'algorithme de Strassen, la structure ainsi obtenue est une chaîne reliant les sept produits, comme nous pouvons le voir en figure 3.9. Cette figure indique également la répartition des multiplications que nous avons adoptée, à savoir le calcul de Q_6, Q_2, Q_4 et Q_1 sur le contexte A et le calcul de Q_3, Q_5 et Q_7 sur le contexte B.

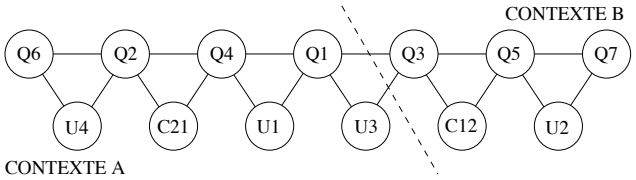


FIG. 3.9 – Chaîne de produits et répartition proposée pour l'algorithme de Strassen.

Cette répartition autorise le calcul de la plupart des additions de la troisième phase (calcul d'U_1, U_2, U_4, C_{12}, et de C_{21}) sans aucun mouvement de données. De plus, cela permet, du fait de la répartition choisie, l'exécution en parallèle de deux additions. Les additions restantes seront exécutées là où doit être distribuée la matrice résultat afin d'éviter des communications superflues.

Dans le cas de l'algorithme de Winograd, la structure obtenue n'est pas aussi simple, comme le montre la figure 3.10. Il est toutefois possible de déterminer une répartition satisfaisante des multiplications, à savoir le calcul de Q_2, Q_3, Q_6 et Q_7 sur le contexte A et le calcul de Q_1, Q_4 et Q_5 sur le contexte B. Comme pour Strassen, cette répartition permet l'exécution en parallèle de deux additions de la troisième phase.

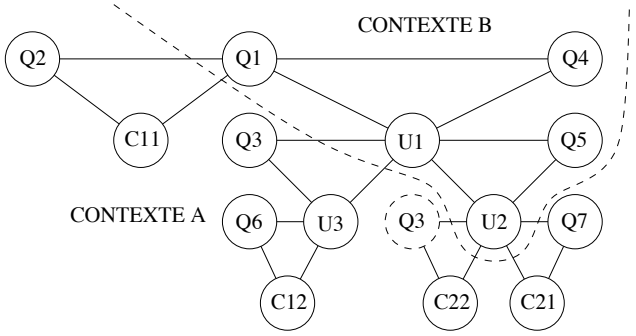


FIG. 3.10 – Structure reliant les produits et répartition proposée pour l'algorithme de Winograd.

Enfin, les additions de la première phase sont toujours exécutées là où sont distribuées les données qu'elles impliquent, et ce pour la même raison que celle évoquée dans la description des algorithmes par phases.

Nous connaissons désormais le placement de chacune des tâches de l'algorithme. Pour déterminer l'ordonnancement de nos algorithmes, nous utilisons une liste de tâches prêtes. Ces tâches sont de quatre types : addition, multiplication, envoi et réception de données. Les tâches de type *addition* et *multiplication* sont considérées prêtes dès que leurs deux opérandes ont été produits. En ce qui concerne les opérations de communication, les tâches sont prêtes dès qu'une donnée à transférer a été produite.

Le choix de la tâche prête à exécuter est dirigé par différents critères. Le plus important est d'équilibrer les calculs entre les contextes le plus longtemps possible. Ensuite, les communications sont prioritaires sur les autres tâches, afin de libérer l'espace mémoire occupé par les données à transférer. Enfin, une addition qui produit une tâche prête est prioritaire sur les autres. En suivant ces règles, nous obtenons les algorithmes des figures 3.11 pour Strassen et 3.12 pour Winograd.

3.3.5 Analyse et comparaison théorique

Afin d'évaluer l'impact de l'utilisation du parallélisme mixte sur des implantations parallèles, nous avons effectué une analyse théorique de nos algorithmes et les avons comparées aux implantations utilisant le parallélisme de données du point de vue de l'occupation mémoire et du temps d'exécution.

3.3.5.1 Allocation de variables temporaires et utilisation de la mémoire

Les algorithmes de Strassen et Winograd produisent nombre de valeurs intermédiaires (21 pour Strassen et 18 pour Winograd) qui doivent être stockées. Si l'on considère des implantations de ces algorithmes dans le cas plus général où les données à multiplier, de taille M, sont déjà alignées sur une grille $p \times 2p$ de processeurs, les variables temporaires nécessaires à ce stockage seront toutes de dimension $M/2p \times M/4p$. En utilisant une politique naïve d'allocation des variables temporaires, l'algorithme de Strassen nécessiterait alors $21M^2/8p^2$ éléments temporaires sur chaque processeur, et celui de Winograd demanderait $9M^2/4p^2$ éléments. Nous allons maintenant étudier et comparer les différentes implantations présentées précédemment du seul point de vue de leur gestion des temporaires.

Dans les implantations utilisant le parallélisme de données présentées par les figures 3.3 et 3.4, l'alignement des matrices A et B implique l'allocation de deux variables de dimension $M/p \times M/2p$. De plus, la redistribution de la matrice résultat C, à la fin du calcul, implique l'allocation d'une variable supplémentaire de la même taille. En plus de ces variables qui ne dépendent pas du calcul en lui-même, l'implantation en parallélisme de données de l'algorithme de Strassen nécessite trois variables temporaires supplémentaires, chacune étant de taille $M/2p \times M/4p$. Cette implantation alloue donc $15M^2/8p^2$ éléments temporaires par processeur. En ce qui concerne la variante de Winograd, elle ne nécessite que deux variables supplémentaires de taille $M/2p \times M/4p$ soit un total de $7M^2/4p^2$ éléments temporaires par processeur.

Dans [42], nous avions proposé des implantations mixtes des algorithmes de Strassen et Winograd allouant $2(M/p)^2$ éléments temporaires sur chacun des processeurs du contexte A et $3(M/p)^2$ éléments sur les processeurs du contexte B. Ces implantations consomment donc légèrement plus de mémoire que les implantations qui utilisent le parallélisme de données. Cependant,

Contexte A				Contexte B			
Donnée : Matrice A Variables temporaires : R_1, R_2, R_3, R_4				**Donnée :** Matrice B Variables temporaires : S_1, S_2, S_3, S_4, S_5			
Stocker dans :		**Calcul**	**Variable Algorithmique**	**Stocker dans :**		**Calcul**	**Variable Algorithmique**
		Redistribuer A_{11} dans S_1 Redistribuer R_1 dans B_{11}					
R_2	$\leftarrow$	$A_{21} + A_{22}$	T_2	S_2	$\leftarrow$	$B_{12} - B_{22}$	T_7
R_3	$\leftarrow$	$R_2 * R_1$	Q_2	S_3	$\leftarrow$	$S_1 * S_2$	Q_3
R_1	$\leftarrow$	$A_{11} + A_{12}$	T_3	S_1	$\leftarrow$	$B_{21} - B_{11}$	T_8
		Redistribuer R_2 dans S_1 Redistribuer R_1 dans S_1					
R_1	$\leftarrow$	$A_{22} * R_2$	Q_4	S_2	$\leftarrow$	$S_1 * B_{22}$	Q_5
C_{21}	$\leftarrow$	$R_1 + R_3$	C_{21}	S_1	$\leftarrow$	$S_2 + S_3$	C_{12}
		Redistribuer C_{12} dans S_1					
R_2	$\leftarrow$	$A_{21} - A_{11}$	T_4	S_1	$\leftarrow$	$B_{21} + B_{22}$	T_{10}
R_4	$\leftarrow$	$A_{12} - A_{22}$	T_5	S_4	$\leftarrow$	$B_{11} + B_{12}$	T_9
		Redistribuer R_4 dans S_5 Redistribuer R_4 dans S_4					
C_{22}	$\leftarrow$	$R_2 * R_4$	Q_6	S_4	$\leftarrow$	$S_5 * S_1$	Q_7
C_{22}	$\leftarrow$	$C_{22} - R_3$	$-U_4$	S_1	$\leftarrow$	$S_4 - S_2$	$-U_2$
		Redistribuer C_{11} dans S_1					
R_2	$\leftarrow$	$A_{11} + A_{22}$	T_1	S_4	$\leftarrow$	$B_{11} + B_{22}$	T_6
		Redistribuer R_3 dans S_3					
C_{22}	$\leftarrow$	$C_{22} + R_3$					
		Redistribuer R_3 dans S_4					
R_4	$\leftarrow$	$R_2 * R_3$	Q_1				
C_{11}	$\leftarrow$	$C_{11} + R_4$					
C_{11}	$\leftarrow$	$C_{11} + R_1$	C_{11}				
C_{22}	$\leftarrow$	$C_{22} + R_4$	C_{22}				
Résultat : $C = (C_{ij})$							

FIG. 3.11 – Implantation par liste de l'algorithme de produit de matrices de Strassen utilisant le parallélisme mixte.

Contexte A				Contexte B			
Donnée : Matrice A Variables temporaires : R_1, R_2, R_3				**Donnée :** Matrice B Variables temporaires : S_1, S_2, S_3, S_4, S_5			
Stocker dans :		**Calcul**	**Variable Algorithmique**	**Stocker dans :**		**Calcul**	**Variable Algorithmique**
		Redistribuer A_{11} dans S_1 Redistribuer R_1 dans B_{21}					
C_{11}	$\leftarrow$	$A_{12} * R_1$	Q_2	S_2	$\leftarrow$	$S_1 * B_{11}$	Q_1
R_1	$\leftarrow$	$A_{11} + A_{22}$	T_1	S_1	$\leftarrow$	$B_{12} - B_{11}$	T_5
R_2	$\leftarrow$	$A_{11} - A_{21}$	T_3	S_3	$\leftarrow$	$B_{22} - B_{12}$	T_7
		Redistribuer R_3 dans S_1 Redistribuer R_2 dans S_4					
C_{22}	$\leftarrow$	$R_1 * R_3$	Q_3	S_5	$\leftarrow$	$S_4 * S_3$	Q_5
R_2	$\leftarrow$	$R_1 - A_{11}$	T_2	S_3	$\leftarrow$	$B_{22} - S_1$	T_6
R_1	$\leftarrow$	$A_{12} - R_2$	T_4	S_1	$\leftarrow$	$B_{21} + S_3$	T_8
		Redistribuer R_2 dans S_4 Redistribuer R_2 dans S_1					
C_{21}	$\leftarrow$	$A_{22} * R_2$	Q_7	S_1	$\leftarrow$	$S_4 * S_3$	Q_4
		Redistribuer R_2 dans B_{22}					
C_{12}	$\leftarrow$	$R_1 * R_2$	Q_6	S_1	$\leftarrow$	$S_1 + S_2$	U_1
				S_5	$\leftarrow$	$S_5 + S_1$	U_2
		Redistribuer R_1 dans S_2					
C_{11}	$\leftarrow$	$C_{11} + R_1$	C_{11}				
		Redistribuer R_1 dans S_1					
C_{12}	$\leftarrow$	$C_{12} + R_1$					
C_{12}	$\leftarrow$	$C_{12} + C_{22}$	C_{12}				
		Redistribuer R_1 dans S_5					
C_{21}	$\leftarrow$	$C_{21} + R_1$	C_{21}				
C_{22}	$\leftarrow$	$C_{22} + R_1$	C_{22}				
Résultat : $C = (C_{ij})$							

FIG. 3.12 – Implantation par liste de l'algorithme de produit de matrices de Winograd utilisant le parallélisme mixte.

les améliorations apportées dans les implantations mixtes présentées précédemment permettent de réduire le nombre de variables temporaires déclarées et donc de réduire la consommation mémoire. Cette réduction provient principalement d'un changement dans la politique de communication entre les différentes implantations. Dans [42], la stratégie employée consistait à grouper les données à envoyer afin de réduire le nombre de latences de communication. Pour réaliser ce groupage des messages, nous utilisions des variables temporaires d'une taille plus importante (M/p au lieu de $M/2p$) afin de stocker les quatre quarts au sein d'une seule matrice. Cette idée de grouper les messages a été abandonnée au profit de celle visant à réutiliser au maximum les variables temporaires. Il est important de noter que cette réutilisation intervient non seulement durant les phases de calcul, notamment pour les implantations par liste, mais aussi pendant les phases de communication, surtout pour les implantations par phases. En utilisant cette politique de réutilisation au niveau des phases de communication, nos implantations mixtes par phases nécessitent autant d'éléments temporaires par processeur que l'implantation de l'algorithme de Winograd utilisant le parallélisme de données.

Comme nous l'avons expliqué précédemment, l'ordonnancement par liste permet une meilleure réutilisation des variables temporaires. L'implantation mixte par liste de l'algorithme de Strassen alloue donc quatre temporaires sur les processeurs du contexte A et cinq sur ceux du contexte B. Concernant la variante de Winograd, l'implantation mixte par liste ne nécessite plus que trois variables sur les processeurs du contexte A et toujours cinq sur ceux du contexte B. Pour ces deux implantations, chacune de ces variables temporaires est de taille $M/2p$.

Nous souhaitons évaluer la quantité maximale de mémoire allouée par les différentes implantations, en comptabilisant les données initiales et le résultat. La figure 3.13 montre l'espace mémoire nécessaire aux différentes versions lorsque les contextes sont de dimension deux et que la taille des matrices varie.

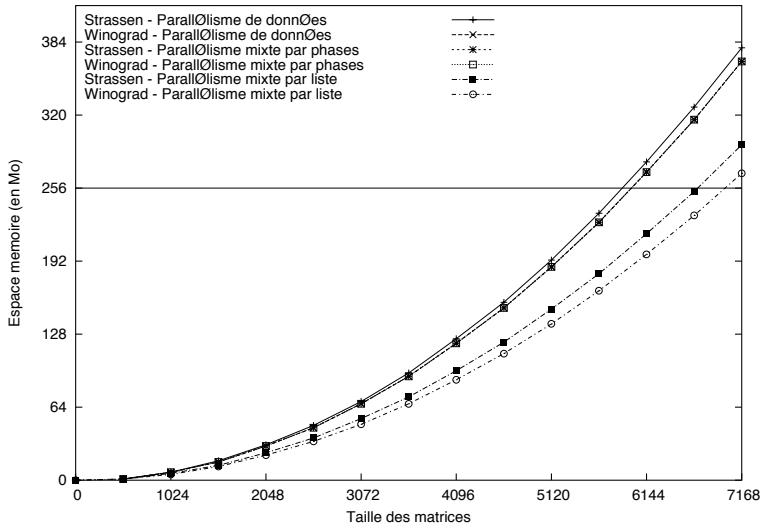


FIG. 3.13 – Comparaison des espaces mémoire nécessaires pour des contextes de dimension deux.

Sur cette figure, nous pouvons voir que l'implantation de l'algorithme de Strassen utilisant le parallélisme de données est celle qui consomme le plus d'espace mémoire. Comme nous l'avons indiqué précédemment, les implantations mixtes par phases nécessitent autant de mémoire que

l'implantation en parallélisme de données de l'algorithme de Winograd. Enfin nous trouvons les implantations mixtes par liste, celle de l'algorithme de Winograd allouant le moins de mémoire.

Si l'on suppose que la quantité de mémoire disponible par processeur est limitée, par exemple à 256 Mo sur la figure 3.13, nous pouvons affirmer que l'utilisation du parallélisme mixte permet potentiellement de traiter des matrices plus grandes qu'en employant le parallélisme de données. Il est à noter que ce gain provient des conditions particulières de notre cadre de travail. En effet, si aucune contrainte de distribution n'était appliquée aux matrices données et résultat, les variables temporaires relatives aux alignements n'auraient plus lieu d'être. Les implantations utilisant le parallélisme de données consommeraient alors moins de mémoire que celles employant le parallélisme mixte. Pratiquement, il n'est pas cependant possible d'atteindre cette limite théorique, les routines internes de calcul et communication allouant également des variables temporaires.

Nous allons maintenant proposer des modèles théoriques des différentes implantations en nous basant sur les techniques présentées au chapitre précédent. Cela nous permettra de comparer leurs temps d'exécution respectifs et donc leur efficacité.

3.3.5.2 Modélisation des temps d'exécution

Dans les algorithmes de produit de matrices de Strassen et Winograd, il n'existe que deux sortes d'opérations de base différentes : l'addition et la multiplication de matrices. Dans toutes les implantations présentées dans ce chapitre, les additions sont effectuées en utilisant une version modifiée d'une routine interne à la bibliothèque PBLAS. Cette routine permet de vectoriser la procédure d'addition tout en effectuant du déroulement de boucle. Notre modification a seulement consisté à autoriser l'écriture du résultat de l'addition dans une troisième variable. Les multiplications sont quant à elles calculées au moyen de la routine `pdgemm` de la bibliothèque ScaLAPACK. Nous ne tiendrons pas compte des effets de cache dans nos modèles, la taille de bloc de distribution choisie étant suffisamment grande pour ne pas subir ces effets.

À ces deux opérations de calcul s'ajoutent deux opérations de communication : la redistribution et la communication entre contextes. La redistribution est utilisée dans le cas des implantations utilisant le parallélisme de données, lors de l'alignement des matrices et du rapatriement du résultat sur le contexte A. Les communications entre contextes sont quant à elles employées dans les implantations utilisant le parallélisme mixte pour déplacer les données manquantes nécessaires à la continuation du calcul.

Nous allons maintenant présenter ou rappeler les modèles de coût pour chacune de ces opérations de base.

Redistribution Dans le cas des implantations utilisant le parallélisme de données, l'opération de redistribution permet non seulement d'aligner les matrices mais aussi de pouvoir encore calculer les additions localement après l'alignement. Pour cela, la taille des blocs de distribution est modifiée au cours de la redistribution pour devenir $R' = R/2$. La figure 3.14 montre, par exemple, la nouvelle distribution de la matrice A lorsque p est égal à 2. De même que dans la figure 3.6, l'indice indique, pour chaque bloc, à quel quartier de matrice ce bloc appartient alors que l'exposant donne la position de ce bloc selon la distribution cyclique par blocs.

Pour effectuer ce type de redistribution, nous avons utilisé la routine de redistribution de la bibliothèque ScaLAPACK [97]. Cette routine est basé sur l'algorithme de la chenille qui est efficace pour la plupart des schémas de communication. Son principe est le suivant : les processeurs sources et destinations forment une seule liste de processeurs ; chaque processeur calcule le schéma de communication, *i.e.*, la quantité de données qu'il a à échanger avec n'importe lequel des autres processeurs impliqués dans la redistribution, lui y compris. Une fois

Contexte A				Contexte B			
A_{11}^{00}	A_{12}^{04}	A_{11}^{01}	A_{12}^{05}	A_{11}^{02}	A_{12}^{06}	A_{11}^{03}	A_{12}^{07}
A_{11}^{20}	A_{12}^{24}	A_{11}^{21}	A_{12}^{25}	A_{11}^{22}	A_{12}^{26}	A_{11}^{23}	A_{12}^{27}
A_{21}^{40}	A_{22}^{44}	A_{21}^{41}	A_{22}^{45}	A_{21}^{42}	A_{22}^{46}	A_{21}^{43}	A_{22}^{47}
A_{21}^{60}	A_{22}^{64}	A_{21}^{61}	A_{22}^{65}	A_{21}^{62}	A_{22}^{66}	A_{21}^{63}	A_{22}^{67}
A_{11}^{10}	A_{12}^{14}	A_{11}^{11}	A_{12}^{15}	A_{11}^{12}	A_{12}^{16}	A_{11}^{13}	A_{12}^{17}
A_{11}^{30}	A_{12}^{34}	A_{11}^{31}	A_{12}^{35}	A_{11}^{32}	A_{12}^{36}	A_{11}^{33}	A_{12}^{37}
A_{21}^{50}	A_{22}^{54}	A_{21}^{51}	A_{22}^{55}	A_{21}^{52}	A_{22}^{56}	A_{21}^{53}	A_{22}^{57}
A_{21}^{70}	A_{22}^{74}	A_{21}^{71}	A_{22}^{75}	A_{21}^{72}	A_{22}^{76}	A_{21}^{73}	A_{22}^{77}

FIG. 3.14 – Distribution de la matrice A sur une grille 2×4 de processeurs après alignement.

ce schéma calculé, chaque processeur a un *rendez-vous* avec chacun des autres processeurs pour échanger des données. La figure 3.15 montre comment une liste de huit processeurs roule telle une chenille pour assurer que toutes les communications sont effectuées. Durant les étapes paires, deux processeurs se communiquent des données à eux même. Ils effectuent en réalité des copies mémoire des données.

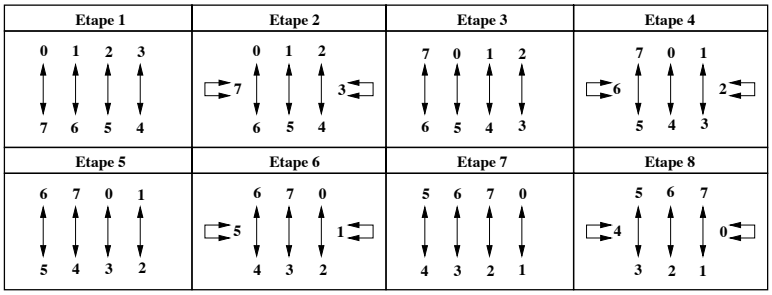


FIG. 3.15 – Schéma de communication de l'algorithme de la chenille.

Pour estimer le coût des redistributions effectuées dans les implantations utilisant le parallélisme de données que nous avons présentées, nous avons tout d'abord déterminé le schéma de communication. Dans le cas particulier où nous nous trouvons, chaque processeur du contexte A (resp. B), soit p^2 processeurs émetteurs, envoie quatre blocs de taille $M/4p \times M/4p$ à quatre autres processeurs. La matrice entière se trouve ainsi communiquée. Pour simplifier notre analyse, nous supposons qu'il n'y a que des communications unilatérales dans chacune des étapes de l'algorithme de la chenille. En effet, un échange de données impliquerait deux communications séquentielles. Cette supposition nous conduit au modèle suivant pour l'alignement d'une matrice distribuée sur une grille $p \times p$ de processeurs vers une grille $p \times 2p$ de processeurs :

$$T_{Alignement} = M^2\tau + 2p^2\lambda. \tag{3.1}$$

Communication entre contextes Dans les implantations utilisant le parallélisme mixte, lorsqu'une donnée nécessaire à un calcul est manquante, on effectue une communication de cette donnée d'un contexte à l'autre. Dans toutes nos implantations, nous avons utilisé des fonctions bloquantes d'envoi et de réception pour effectuer ces communications. Étant donné que les contextes de calcul ont la même taille et la même forme, une propriété intéressante du schéma de communication peut être utilisée. En effet, le processeur $[MaLigne, MaColonne]$ du contexte A n'échangera des données qu'avec le processeur $[MaLigne, MaColonne + p]$ du contexte B, et ce quelles que soient les valeurs de $MaLigne$ et $MaColonne$. Cette propriété permet de communiquer toute un quart de matrice de taille $M/2$ entre deux contextes de taille p en une seule communication modélisée par :

$$T_{Comm} = \left(\frac{M}{2p}\right)^2 \tau + \lambda. \tag{3.2}$$

Multiplication de matrices Le modèle de la routine `pdgemm` que nous allons employer pour cette étude est celui que nous avons présenté dans le chapitre précédent dans l'équation 2.3 et dont une version adaptée au cas où les données à multiplier sont des quarts de matrices de dimension $M/2$ est donnée dans l'équation 3.3 pour la multiplication sur un contexte et par l'équation 3.4 pour la multiplication sur l'ensemble des processeurs.

$$T_{Mult}^{contexte} = \left\lceil \frac{M}{2R} \right\rceil \times \texttt{temps\_dgemm\_contexte} + \frac{M^2}{2}\tau_p^p + \left\lceil \frac{M}{R} \right\rceil \lambda_p^p. \tag{3.3}$$

Dans ce cas, les matrices passées en paramètres de l'appel FAST permettant de calculer `temps_dgemm_contexte` sont de tailles $\lceil M/2p \rceil \times R$ pour le premier opérande et $R \times \lceil M/2p \rceil$ pour le second.

$$T_{Mult}^{tous} = \left\lceil \frac{M}{2R'} \right\rceil \times \texttt{temps\_dgemm\_tous} + \frac{M^2}{4}\left(\tau_p^{2p} + \tau_{2p}^p\right) + \left\lceil \frac{M}{2R'} \right\rceil \left(\lambda_p^{2p} + \lambda_{2p}^p\right). \tag{3.4}$$

Ici, les matrices passées en paramètres de l'appel FAST permettant de calculer `temps_dgemm_tous` sont de tailles $\lceil M/2p \rceil \times R$ pour le premier opérande et $R \times \lceil M/4p \rceil$ pour le second.

Addition de matrices En raison de la distribution des données choisie, toutes les additions peuvent être exécutées localement sans communications, comme nous l'avons expliqué au paragraphe 3.3.4. Dans le cas du calcul d'une addition sur un contexte, le coût `temps_addition_contexte` peut être obtenu par l'appel FAST suivant :

```
fast_comp_time (hôte, add_desc, &temps_addition_contexte)
```

où les deux quarts de matrice passés en argument dans `add_desc` sont de taille $\lceil M/2p \rceil \times \lceil M/2p \rceil$. Lorsque l'addition est effectuée sur l'ensemble des processeurs, `temps_addition_tous` peut être estimé par l'appel FAST suivant :

```
fast_comp_time (hôte, add_desc, &temps_addition_tous)
```

où les deux quarts de matrice passés en argument dans `add_desc` sont de taille $\lceil M/2p \rceil \times \lceil M/4p \rceil$.

Nous savons désormais estimer chacune des différentes composantes des modèles des implantations des algorithmes de Strassen et Winograd présentées dans ce chapitre. Nous allons

maintenant étudier plus précisément chaque implantation en commençant par celles utilisant le parallélisme de données.

Dans ces implantations, la seule différence entre les algorithmes de Strassen et de Winograd se situe au niveau du nombre d'additions calculées. Si les trois redistributions et les sept produits internes sont communs aux deux algorithmes, l'algorithme de Strassen nécessite dix-huit additions et celui de Winograd seulement quinze. Si l'on somme les coûts des différentes composantes, nous obtenons les modèles suivants :

$$\begin{aligned} T_{Strassen} &= 14p \times \texttt{temps\_dgemm\_tous} + 18 \times \texttt{temps\_addition\_tous} \\ &\quad + \frac{7M^2}{4}\left(\tau_p^{2p} + \tau_{2p}^{p}\right) + 3M^2\tau + 14p\left(\lambda_p^{2p} + \lambda_{2p}^{p}\right) + 6p^2\lambda. \end{aligned} \tag{3.5}$$

$$\begin{aligned} T_{Winograd} &= 14p \times \texttt{temps\_dgemm\_tous} + 15 \times \texttt{temps\_addition\_tous} \\ &\quad + \frac{7M^2}{4}\left(\tau_p^{2p} + \tau_{2p}^{p}\right) + 3M^2\tau + 14p\left(\lambda_p^{2p} + \lambda_{2p}^{p}\right) + 6p^2\lambda. \end{aligned} \tag{3.6}$$

En ce qui concerne les implantations utilisant le parallélisme mixte, il est nécessaire de déterminer le chemin critique de chaque implantation pour construire le modèle correspondant. Le seul point commun à toutes les versions est le nombre de produits se trouvant dans le chemin critique, à savoir quatre multiplications effectuées sur le contexte A et portant sur des quarts de matrices. Ces produits peuvent être modélisés par l'équation suivante :

$$\begin{aligned} T_{Mult}^{Mixte} &= 4 \times T_{Mult}^{contexte} \\ &= 4p \times \texttt{temps\_dgemm\_contexte} + 2M^2\tau_p^p + 8p\lambda_p^p. \end{aligned} \tag{3.7}$$

Du point de vue des communications, toutes nos implantations mixtes effectuent dix déplacements de données, sauf l'implantation par phases de l'algorithme de Winograd qui n'en nécessite que neuf. Cette réduction du volume de communication est due au calcul redondant de T_6 que nous avons expliqué précédemment. Enfin, le nombre d'additions calculées varie en fonction de l'implantation choisie. Les implantations par phases des algorithmes de Strassen et Winograd effectuent respectivement treize et douze additions, l'implantation par liste de Strassen seulement onze et l'implantation par liste de Winograd plus que neuf. Nous obtenons ainsi les quatre modèles suivants pour nos implantations mixtes des algorithmes de Strassen et Winograd :

$$\begin{aligned} T_{Strassen}^{Phase} &= 4p \times \texttt{temps\_dgemm\_contexte} + 13 \times \texttt{temps\_addition\_contexte} \\ &\quad + 2M^2\tau_p^p + + \frac{5M^2}{2p^2}\tau + 8p\lambda_p^p + 10\lambda, \end{aligned} \tag{3.8}$$

$$\begin{aligned} T_{Winograd}^{Phase} &= 4p \times \texttt{temps\_dgemm\_contexte} + 12 \times \texttt{temps\_addition\_contexte} \\ &\quad + 2M^2\tau_p^p + \frac{9M^2}{4p^2}\tau + 8p\lambda_p^p + 9\lambda, \end{aligned} \tag{3.9}$$

$$T^{Liste}_{Strassen} = 4p \times \texttt{temps\_dgemm\_contexte} + 11 \times \texttt{temps\_addition\_contexte} + 2M^2\tau^p_p + + \frac{5M^2}{2p^2}\tau + 8p\lambda^p_p + 10\lambda, \quad (3.10)$$

$$T^{Liste}_{Winograd} = 4p \times \texttt{temps\_dgemm\_contexte} + 9 \times \texttt{temps\_addition\_contexte} + 2M^2\tau^p_p + + \frac{5M^2}{2p^2}\tau + 8p\lambda^p_p + 10\lambda. \quad (3.11)$$

En supposant que l'on utilise une diffusion par arbre dans nos implantations, les valeurs de τ^p_p, τ^p_{2p}, τ^{2p}_p, λ^p_p, λ^p_{2p}, et λ^{2p}_p peuvent être réécrites uniquement en fonction de τ et *lambda* dans les modèles de communications. Ces valeurs sont respectivement $\lceil \log_2(p) \rceil * \tau/p$, $(\lceil \log_2(p) \rceil + 1) * \tau/p$, $(\lceil \log_2(p) \rceil + 1) * \tau/p$, $\lceil \log_2(p) \rceil \lambda$, $(\lceil \log_2(p) \rceil + 1) * \lambda$, et $(\lceil \log_2(p) \rceil + 1) * \lambda$. Le tableau 3.1 présente ces nouveaux coûts de communication.

	Communication	
Implantation	$M^2\tau$	λ
Strassen Données	$3 + 7(2\lceil \log_2(p) \rceil + 1)/4p$	$14p(2\lceil \log_2(p) \rceil + 1) + 6p^2$
Strassen Phases	$5/2p^2 + 2\lceil \log_2(p) \rceil /p$	$8p\lceil \log_2(p) \rceil + 10$
Strassen Liste	$5/2p^2 + 2\lceil \log_2(p) \rceil /p$	$8p\lceil \log_2(p) \rceil + 10$
Winograd Données	$3 + 7(2\lceil \log_2(p) \rceil + 1)/4p$	$14p(2\lceil \log_2(p) \rceil + 1) + 6p^2$
Winograd Phases	$9/4p^2 + 2\lceil \log_2(p) \rceil /p$	$8p\lceil \log_2(p) \rceil + 9$
Winograd Liste	$5/2p^2 + 2\lceil \log_2(p) \rceil /p$	$8p\lceil \log_2(p) \rceil + 10$

TAB. 3.1 – Résumé des coûts de communication en fonction de τ et λ.

Les implantations des algorithmes de Strassen et Winograd utilisant le parallélisme de données ont approximativement le même modèle théorique puisque la seule différence entre ces deux algorithmes est le nombre d'additions effectuées (dix-huit contre quinze). La complexité des additions étant en $O(M^2)$, cette différence est négligeable au regard du reste du modèle. Pour la même raison, nous pouvons dire que les implantations de l'algorithme de Strassen utilisant le parallélisme mixte et l'implantation par liste de l'algorithme de Winograd ont des modèles de coût très comparables. En ce qui concerne l'implantation par phases de l'algorithme de Winograd, la différence en termes de volume de communication due au calcul redondant peut devenir importante en fonction du ratio entre puissance de calcul et capacités de communication.

La figure 3.16 montre les instanciations de quelques modèles en utilisant des valeurs expérimentales pour les temps d'addition et de multiplication, ainsi que pour τ et λ. Ces valeurs correspondent à une grappe de Pentium III connectée par un réseau Fast Ethernet. Nous pouvons voir qu'une implantation utilisant le parallélisme mixte permet d'obtenir de meilleures performances lorsque l'on souhaite multiplier des matrices distribuées sur des grilles disjointes. Ce gain peut être supérieur à 45% sur une grappe telle que celle simulée dans la figure 3.16.

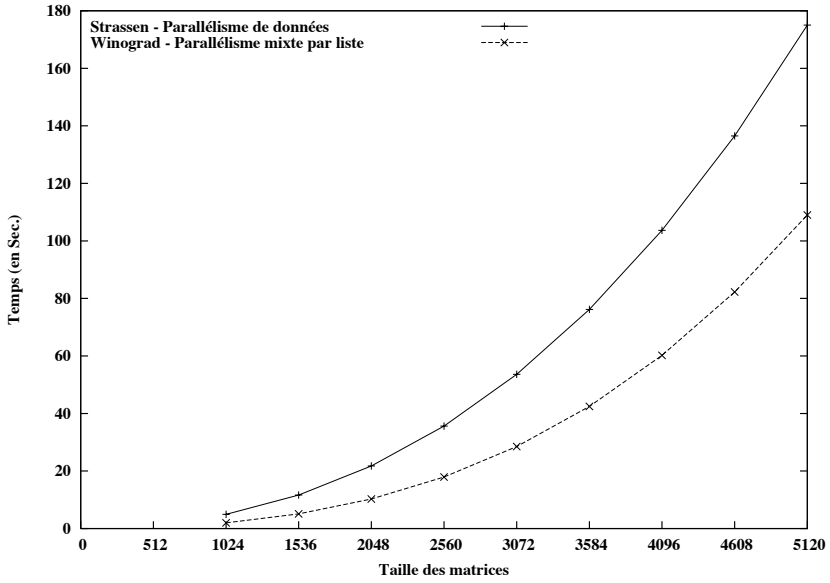


FIG. 3.16 – Performances théoriques de l'implantation en parallélisme de données de l'algorithme de Strassen, et de l'implantation mixte par liste de l'algorithme de Winograd.

3.3.6 Étude d'une version récursive

La complexité asymptotique en $O(M^{2.807})$ des algorithmes de Strassen et Winograd ne peut être atteinte que si la décomposition est appliquée récursivement. Des versions récursives de nos implantations utilisant le parallélisme mixte sont relativement faciles à mettre en œuvre. En effet, chacun des sept produits internes implique une matrice distribuée sur le contexte A et une distribuée sur le contexte B. Une partie des hypothèses faites pour le développement de nos implantations est donc vérifiée. L'autre hypothèse majeure suppose que la taille des blocs de distribution permette de calculer les additions sans communications. Pour que cette hypothèse reste vraie jusqu'au niveau de décomposition le plus profond, la taille de bloc doit être adaptée. $R = M/2^i p$, où i est le nombre de pas de récursion, est une taille de bloc permettant de satisfaire cette condition. Enfin, si l'on souhaite conserver la même répartition des tâches dans chaque appel récursif que dans la version non récursive, nous devons au préalable développer une version de chaque implantation où le résultat est distribué sur le contexte B. La figure 3.17 présente la répartition des produits permettant de dériver la version de l'implantation par liste de l'algorithme de Strassen, donnée par la figure 3.18, où le résultat est distribué sur le contexte B.

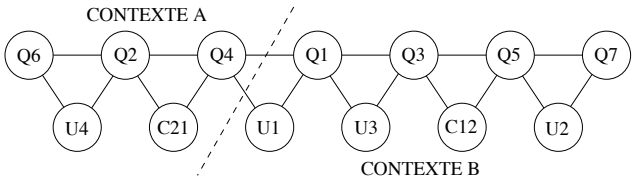


FIG. 3.17 – Chaîne de produits et répartition proposée pour la version de l'algorithme de Strassen où le résultat est distribué sur le contexte B.

Contexte A				Contexte B			
Donnée : Matrice A Variables temporaires : R_1, R_2, R_3, R_4, R_5				**Donnée :** Matrice B Variables temporaires : S_1, S_2, S_3, S_4			
Stocker dans :		**Calcul**	**Variable Algorithmique**	**Stocker dans :**		**Calcul**	**Variable Algorithmique**
			Redistribuer A_{11} dans S_1 Redistribuer R_1 dans B_{11}				
R_2	$\leftarrow$	$A_{21} + A_{22}$	T_2	S_2	$\leftarrow$	$B_{12} - B_{22}$	T_7
R_3	$\leftarrow$	$R_2 * R_1$	Q_2	S_3	$\leftarrow$	$S_1 * S_2$	Q_3
R_1	$\leftarrow$	$A_{11} + A_{12}$	T_3	S_1	$\leftarrow$	$B_{21} - B_{11}$	T_8
			Redistribuer R_2 dans S_1 Redistribuer R_1 dans S_1				
R_1	$\leftarrow$	$A_{22} * R_2$	Q_4	S_2	$\leftarrow$	$S_1 * B_{22}$	Q_5
R_2	$\leftarrow$	$R_1 + R_3$	C_{21}	C_{12}	$\leftarrow$	$S_2 + S_3$	C_{12}
			Redistribuer R_2 dans C_{21}				
R_2	$\leftarrow$	$A_{21} - A_{11}$	T_4	S_1	$\leftarrow$	$B_{21} + B_{22}$	T_{10}
R_4	$\leftarrow$	$A_{12} - A_{22}$	T_5	S_4	$\leftarrow$	$B_{11} + B_{12}$	T_9
			Redistribuer R_5 dans S_4 Redistribuer R_4 dans S_4				
R_4	$\leftarrow$	$R_2 * R_5$	Q_6	C_{11}	$\leftarrow$	$S_4 * S_1$	Q_7
R_4	$\leftarrow$	$R_4 - R_3$	$-U_4$	C_{11}	$\leftarrow$	$C_{11} - S_2$	$-U_2$
			Redistribuer R_4 dans C_{22}				
R_2	$\leftarrow$	$A_{11} + A_{22}$	T_1	S_4	$\leftarrow$	$B_{11} + B_{22}$	T_6
			Redistribuer R_1 dans S_1				
				C_{11}	$\leftarrow$	$C_{11} + S_1$	
			Redistribuer R_1 dans S_1				
				S_2	$\leftarrow$	$S_1 * S_4$	Q_1
				C_{11}	$\leftarrow$	$C_{11} + S_2$	C_{11}
				C_{22}	$\leftarrow$	$C_{22} + S_3$	
				C_{22}	$\leftarrow$	$C_{22} + S_2$	C_{22}
				Résultat : $C = (C_{ij})$			

FIG. 3.18 – Implantation par liste de l'algorithme de produit de matrices de Strassen utilisant le parallélisme mixte où le résultat est distribué sur le contexte B.

Mais si une implantation récursive utilisant le parallélisme mixte réduit le nombre de produits effectués, elle introduit une quantité non négligeable de communications. Sur une plate-forme composée de processeurs rapides connectés par un réseau lent, le gain en termes de calcul ne permet pas de compenser le coût de cette augmentation du volume de communication. Les performances que l'on peut obtenir sont donc moins bonnes qu'en utilisant une implantation non récursive.

3.3.7 Étude d'une implantation mixte hétérogène

Il existe dans la littérature quelques travaux de développement de noyaux numériques ciblant des plates-formes hétérogènes. Dans [7, 8], les auteurs ont prouvé la NP-complétude du problème de la distribution de données dans le cas d'un produit de matrices standard exécuté sur une plate-forme où les processeurs ont des vitesses différentes. Une heuristique polynômiale basée sur une distribution des matrices par blocs de colonnes est également présentée dans ces articles. Les algorithmes découlant de cette heuristique sont très efficaces par rapport à l'exécution d'une implantation homogène, comme celle de la bibliothèque SCALAPACK, sur le même type de plate-forme hétérogène. Cependant, la distribution utilisée est très irrégulière puisqu'elle est déterminée en fonction des différentes vitesses des processeurs. Cela implique des coûts de redistribution très élevés lorsque l'on se trouve dans une configuration comparable à celle de notre cadre de travail, où les données ont déjà une certaine distribution.

Dans [32], les auteurs proposent des implantations de routines BLAS et SCALAPACK s'adaptant de variations de charge, aussi bien au niveau des processeurs qu'à celui du réseau. Cet outil permet de modifier la taille des blocs de calcul en fonction des conditions du système au moment de l'exécution. Ici, la distribution des données reste régulière mais une telle modification de la taille des blocs peut ne pas être possible dans notre cadre d'étude où les données ont déjà une distribution. De plus, cette optimisation permet certes d'améliorer les performances des routines étudiées, mais n'est pas très tolérante à des différences importantes entre les vitesses des processeurs.

Dans le cadre de cette étude, nous supposons que la plate-forme hétérogène sur laquelle nous avons à effectuer notre produit de matrices est en fait la connexion de deux grappes homogènes de machines. L'homogénéité à l'intérieur de chacune de ces deux grappes s'entend aussi bien du point de vue de la vitesse des processeurs que de celui des connexions réseau. Nous supposons également qu'un processeur de l'une des deux grappes est capable d'effectuer un calcul plus vite qu'un processeur de l'autre grappe. Enfin, nous n'émettons aucune hypothèse quant au rapport de vitesse entre les réseaux.

Pour équilibrer efficacement la charge de calcul entre deux grilles de processeurs de puissances différentes, il faut s'intéresser aux tâches dont la complexité est la plus élevée, donc aux multiplications. Nous avons choisi de dériver deux versions hétérogènes. La première correspond au cas où le contexte A est plus puissant que le contexte B et la seconde pour le cas inverse. Dans les deux cas, le résultat doit toujours être distribué sur le contexte A à la fin du calcul. Nous nous sommes pour cela basés sur l'implantation mixte par liste de l'algorithme de Strassen. Si cette implantation n'est pas la meilleure de celles nous avons précédemment présentées, la structure de chaîne reliant les sept produits internes permet de développer facilement une version hétérogène. En effet, pour obtenir une bonne répartition, il suffit de déplacer la ligne de séparation le long de la chaîne de produits. Ainsi, nous souhaitons obtenir un déséquilibre de répartition des opérations les plus coûteuses qui correspond au ratio de puissance entre les deux contextes. Sur la figure 3.19 (haut), on peut voir que si le contexte A est plus puissant que le contexte B, on lui attribuera environ

deux fois plus de produits (cinq contre deux). Réciproquement, si le contexte B est plus puissant, la figure 3.19 (bas) montre la répartition à adopter afin d'équilibrer au mieux la charge de calcul.

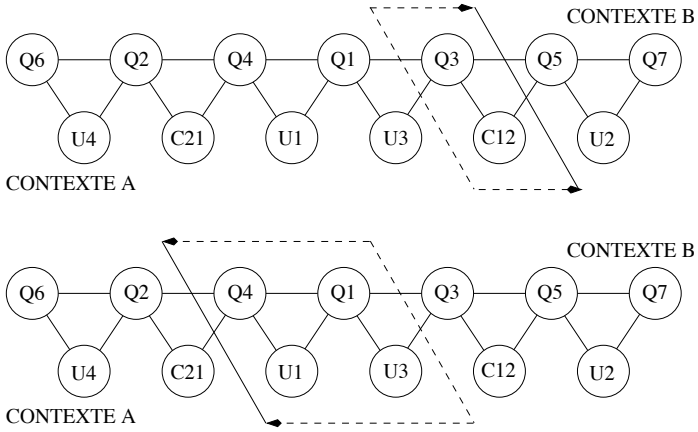


FIG. 3.19 – Propositions de répartitions de charge pour des implantations mixtes hétérogènes de l'algorithme de Strassen où le contexte A (resp. B) est le plus puissant (haut)(resp. bas).

En ce qui concerne les additions de la troisième phase, la répartition proposée par la figure 3.19 (haut) crée un déséquilibre plus important que celui provoqué par l'hétérogénéité de puissance des processeurs. On peut penser qu'essayer d'équilibrer également les additions pourrait améliorer la performance de l'implantation. En effet, s'il on effectue le calcul de C_{12} sur le contexte B, on obtient une répartition « quatre contre deux » des additions. Cependant, l'exécution de cette addition sur le contexte B engendre un déplacement de donnée supplémentaire (celui du résultat de l'addition). Or, comme nous l'avons déjà indiqué précédemment, le coût d'une communication est supérieur à celui d'une addition pour taille de matrice donnée. Cela est d'autant plus vrai dans le cas hétérogène où la vitesse de la communication sera limitée par le réseau le plus lent alors que l'addition sera effectuée sur le contexte le plus rapide, si l'on suit la répartition de la figure 3.19 (haut). C'est pour la même raison que dans la répartition de la figure 3.19 (bas), le calcul de C_{21} est placé sur le contexte A.

Les figures 3.20 et 3.21 montrent les implantations obtenues en appliquant notre algorithme d'ordonnancement par liste aux placements définis par la figure 3.19.

Il est important de noter que, par exemple pour la figure 3.20, l'ordonnancement essaye au maximum de faire en sorte que l'exécution d'une tâche de calcul sur le contexte B corresponde à l'exécution de deux tâches de même type sur le contexte A. Nous espérons ainsi perdre le moins de temps possible lors des synchronisations implicites dues aux communications.

3.3.8 Validation expérimentale

Afin de valider expérimentalement l'impact du parallélisme mixte sur des implantations parallèles des algorithmes de Strassen et Winograd, nous avons effectué différents tests sur une plate-forme homogène et sur une plate-forme hétérogène.

Notre plate-forme homogène est la grappe de PC *i-cluster* et notre plate-forme hétérogène est la connexion de deux grappes de PC homogènes situées au LIP. Les caractéristiques de ces deux machines ont été détaillées au chapitre 1. Dans toutes les expérimentations, nous avons utilisé,

Contexte A				Contexte B			
Donnée : Matrice A Variables temporaires : R_1, R_2, R_3, R_4, R_5, R_6, R_7				**Donnée :** Matrice B Variables temporaires : S_1, S_2, S_3, S_4			
Stocker dans :		**Calcul**	**Variable Algorithmique**	**Stocker dans :**		**Calcul**	**Variable Algorithmique**
			Redistribuer R_1 dans B_{11}				
R_2	$\leftarrow$	$A_{11} + A_{22}$	T_1	S_1	$\leftarrow$	$B_{11} + B_{22}$	T_6
R_3	$\leftarrow$	$A_{21} + A_{22}$	T_2				
			Redistribuer R_4 dans S_1				
R_5	$\leftarrow$	$A_{11} + A_{12}$	T_3	S_1	$\leftarrow$	$B_{12} - B_{22}$	T_7
R_6	$\leftarrow$	$A_{21} - A_{11}$	T_4				
			Redistribuer R_5 dans S_2 Redistribuer R_5 dans S_1				
R_7	$\leftarrow$	$R_2 * R_4$	Q_1	S_1	$\leftarrow$	$S_2 * B_{22}$	Q_5
R_2	$\leftarrow$	$A_{11} * R_5$	Q_3				
C_{22}	$\leftarrow$	$R_7 + R_2$	U_3	S_2	$\leftarrow$	$B_{21} - B_{11}$	T_8
R_4	$\leftarrow$	$A_{12} - A_{22}$	T_5				
			Redistribuer R_4 dans S_3 Redistribuer R_4 dans S_2				
				S_2	$\leftarrow$	$B_{21} + B_{22}$	T_{10}
R_5	$\leftarrow$	$R_3 * R_1$	Q_2	S_4	$\leftarrow$	$S_3 * S_2$	Q_7
R_1	$\leftarrow$	$A_{22} * R_4$	Q_4				
C_{21}	$\leftarrow$	$R_5 + R_1$	C_{21}	S_2	$\leftarrow$	$B_{11} + B_{12}$	T_9
C_{11}	$\leftarrow$	$R_7 + R_1$	U_1				
			Redistribuer R_1 dans S_2				
R_7	$\leftarrow$	$R_6 * R_1$	Q_6	S_2	$\leftarrow$	$S_4 - S_1$	$-U_2$
			Redistribuer R_1 dans S_1 Redistribuer R_3 dans S_2				
C_{22}	$\leftarrow$	$C_{22} - R_5$					
C_{22}	$\leftarrow$	$C_{22} + R_7$	C_{22}				
C_{12}	$\leftarrow$	$R_2 + R_1$	C_{12}				
C_{11}	$\leftarrow$	$C_{11} + R_3$	C_{11}				
Résultat : $C = (C_{ij})$							

FIG. 3.20 – Implantation mixte de l'algorithme de Strassen ciblant une plate-forme hétérogène où le contexte A est deux fois plus puissant que le contexte B.

Contexte A				Contexte B			
Donnée : Matrice A Variables temporaires : R_1, R_2, R_3, R_4				**Donnée :** Matrice B Variables temporaires : S_1, S_2, S_3, S_4, S_5, S_6, S_7			
Stocker dans :		**Calcul**	**Variable Algorithmique**	**Stocker dans :**		**Calcul**	**Variable Algorithmique**
			Redistribuer R_1 dans B_{11} Redistribuer A_{11} dans S_1 Redistribuer A_{22} dans S_2				
R_2	$\leftarrow$	$A_{21} + A_{22}$	T_2	S_3	$\leftarrow$	$B_{12} - B_{22}$	T_7
				S_4	$\leftarrow$	$B_{21} - B_{11}$	T_8
R_3	$\leftarrow$	$R_2 * R_1$	Q_2	S_5	$\leftarrow$	$S_1 * S_3$	Q_3
				S_1	$\leftarrow$	$S_2 * S_4$	Q_4
R_1	$\leftarrow$	$A_{11} + A_{22}$	T_1	S_2	$\leftarrow$	$B_{11} + B_{22}$	T_6
				S_3	$\leftarrow$	$B_{11} + B_{12}$	T_9
			Redistribuer R_1 dans S_4				
R_1	$\leftarrow$	$A_{11} + A_{12}$	T_3	R_6	$\leftarrow$	$S_4 * S_2$	Q_1
R_2	$\leftarrow$	$A_{21} - A_{11}$	T_4				
R_4	$\leftarrow$	$A_{12} - A_{22}$	T_5	S_2	$\leftarrow$	$B_{21} + B_{22}$	T_{10}
			Redistribuer R_1 dans S_4 Redistribuer S_3 dans R_1 Redistribuer R_4 dans S_3				
R_4	$\leftarrow$	$R_2 * R_1$	Q_6	S_7	$\leftarrow$	$S_4 * B_{22}$	Q_5
				S_4	$\leftarrow$	$S_3 * S_2$	Q_7
R_4	$\leftarrow$	$R_4 - R_3$	$-U_4$	S_2	$\leftarrow$	$S_5 + S_6$	U_3
				S_4	$\leftarrow$	$S_4 - S_7$	$-U_2$
			Redistribuer S_2 dans C_{22}				
C_{22}	$\leftarrow$	$C_{22} + R_4$	C_{22}	S_7	$\leftarrow$	$S_7 + S_5$	C_{12}
				S_6	$\leftarrow$	$S_6 + S_1$	U_1
			Redistribuer S_1 dans C_{21} Redistribuer S_7 dans C_{12}				
C_{21}	$\leftarrow$	$C_{21} + R_3$	C_{21}	S_6	$\leftarrow$	$S_6 + S_4$	C_{11}
			Redistribuer S_6 dans C_{11}				
Résultat : $C = (C_{ij})$							

FIG. 3.21 – Implantation mixte de l'algorithme de Strassen ciblant une plate-forme hétérogène où le contexte B est deux fois plus puissant que le contexte A.

pour l'ensemble des communications, des implantations de la bibliothèque BLACS [52] au dessus de MPI (LAM sur *i-cluster* et MPICH sur la plate-forme hétérogène). La bibliothèque PBLAS [29], dans sa version 1.0, a été utilisée pour effectuer les tâches générées par les décompositions de Strassen et Winograd. Le noyau séquentiel BLAS utilisé par cette bibliothèque est celui généré par ATLAS [49] et est par conséquent parfaitement adapté à chacune des architectures utilisées.

Nous allons à présent détailler les diverses expériences effectuées dans le but de comparer les performances relatives des implantations présentées dans ce chapitre.

3.3.8.1 Implantations homogènes

Nous avons exécuté chacune des implantations des algorithmes de Strassen et Winograd, celles utilisant le parallélisme de données et nos implantations mixtes, sur 8 puis sur 32 processeurs de l'*i-cluster*. Les figures 3.22 et 3.23 montrent les temps d'exécutions obtenus sur ces configurations.

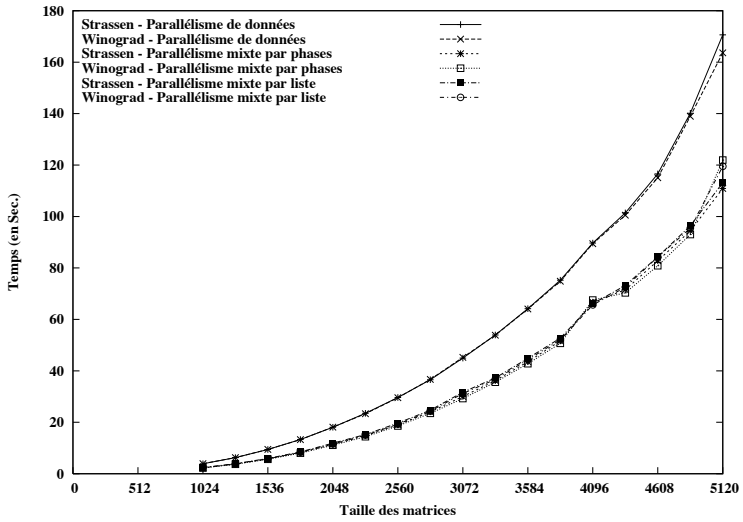


FIG. 3.22 – Comparaison des temps d'exécution des différentes implantations homogènes des algorithmes rapides de produit de matrices sur 8 processeurs de l'*i-cluster*.

Sur 8 processeurs, les temps d'exécution obtenus expérimentalement correspondent aux temps d'exécution théoriques de la figure 3.16. Ce résultat corrobore donc notre analyse théorique. Nous pouvons voir que les implantations utilisant le parallélisme de données ont des temps d'exécution supérieurs à celles utilisant le parallélisme mixte. Les différentes versions que nous avons développées ont toutes des temps d'exécution très proches, leur différence portant principalement sur le nombre d'additions. Cependant, il est à noter que si les implantations mixtes par phases sont théoriquement moins efficaces que les implantations par liste, le résultat inverse est observé expérimentalement. Ceci peut s'expliquer par une meilleure synchronisation des processeurs dans les versions par phases, les communications étant regroupées.

Les courbes obtenues sur 32 processeurs présentent les mêmes caractéristiques que celles obtenues sur 8 processeurs. Là encore, les implantations mixtes s'exécutent plus rapidement que les implantations utilisant le parallélisme de données. Il est également à noter qu'en utilisant quatre fois plus de processeurs, nous parvenons à traiter des matrices quatre fois plus grandes. Nos implantations sont donc extensibles en termes de consommation mémoire.

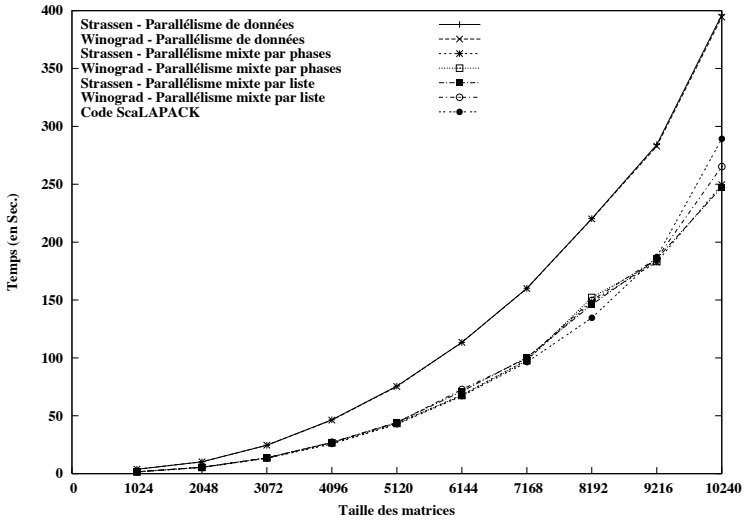


FIG. 3.23 – Comparaison des temps d'exécution des différentes implantations homogènes des algorithmes rapides de produit de matrices sur 32 processeurs de l'*i-cluster*.

Nous avons également comparé nos implantations à un code uniquement composé d'appels à des routines de la bibliothèque SCALAPACK. Ce code réalise l'alignement de A et B, le produit sur l'ensemble des processeurs et la redistribution de la matrice résultat. Les performances obtenues sont très proches de celles de nos implantations mixtes. Mais comme nous l'avons présenté dans le paragraphe 3.3.7 et comme nous allons le vérifier par les expériences menées sur la plate-forme hétérogène, les implantations utilisant le parallélisme mixte peuvent être efficacement adaptées au cas hétérogène, contrairement aux codes n'utilisant que des routines SCALAPACK.

3.3.8.2 Implantations hétérogènes

Pour valider la version hétérogène de notre implantation mixte par liste de l'algorithme de Strassen, nous avons réalisé des tests sur deux plates-formes hétérogènes. Nous avons tout d'abord simulé un déséquilibre entre les puissances de calcul des deux contextes sur l'*i-cluster* de manière à ce que les processeurs du contexte A calculent plus rapidement que ceux du contexte B. Pour arriver à ce résultat, les processeurs du contexte B exécutent chaque tâche de calcul plusieurs fois. Le nombre d'exécutions consécutives détermine le ratio de puissance entre les contextes. Il doit être noté que dans cette simulation de plate-forme hétérogène, le réseau reste homogène. La figure 3.24 présente les résultats de cette expérience sous la forme du gain obtenu par la version hétérogène par rapport à la version homogène correspondante exécutée sur la même plate-forme. Nous pouvons voir que ce gain augmente lorsque nous accroissons le ratio de puissance, pouvant atteindre environ 27%. Bien entendu, la version hétérogène s'avère moins efficace lorsque tous les processeurs sont homogènes.

Nous avons ensuite exécuté les implantations mixtes homogène et hétérogène de l'algorithme de Strassen, ainsi qu'un code composé uniquement d'appels à des routines SCALAPACK, sur la plate-forme hétérogène décrite au début du paragraphe 3.3.8. La figure 3.25 montre le gain obtenu par l'implantation hétérogène par rapport aux deux autres codes exécutés. Ce gain est clair, et ce même pour des petites matrices. De plus, ces résultats obtenus sur une vraie plate-

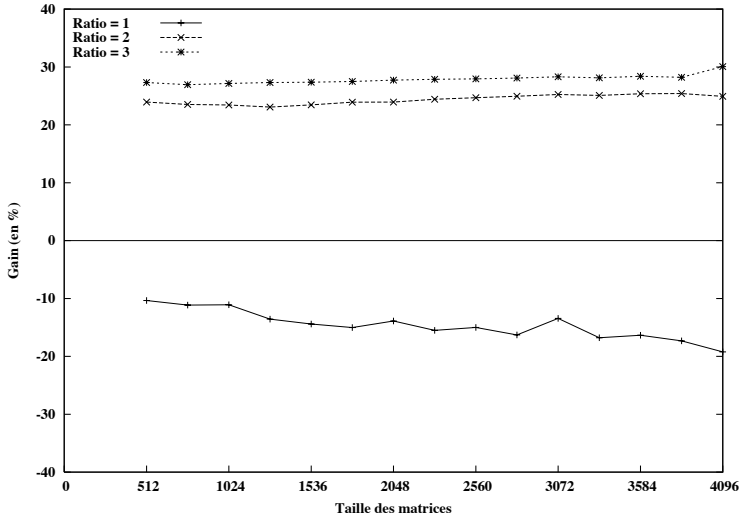


FIG. 3.24 – Gain de l'implantation mixte hétérogène de l'algorithme de Strassen par rapport à la version homogène correspondante sur une plate-forme hétérogène simulée avec différents ratios de puissance.

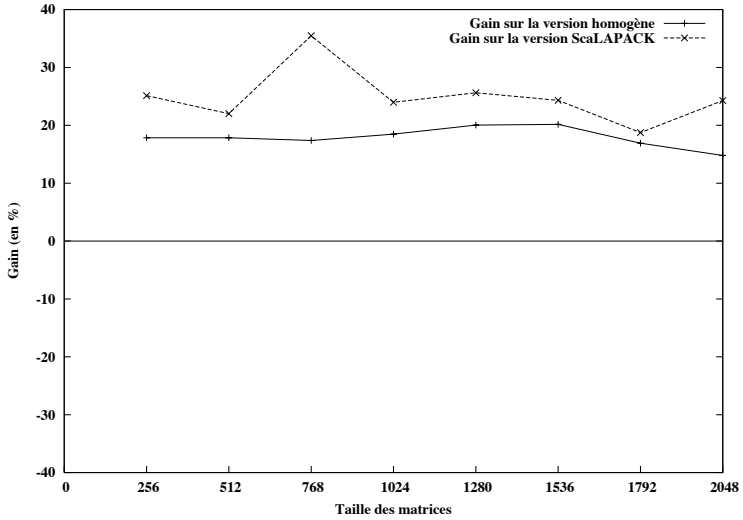


FIG. 3.25 – Gain de l'implantation mixte hétérogène de l'algorithme de Strassen par rapport à la version homogène correspondante et à un code SCALAPACK sur une plate-forme véritablement hétérogène.

forme hétérogène sont proches de ceux mesurés sur une plate-forme simulée. La différence peut provenir de l'impact du réseau sur les communications. En effet, si le ratio de puissance entre les deux contextes de calcul est relativement similaire à ceux simulés dans l'expérience précédente, il faut noter que sur cette plate-forme véritablement hétérogène, la vitesse du réseau n'est plus homogène.

3.3.9 Conclusion

Les différents travaux et expériences que nous avons menés autour de l'application du parallélisme mixte aux algorithmes de produit de matrices de Strassen et Winograd nous ont permis de montrer que cette approche permettait d'obtenir de meilleures performances qu'en utilisant uniquement le parallélisme de données. En effet, dans le cas où les données sont déjà distribuées, mais non alignées, l'utilisation du parallélisme mixte permet de passer outre la phase d'alignement et par conséquent de réduire le volume de communication de l'application.

Nous avons également montré que dans le cadre d'une exécution sur une plate-forme hétérogène, il est possible de dériver simplement une version hétérogène de nos implantations pour s'adapter aux différences de puissances entre les machines parallèles disponibles.

Nous envisageons par la suite de poursuivre l'étude des versions récursives de nos algorithmes en testant nos implantations sur des plates-formes pour lesquelles le ratio entre puissance de calcul et vitesse de communication est inversée. Par exemple, nous souhaitons porter nos codes sur une grappe de processeurs de puissance modeste connectés par un réseau rapide de type Myrinet. Dans une telle configuration, l'augmentation du volume de communication affectera moins les performances, et le gain en termes de calcul sera plus perceptible.

3.4 Algorithme d'ordonnancement mixte à étape unique sans réplication de données

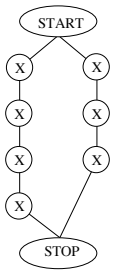
Toutes les implantations présentées dans la paragraphe précédent ont été développées « à la main » et maintes fois raffinées pour améliorer les performances obtenues. Nous avons donc souhaité développer un algorithme d'ordonnancement combinant parallélismes de tâches et de données tels que ceux présentés dans [100, 101, 103]. La spécificité de notre algorithme est de baser l'ordonnancement sur une connaissance précise du temps d'exécution de chacune des tâches de l'application. Il nous est en effet possible, grâce à FAST et à son extension parallèle, de déterminer quelle est la meilleure plate-forme d'exécution pour une routine donnée. De plus, il est relativement aisé d'estimer les coûts de redistributions entre deux grilles de processeurs en combinant une analyse du schéma de communication et les informations fournies par FAST sur l'état actuel du réseau d'interconnexion. Du fait de l'état actuel de l'extension de FAST pour la gestion des routines parallèles, nous avons limité notre étude aux applications composées d'appels à des routines d'algèbre linéaire de type SCALAPACK.

Nous avons cherché à améliorer différents aspects des algorithmes d'ordonnancement mixte que nous avons étudiés. Tout d'abord, ces algorithmes dissocient les phases d'allocation et d'ordonnancement. Le fait de séparer ces deux phases peut amener à ne pas détecter une exécution parallèle possible de plusieurs tâches si moins de processeurs sont alloués à chacune de ces tâches. Un exemple de ce genre de situation sera présenté dans le paragraphe suivant. De plus, les coûts de redistributions entre les différentes tâches sont rarement gérés. Pour répondre à ces problèmes, l'allocation et l'ordonnancement des tâches d'un graphe sont effectués simultanément dans notre

algorithme. Afin de limiter les multiples copies de données, et par conséquent l'augmentation de la consommation mémoire, dues à l'utilisation du parallélisme mixte, nous avons également choisi de développer notre algorithme d'ordonnancement en interdisant la réplication de données. Cette contrainte n'est en effet pas prise en compte dans les différents algorithmes étudiés. Dans le paragraphe suivant, nous nous efforcerons donc de justifier ce choix en présentant, sur un exemple simple, le gain en termes de temps apporté par l'utilisation du parallélisme mixte et l'évolution de la consommation mémoire en fonction de la solution choisie. Nous présenterons ensuite le modèle de graphe de tâches que nous avons défini pour représenter les applications à ordonnancer. Puis nous détaillerons les différents points de notre algorithme d'ordonnancement. Enfin, nous présenterons le type d'ordonnancement obtenu pour une multiplication de matrices complexes et pour l'algorithme de Strassen.

3.4.1 Exemples motivants

Pour démontrer pourquoi il est important de ne pas dissocier les phases d'allocation et d'ordonnancement, nous avons considéré le graphe de tâches présenté par la figure 3.26 (gauche), dont toutes les tâches sont identiques. La plate-forme sur laquelle doivent être placées ces tâches est composée de treize processeurs. Comme nous l'avons expliqué au chapitre précédent, l'exécution d'un produit de matrices sur un tel nombre de processeurs implique une dégradation notable des performances. Un ordonnancement utilisant le parallélisme mixte peut dans ce cas s'avérer très efficace. Nous avons considéré dans cet exemple trois sous–grilles composées respectivement de quatre, neuf et douze processeurs, sur lesquelles placer les tâches du graphes. Les temps d'exécution obtenus sur chacune de ces configurations sont données par la figure 3.26 (droite).



Configuration	Temps
2×2	$11,56$
3×3	$8,44$
3×4	$6,84$
1×13	$16,70$

FIG. 3.26 – Exemple de graphe de tâches identiques (gauche) et temps d'exécution d'une tâche sur différentes grilles de processeurs (droite).

Lorsque les phases d'allocation et d'ordonnancement sont séparées, la phase d'allocation va d'abord chercher à allouer chaque tâche sur la grille de processeurs où le temps d'exécution sera le plus faible, soit la configuration 3×4. Cette solution a pour conséquence de séquentialiser l'exécution de ces tâches lors de la phase d'ordonnancement. Le temps de complétion sera dans ce cas de l'ordre de 48 secondes.

En revanche, si allocation et ordonnancement sont gérés simultanément en associant les tâches aux différentes estimations de temps d'exécution, il est alors possible d'étudier une solution allouant moins de processeurs à une tâche pour permettre l'exécution d'une seconde en parallèle. Ainsi, notre algorithme est capable de produire un ordonnancement dont le temps de complétion sera de l'ordre de $32,5$ secondes, soit un gain d'environ 30% par rapport à la solution précédente.

Pour que les algorithmes d'ordonnancement de la littérature parviennent à trouver cette so-

lution mixte, leur phase d'allocation repose sur une fonction de minimisation, par exemple du chemin critique. Plusieurs phases de raffinement seront alors nécessaires.

Pour justifier notre choix de ne pas répliquer les données, nous allons étudier l'application du parallélisme mixte à l'opération suivante de multiplication de matrices complexes :

$$C = \begin{cases} C_r = A_r \times B_r - A_i \times B_i \\ C_i = A_r \times B_i + A_i \times B_r \end{cases}$$

Supposons que l'on dispose pour cette expérience de deux grilles de processeurs de taille p. Nous avons donc $2p^2$ processeurs disponibles pour exécuter nos calculs. Nous supposons également les données de ce problème sont distribuées de la manière suivante : A_r et A_i sont alignées sur les p^2 premiers processeurs, alors que B_r et B_i sont distribuées sur les p^2 derniers. Enfin, nous ajoutons la contrainte supplémentaire que C_r et C_i soient sur les p^2 premiers processeurs à la fin du calcul. La figure 3.27 présente les temps d'exécution obtenus en utilisant respectivement le parallélisme de données (a), le parallélisme mixte avec réplication de données (b) et le parallélisme mixte sans réplication (c).

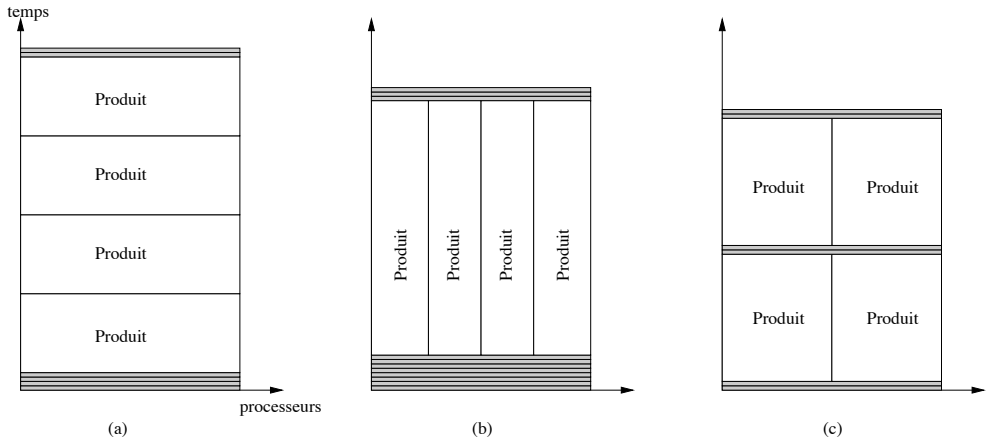


FIG. 3.27 – Temps d'exécution d'un produit de matrices complexes sur $2p^2$ processeurs pour différents types de parallélisme.

Sur cette figure, les parties grisées représentent les redistributions de données. Les deux opérations d'addition et de soustraction ne sont pas représentées, leur temps d'exécution étant négligeable par rapport au reste de l'application. Nous pouvons voir que l'utilisation du parallélisme mixte permet de réduire le temps d'exécution de l'application. De plus, nous pouvons constater que, lorsque les données du problèmes possèdent déjà une distribution, le fait de ne pas les répliquer permet de diminuer le nombre de redistributions effectuées.

Nous avons comparé l'espace mémoire nécessaire à chacun des trois types d'exécution précédents. Pour cela, nous avons dénombré le nombre de variables temporaires allouées sur les p^2 premiers processeurs. En effet, ces processeurs stockent les parties réelles et imaginaires des matrices A et C. C'est donc sur ces processeurs que l'augmentation de la consommation mémoire sera la plus préjudiciable. Nous sommes parvenus aux résultats suivants. Si chacune des matrices impliquées dans le calcul est de rang N, la version utilisant le parallélisme de données et notre version sans réplication nécessiteront $3N^2/4$ éléments temporaires par processeur, alors que la version mixte avec réplication en demandera $3N^2/2$. Le fait de répliquer les données multiplie donc par deux le nombre d'éléments temporaires nécessaires à l'exécution du calcul. Nous pouvons donc affirmer que, dans le cas où les données d'une application ont déjà une distribu-

tion, l'utilisation du parallélisme mixte sans réplication permet non seulement de réduire le temps d'exécution, mais aussi de ne pas augmenter la consommation mémoire de manière importante.

3.4.2 Modèle de graphe de tâches

Notre algorithme d'ordonnancement à étape unique est basé sur une structure de graphe de tâches proche de la structure de MDG proposée par Ramaswamy [103]. Une application est ainsi décrite par un graphe orienté acyclique $G = (V, E)$ où les nœuds représentent les tâches, potentiellement parallèles, et les arcs indiquent les dépendances d'exécution entre les tâches. Deux nœuds particuliers sont ajoutés à cette structure de graphe. Le nœud `START` précède tous les autres nœuds du graphe. Les tâches ayant ce nœud pour prédécesseur impliquent des données initiales de l'application. À l'inverse, le nœud `STOP` succède à tous les autres nœuds du graphe. Les tâches ayant ce nœud pour successeur impliquent des données terminales de l'application.

Les données initiales et terminales de l'application possèdent des distributions fixes que l'ordonnancement devra respecter. Par exemple, si une application utilise deux matrices, la première distribuée sur la moitié des processeurs et la seconde distribuée sur l'autre moitié, un ordonnancement basé sur le parallélisme de données devra inclure les coûts de redistribution de ces matrices sur l'ensemble des processeurs. Il en va de même pour les résultats produits par l'application.

Puisque les tâches que nous considérons correspondent à des routines de type ScaLAPACK, chacune d'elles n'implique au plus que trois données en entrée et ne produit qu'un seul résultat. Ceci a pour conséquence qu'un nœud du graphe n'a au plus que trois prédécesseurs. En revanche, le nombre de successeurs d'un nœud n'est pas limité, une même donnée pouvant être utilisée par plusieurs autres tâches.

Comme nous l'avons indiqué précédemment, notre algorithme d'ordonnancement repose sur des informations relatives à chacune des tâches de l'application, qui sont intégrées dans notre structure de graphe de tâches de la manière suivante. Tout d'abord, une table des coûts de redistribution est associée à chaque arc. Nous associons ensuite à chacun des nœuds une date de début, à partir de laquelle cette tâche sera considérée comme prête, la localisation courante des données d'entrée de cette tâche et une liste de couples { configuration ; temps d'exécution }. Chacune de ces configurations est définie par un tuple $\{p, q, mb, nb, liste\}$. p et q sont respectivement les nombres de lignes et de colonnes de la grille de processeurs utilisée. mb et nb sont les dimensions d'un bloc de distribution. Enfin, le dernier paramètre de ce tuple est la $liste$ des processeurs de la grille. Nous avons choisi d'utiliser une liste des processeurs appartenant à une configuration plutôt que les coordonnées d'un processeur particulier afin de pouvoir gérer des grilles de processeurs non contiguës comme le montre la figure 3.28. Nous supposerons cependant que les phénomènes de contention sont négligeables.

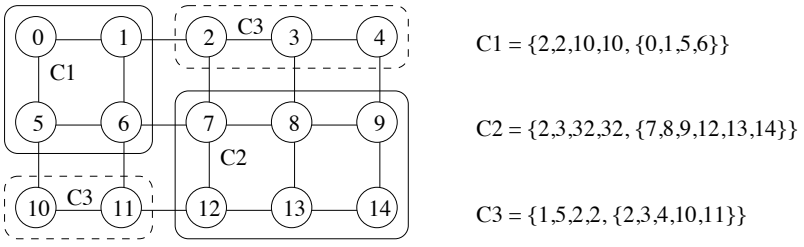


FIG. 3.28 – Exemple de configurations et de leur description.

Ce système de configurations permet en outre de ne pas faire d'hypothèse forte concernant

l'homogénéité de la plate-forme d'exécution. En effet, si les processeurs d'une même configuration sont homogènes, cela suffit pour obtenir de bonnes performances. De plus, si des routines adaptées à une exécution sur plate-forme hétérogène sont utilisées, il n'existe alors plus aucune contrainte d'homogénéité. Cela nous permet de plus de prendre en compte la forme de grille, ce qui, comme nous l'avons expliqué au chapitre précédent, peut avoir un impact non négligeable sur les performances obtenues.

3.4.3 Description de l'algorithme d'ordonnancement et de placement simultanés

Notre but est de déterminer pour chacune des tâches s'il est plus intéressant de l'exécuter sur l'ensemble des processeurs disponibles en utilisant le parallélisme de données ou bien d'assigner cette tâche à un sous-ensemble de processeurs à la manière du parallélisme mixte. Dans la seconde solution, certains processeurs restent disponibles pour l'exécution d'une autre tâche. Nous allons tout d'abord décrire l'algorithme servant à déterminer quelle décision prendre dans le cas de l'ordonnancement d'un ensemble de tâches indépendantes d'une autre. Nous étendrons ensuite cet algorithme de décision au cas d'un graphe de tâches entier.

3.4.3.1 Ordonnancement mixte de tâches indépendantes

Considérons une tâche T_1 et un ensemble de tâches $\mathcal{I} = \{T_2, \cdots, T_n\}$ indépendantes de T_1. Chaque tâche T_i dans $T_1 \cup \mathcal{I}$ est associée à un sous-ensemble de $\mathcal{C} = \{C_1, \cdots, C_m\}$ l'ensemble des configurations possibles. Le temps d'exécution d'une tâche T_i sur la configuration C_j est noté $t_{i,j}$. Ce temps n'inclut cependant pas les éventuelles communications nécessaires au transfert des données vers les processeurs de la configuration choisie.

L'idée principale pour déterminer si T_1 doit être assignée sur l'ensemble des processeurs ou sur l'une de ses configurations associées est la suivante. Supposons que T_1 soit assignée sur la configuration C_i. L'instant à partir duquel tous les processeurs de cette configuration sont prêts à recevoir les données nécessaires au calcul sera alors noté $disp(C_i)$. Pour chaque donnée $D_i \in \mathcal{D}$, où $\mathcal{D}$ est l'ensemble des données à transférer pour pouvoir exécuter T_1, C_{D_i} désigne la configuration sur laquelle cette donnée est actuellement distribuée et $R(D_i)$ représente le temps de transfert de D_i vers C_i. La fonction présentée par la figure 3.29 est alors utilisée pour mettre à jour les différents temps de disponibilité des configurations impliquées dans cette phase de redistribution. Cela garantit qu'un processeur ne puisse commencer à calculer que s'il a terminé de participer aux éventuelles redistributions des données qu'il possède.

Mise à jour $(C, \mathcal{D})$
$t \longleftarrow disp(C)$
pour $i = 1$ à $\|\mathcal{D}\|$ **faire**
  $m \longleftarrow \max(t, disp(C_{D_i}))$
  $t = m + R(D_i)$
  pour Chaque configuration C_j dont au moins un processeur possède une partie de D_i **faire**
    $disp(C_j) = m + R(D_i)$
  fin pour
fin pour

FIG. 3.29 – Fonction de mise à jour des temps de disponibilité.

Afin d'ordonnancer d'autres tâches en parallèle de l'exécution de T_1 en utilisant le parallélisme mixte, nous devons considérer l'ensemble $\mathcal{U}$ constitué de couples { tâche ; configuration } où chaque tâche est indépendante de T_1 et chacune des configurations ne comprend que des processeurs de $\mathcal{P}\backslash C_i$. Il est à noter qu'une même tâche peut apparaître plusieurs fois dans cet ensemble, si plusieurs de ses configurations associées n'impliquent aucun processeur de C_i. Les éléments de cet ensemble sont triés en fonction de la quantité minimale de redistribution qu'ils engendrent du fait du placement de T_1 sur C_i. De plus, les tâches générant une donnée terminale devant être distribuée sur C_i à la fin du calcul ne seront pas incluses dans $\mathcal{U}$. Si ce choix nous prive de la possibilité d'exécuter plus de tâches en parallèle de T_1, il permet de ne pas compliquer d'avantage l'algorithme. En effet, si nous gérions ces tâches, il nous faudrait penser à retarder la redistribution des données terminales à la fin du calcul de T_1. Nous ne considérerons ces tâches produisant des données terminales que lorsqu'elles peuvent assignées sur la configuration où doivent se trouver les données.

Considérons l'un de ces couples $\{T_j, C_k\}$. La tâche T_j est supposée commencer au temps $disp(C_k)$. Cependant, il est possible que la redistribution des données nécessaires au calcul de T_j sur C_k implique certains processeurs de C_i. Pour ne pas séquentialiser l'exécution de ces deux tâches, cette communication doit être effectuée avant l'exécution de T_1. Dans ce cas, le temps à partir duquel C_i et chacune des configurations dont au moins un des processeurs possèdent une partie des données à transférer, doit être augmenté du temps de redistribution correspondant comme le montre la figure 3.30. La fonction de mise à jour est pour cela appelée une nouvelle fois.

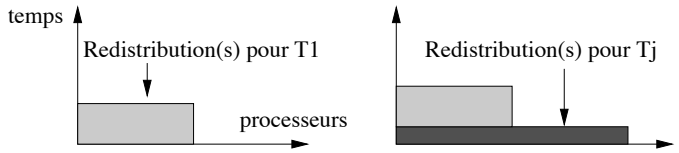


FIG. 3.30 – Augmentation de $disp(C_i)$ lors de la redistribution engendrée par T_j.

Si le placement de T_j sur C_k conserve la condition 3.12, alors cette tâche sera ordonnancée en utilisant le parallélisme mixte et le couple correspondant sera ajouté à $\mathcal{S}$, l'ensemble des tâches déjà ordonnancées. De plus, l'ensemble $\mathcal{U}$ est mis à jour. Cette mise à jour porte sur deux points. Tout d'abord, tous les couples impliquant T_j sont supprimés de $\mathcal{U}$. Ensuite, si l'ordonnancement de T_j génère des tâches prêtes indépendantes de T_1, les couples correspondants à ces tâches sont ajoutés à $\mathcal{U}$.

$$\max_k \left(disp(C_k) + \sum_j t_{j,k} \right) \leq disp(C_i) + t_{1,i}. \tag{3.12}$$

En revanche, si la condition n'est plus vérifiée, la mise à jour des temps de disponibilité est annulée, le couple $\{T_j, C_k\}$ est enlevé de $\mathcal{U}$, et l'algorithme passe au candidat suivant.

Lorsque tous les couples de $\mathcal{U}$ ont été considérés, $t_{mixte} = disp(C_i) + t_{1,i}$ est le temps d'exécution total de toutes les tâches ordonnancées. Il se peut qu'à la fin du parcours de $\mathcal{U}$, la seule tâche à ordonnancer soit T_1. Dans ce cas, la solution mixte proposée sera de placer T_1 sur C_i et de laisser les autres configurations inactives. Cette situation peut par exemple se produire lorsqu'il ne reste plus que des tâches produisant des données terminales, ou de durée plus élevée, à ordonnancer en parallèle de T_1.

Après avoir réinitialisé le temps de disponibilité de l'ensemble des processeurs, il est possible

de calculer le temps de complétion correspondant à l'exécution de ces mêmes tâches en utilisant uniquement le parallélisme de données, *i.e.*, $t_{//} = \sum_i (t_{i,\mathcal{P}} +$ coût de redistribution associée à $T_i)$, où les données associées à chaque tâche T_i sont actuellement distribuées sur la configuration C_i. Si t_{mixte} est inférieur à ce $t_{//}$, cela veut dire que la solution mixte produite est meilleure que celle basée sur le parallélisme de données. L'algorithme retourne alors $\mathcal{S}$, *i.e.*, T_1 sera assignée sur C_i et chacune des T_j retenues sur le C_k correspondant. Dans le cas contraire, l'algorithme cherche à assigner T_1 sur une autre de ses configurations associées. Si aucune des configurations disponibles pour T_1 ne produit un ordonnancement plus efficace qu'une exécution n'utilisant que le parallélisme de données, T_1 sera alors assignée sur tous les processeurs.

Décision ($T_1, \mathcal{I}$)
pour Chaque C_i associée à T_1 **faire**
  $t_{//} = disp(\mathcal{P})$
  $\mathcal{S} \longleftarrow \{T_1, C_i\}$
  Mise à jour ($C_i, \mathcal{D}$)
  Construction de $\mathcal{U}$
  tant que $\mathcal{U} \neq \emptyset$ **faire**
    Soit $\{T_j, C_k\}$ le premier couple de $\mathcal{U}$
    Mise à jour ($C_k, \mathcal{D}$)
    si $\max_k \left(disp(C_k) + \sum_j t_{j,k} \right) \leq disp(C_i) + t_{1,i}$ **alors**
      Ajouter $\{T_j, C_k\}$ à $\mathcal{S}$ et mettre à jour $\mathcal{U}$
    sinon
      Annuler la mise à jour des temps de disponibilité
      Retirer $\{T_j, C_k\}$ de $\mathcal{U}$
    fin si
  fin tant que
  $t_{mixte} = disp(C_i) + t_{1,i}$
  $disp(\mathcal{P}) = t_{//}$
  pour Chaque $T_i \in \mathcal{S}$ **faire**
    Mise à jour ($\mathcal{P}, \mathcal{D}$)
    $t_{//} = t_{//} + t_{i,\mathcal{P}}$ + Temps de redistribution des données terminales
    $disp(\mathcal{P}) = t_{//}$
  fin pour
  si $t_{mixte} \leq t_{//}$ **alors**
    Retourner $\mathcal{S}$
  fin si
fin pour
$\mathcal{S} \longleftarrow \{T_1, \mathcal{P}\}$
Retourner ($\mathcal{S}$)

FIG. 3.31 – Algorithme de décision d'exécution en parallélisme mixte.

L'algorithme 3.31 nous permet de décider s'il est ou non possible d'effectuer d'autre tâches pendant l'exécution d'une tâche T_1 en utilisant le parallélisme mixte. La complexité de cet algorithme est $\mathcal{O}(c^3 \|\mathcal{I}\|)$, où c est le nombre maximal de configurations associées à une tâche.

3.4.3.2 Algorithme

Nous sommes désormais capables de décider, pour un ensemble de tâches $\mathcal{I}$ indépendantes d'une autre tâche T_1, s'il est préférable d'effectuer T_1 en utilisant le parallélisme de données ou d'exécuter cette tâche en parallèle d'un sous-ensemble de $\mathcal{I}$.

Considérons maintenant le graphe de tâches extrait de l'application dans son ensemble. À chaque instant, il existe un nombre potentiellement grand de tâches prêtes à être exécutées. L'idée de notre algorithme est d'isoler une de ces tâches prêtes et de considérer l'ensemble des tâches qui en sont indépendantes. L'algorithme de décision, donné en figure 3.31, est alors appliqué sur ces données. La liste des tâches prêtes est ensuite mise à jour et l'algorithme continue jusqu'à ce que cette liste soit vide. Cette liste est triée en fonction de la *priorité* des tâches, définie comme étant le poids du plus long chemin allant d'une tâche au nœud `STOP`. Pour déterminer ce poids, nous estimons le temps d'exécution de chacune des tâches du chemin en utilisant la configuration impliquant tous les processeurs de la plate-forme.

Soit $\mathcal{T}$ la liste triée des tâches prêtes. Nous notons T_1 la première tâche de cette liste, *i.e.*, celle qui est la plus éloignée de la fin de l'application. Les tâches prêtes restantes sont indépendantes de T_1 du point de vue du flot d'exécution, mais pas nécessairement du point de vue des données, puisqu'elles peuvent partager une donnée commune. Comme nous supposons que les données ne sont pas répliquées, si deux tâches partagent une donnée, elles ne peuvent pas être exécutées en parallèle. Il est donc nécessaire de déterminer le sous-ensemble de tâches prêtes qui sont également indépendantes de T_1 du point de vue des données. Pour chaque $T_i \in \mathcal{T}$, si cette tâche et T_1 ont un prédécesseur commun alors T_i n'est pas indépendante de T_1.

Notre algorithme d'ordonnancement mixte à étape unique, présentée par la figure 3.32, prend en entrée un ensemble de processeurs $\mathcal{P}$ et un graphe de tâches G. La liste des tâches initiales, *i.e.*, celles ayant le nœud `START` pour prédécesseur, est construite puis triée en fonction du coût estimé du plus long chemin reliant chacune de ces tâches au nœud `STOP`. L'algorithme va ensuite déterminer l'ensemble des tâches indépendantes de la première tâche de cette liste puis décider s'il est possible d'exhiber du parallélisme mixte grâce à l'algorithme de la figure 3.31. Si ce n'est pas le cas, cette tâche sera assignée sur l'ensemble des processeurs et l'algorithme considérera la tâche suivante. Ce processus est alors réitéré tant qu'il reste des tâches prêtes.

Ordonnancement ($\mathcal{P}, G$)
Calculer la priorité de chaque tâche
$\mathcal{T} \longleftarrow$ Ensemble trié des tâches initiales
tant que $\mathcal{T} \neq \emptyset$ **faire**
  $T_1 \longleftarrow$ première tâche de $\mathcal{T}$
  $\mathcal{I} \longleftarrow$ Ensemble des tâches indépendantes de T_1
  $\mathcal{S} \longleftarrow$ **Décision** ($T_1, \mathcal{I}$)
  pour Chaque tâche $T_i \in \mathcal{S}$ **faire**
    Assigner T_i sur C_i
    Mettre à jour $\mathcal{T}$
  fin pour
fin tant que

FIG. 3.32 – Algorithme d'ordonnancement mixte à étape unique.

Dans le pire des cas, notre algorithme produit un ordonnancement n'utilisant que le pa-

rallélisme de données. L'algorithme n'assigne alors qu'une seule tâche par étape et effectue donc le calcul d'indépendance et le test des solutions mixtes $||V||$ fois. La complexité dans le pire des cas est donc $\mathcal{O}(c^3||V||^2)$, où c est le nombre maximal de configurations associées à une tâche.

3.4.4 Validation expérimentale

Afin de comparer les ordonnancements produits par notre algorithme à des exécutions utilisant uniquement le parallélisme de données, nous avons étudié deux exemples : la multiplication de matrices complexes et l'algorithme de Strassen. Les graphes de tâches correspondant à ces deux applications sont présentés par la figure 3.33.

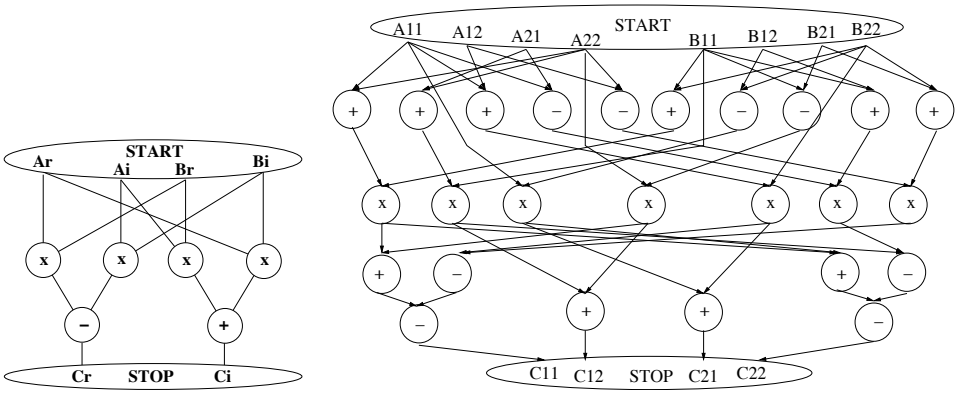


FIG. 3.33 – Graphes de tâches de la multiplication de matrices complexes (gauche) et de l'algorithme de Strassen (droite).

Les conditions d'exécution de ces deux applications sont identiques. En effet, nous avons considéré le cas où la matrice A est distribuée sur une grille 2×2 de processeurs, la matrice B étant distribuée sur une grille de même taille, mais totalement disjointe de la première. Une exécution utilisant le parallélisme de données s'effectuera donc sur 8 processeurs, organisés en une grille 2×4. Dans le cas de la multiplication de matrices complexes, la matrice A (resp. B) est en fait découpée en deux matrices Ar et Ai (resp. Br et Bi) représentant respectivement les parties réelle et imaginaire. De même, dans le cadre de l'algorithme de Strassen, chacune des matrices impliquées est en fait découpée en quatre quarts. Nous avons enfin ajouté une contrainte obligeant le résultat à être distribuée sur la même grille que la matrice A.

Nous avons tout d'abord ordonnancé ces deux applications sur une plate-forme homogène. Nous avons ensuite cherché à valider le comportement de notre algorithme lorsque la plate-forme cible est hétérogène. Nous n'avons conservé pour cela que l'algorithme de Strassen. Les deux paragraphes suivants présentent les conditions expérimentales et les résultats obtenus respectivement pour ces deux types de plate-forme.

3.4.4.1 Ordonnancement mixte homogène

Le tableau 3.2 présente les temps d'exécution des opérations composant les graphes de tâches de nos deux applications sur les différentes grilles de processeurs considérées. Chacune des matrices impliquées dans les calcul est de taille 2048×2048. Le tableau 3.3 présente quant à lui les

temps de transfert d'une donnée de taille 2048×2048 d'une configuration à une autre. C_1 et C_2 sont les grilles sur lesquelles sont respectivement distribuées A et B, alors que $\mathcal{P}$ désigne la grille globale composée de tous les processeurs disponibles.

	2×2	2×4
Produit	$23,59$	$14,13$
Addition	$0,11$	$0,05$

TAB. 3.2 – Temps d'exécution des différentes opérations utilisées dans CMM et Strassen.

	C_1	C_2	$\mathcal{P}$
C_1	0	$1,18$	$0,75$
C_2	$1,18$	0	$0,75$
$\mathcal{P}$	$0,75$	$0,75$	0

TAB. 3.3 – Coûts de redistribution d'une matrice entre les différentes configurations.

La figure 3.34 donne l'ordonnancement correspondant à l'exécution de l'algorithme donnée par la figure 3.35. Pour des raisons de lisibilité, et afin de faire apparaître toutes les opérations, nous n'avons pas respecté l'échelle correspondant aux temps d'exécutions des différentes tâches donnés dans la figure 3.35. En revanche, les parties grisées représentent les temps d'inactivité des configurations. Nous pouvons voir que les multiplications sont effectuées deux par deux en parallèle. En revanche, l'addition et la soustraction sont ordonnancées séquentiellement sur la configuration C_1. Ceci s'explique par le fait que les résultats de ces opérations sont des données terminales de l'application. Ils doivent être distribuées sur cette configuration à la fin du calcul. Cette solution est choisie par notre algorithme, le coût d'un transfert de donnée d'une configuration à l'autre suivi d'une addition sur la moitié des processeurs étant nettement inférieur à deux redistributions suivies d'une addition sur tous les processeurs et de la redistribution du résultat.

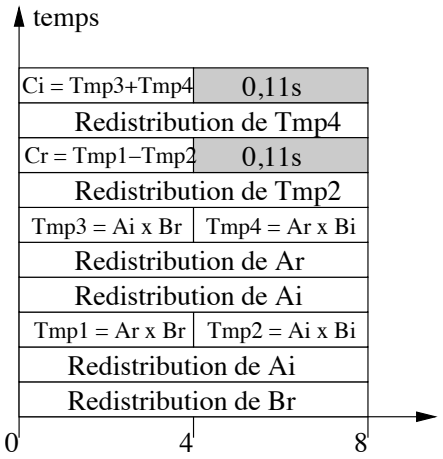


FIG. 3.34 – Ordonnancement mixte produit pour la multiplication de matrices complexes.

La figure 3.36 présente quant à elle l'ordonnancement produit par notre algorithme mixte à étape unique pour l'algorithme de Strassen. Comme nous pouvons le voir, les additions sont assignées sur la configuration où sont distribuées les données qu'elles impliquent. Les six premières multiplications de l'algorithme de Strassen vont ensuite être effectuées par paire en utilisant le parallélisme mixte. En revanche, le septième et dernier produit sera réalisé sur l'ensemble des processeurs en utilisant le parallélisme de données puisqu'aucune tâche de durée suffisante n'est à effectuer en parallèle de ce produit. Le parallélisme mixte va de nouveau être employé pour

Priorités : $T_{\times 1} = T_{\times 2} = T_{\times 3} = T_{\times 4} = 14,18$, $T_+ = T_- = 0,05$
$\mathcal{T} = \{T_{\times 1}, T_{\times 2}, T_{\times 3}, T_{\times 4}\}$
$T_1 \longleftarrow T_{\times 1}$
$\mathcal{I} \longleftarrow \{T_{\times 2}\}$

$T_{\times 1}$ sur C_1
$t_{//} = 0$
$\mathcal{S} \longleftarrow \{T_{\times 1}, C_1\}$
Mise à jour (C_1, B_r) $disp(C_1) = disp(C_2) = disp(\mathcal{P}) = 1,18$
$\mathcal{U} \longleftarrow \{T_{\times 2}, C_2\}$
Mise à jour (C_2, A_i) $disp(C_2) = disp(C_1) = disp(\mathcal{P}) = 2,36$
$disp(C_2) + t_{\times 2,2} \leq disp(C_1) + t_{\times 1,1} (25,95 \leq 25,95)$
$\mathcal{S} \longleftarrow \{\{T_{\times 1}, C_1\}, \{T_{\times 2}, C_2\}\}$
Décision ($T_{\times 1}, \mathcal{I}$)
$t_{mixte} = 25,95$
$disp(\mathcal{P}) = 0$
Mise à jour ($\mathcal{P}$, $\{A_r, B_r\}$) $disp(\mathcal{P}) = disp(C_1) = disp(C_2) = 1,5$
$disp(\mathcal{P}) = t_{//} = 15,63$
Mise à jour ($\mathcal{P}$, $\{A_i, B_i\}$) $disp(\mathcal{P}) = disp(C_1) = disp(C_2) = 17,13$
$disp(\mathcal{P}) = t_{//} = 31,26$
$t_{mixte} \leq t_{//} (25,95 \leq 31,26)$
Retourner $\{\{T_{\times 1}, C_1\}, \{T_{\times 2}, C_2\}\}$

$\mathcal{T} = \{T_{\times 3}, T_{\times 4}, T_+\}$
$T_1 \longleftarrow T_{\times 3}$
$\mathcal{I} \longleftarrow \{T_{\times 4}, T_+\}$

$T_{\times 3}$ sur C_1
$t_{//} = 25,95$
$\mathcal{S} \longleftarrow \{T_{\times 3}, C_1\}$
Mise à jour (C_1, A_i) $disp(C_1) = disp(C_2) = disp(\mathcal{P}) = 27,13$
$\mathcal{U} \longleftarrow \{T_{\times 4}, C_2\}$
Mise à jour (C_2, A_r) $disp(C_2) = disp(C_1) = disp(\mathcal{P}) = 28,31$
$disp(C_2) + t_{\times 4,2} \leq disp(C_1) + t_{\times 3,1} (51,9 \leq 51,9)$
$\mathcal{S} \longleftarrow \{\{T_{\times 3}, C_1\}, \{T_{\times 4}, C_2\}\}$
Décision ($T_{\times 3}, \mathcal{I}$)
$t_{mixte} = 51,9$
$disp(\mathcal{P}) = 25,95$
Mise à jour ($\mathcal{P}$, $\{A_i, B_r\}$) $disp(\mathcal{P}) = disp(C_1) = disp(C_2) = 27,45$
$disp(\mathcal{P}) = t_{//} = 41,58$
Mise à jour ($\mathcal{P}$, $\{A_r, B_i\}$) $disp(\mathcal{P}) = disp(C_1) = disp(C_2) = 43,08$
$disp(\mathcal{P}) = t_{//} = 57,21$
$t_{mixte} \leq t_{//} (51,9 \leq 57,21)$
Retourner $\{\{T_{\times 3}, C_1\}, \{T_{\times 4}, C_2\}\}$

$\mathcal{T} = \{T_+, T_-\}$
$T_1 \longleftarrow T_+$
$\mathcal{I} \longleftarrow \{T_-\}$

T_+ sur C_1
$t_{//} = 51,9$
$\mathcal{S} \longleftarrow \{T_+, C_1\}$
Mise à jour (C_1, $Tmp2$) $disp(C_1) = disp(C_2) = disp(\mathcal{P}) = 53,08$
$t_{mixte} = 53,19$
$disp(\mathcal{P}) = 51,9$
Décision ($T_+, \mathcal{I}$)
Mise à jour ($\mathcal{P}$, $\{Tmp1, Tmp2\}$) $disp(\mathcal{P}) = disp(C_1) = disp(C_2) = 53,4$
$disp(\mathcal{P}) = t_{//} = 54,2$
$t_{mixte} \leq t_{//} (53,19 \leq 54,2)$
Retourner $\{T_+, C_1\}$

$\mathcal{T} = T_-$
$T_1 \longleftarrow T_-$
$\mathcal{I} \longleftarrow \emptyset$

T_- sur C_1
$t_{//} = 53,19$
$\mathcal{S} \longleftarrow \{T_-, C_1\}$
Mise à jour (C_1, $Tmp4$) $disp(C_1) = disp(C_2) = disp(\mathcal{P}) = 54,37$
$t_{mixte} = 54,48$
$disp(\mathcal{P}) = 53,19$
Décision ($T_-, \mathcal{I}$)
Mise à jour ($\mathcal{P}$, $\{Tmp3, Tmp4\}$) $disp(\mathcal{P}) = disp(C_1) = disp(C_2) = 54,69$
$disp(\mathcal{P}) = t_{//} = 55,49$
$t_{mixte} \leq t_{//} (54,48 \leq 55,49)$
Retourner $\{T_-, C_1\}$

FIG. 3.35 – Exécution de l'algorithme d'ordonnancement pour l'application de multiplication de matrices complexes.

ordonnancer les quatre additions intermédiaires servant à calculer C_{11} et C_{22}. Enfin, les quatre dernières additions terminant le calcul des quatre quarts de C sont effectuées sur la première configuration, leurs résultats étant des données terminales de l'application.

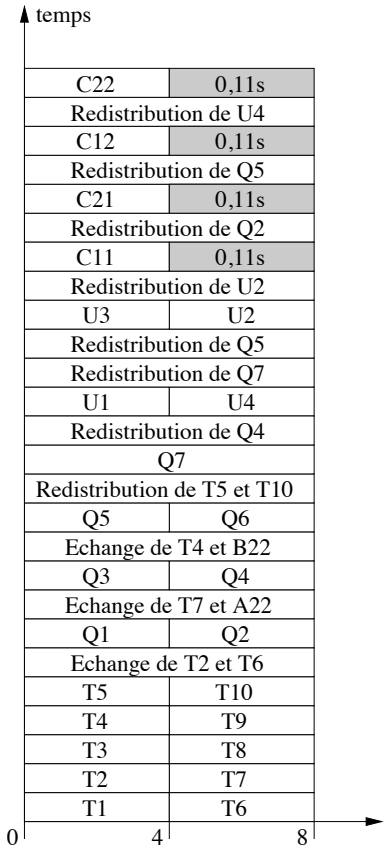


FIG. 3.36 – Ordonnancement mixte produit pour l'algorithme de Strassen.

La figure 3.37 présente le gain obtenu par les ordonnancements produits par notre algorithme par rapport à des ordonnancements utilisant uniquement le parallélisme de données. La diminution de ce gain s'explique par l'évolution des temps d'exécution d'une multiplication respectivement sur quatre et huit processeurs. En effet, lorsque la taille des matrices augmente, le rapport entre ces deux temps tend vers deux. Le temps nécessaire au calcul d'une paire de produit en utilisant le parallélisme mixte se rapproche donc du temps qu'il faut pour effectuer ces mêmes produits l'un après l'autre en utilisant le parallélisme de données.

3.4.4.2 Ordonnancement mixte hétérogène

Comme nous l'avons indiqué précédemment, nous n'avons pas posé d'hypothèses fortes quant à l'homogénéité de la plate-forme d'exécution lors de la conception de notre algorithme. Nous avons donc ordonnancé l'algorithme de Strassen sur une plate-forme composée deux grappes homogènes de puissances différentes connectées par un lien Fast Ethernet. Nous

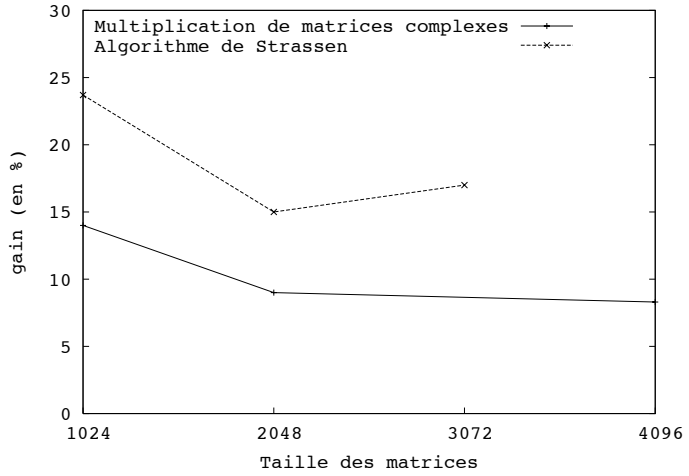


FIG. 3.37 – Gains des ordonnancements mixtes sur les ordonnancements en parallélisme de données pour la multiplication de matrices complexes et l'algorithme de Strassen.

dénotons par C_1 la configuration « lente » composée de quatre processeurs et par C_2 la configuration « rapide » impliquant le même nombre de processeurs, alors que $\mathcal{P}$ désigne la grille globale composée de tous les processeurs disponibles. Les matrices A et B sont respectivement distribuées sur les grilles C_1 et C_2.

Le tableau 3.4 présente les temps d'exécution des opérations composant le graphe de tâches de l'algorithme de Strassen sur les différentes configurations considérées. Chacune des matrices impliquées dans les calcul est de taille 1024×1024. Le tableau 3.5 présente quant à lui les temps de transfert d'une donnée de taille 1024×1024 d'une configuration à une autre.

	C_1	C_2	$\mathcal{P}$
Produit	$25,1$	$5,7$	$23,1$
Addition	$0,06$	$0,04$	$0,02$

TAB. 3.4 – Temps d'exécution des différentes opérations de l'algorithme de Strassen.

	C_1	C_2	$\mathcal{P}$
C_1	0	$0,35$	$0,22$
C_2	$0,35$	0	$0,22$
$\mathcal{P}$	$0,22$	$0,22$	0

TAB. 3.5 – Coûts de redistribution d'une matrice entre les différentes configurations.

Nous pouvons constater que sur une telle plate-forme, un ordonnancement uniquement basé sur le parallélisme de données aboutira à de mauvaises performances, le temps d'exécution d'un produit sur l'ensemble des processeurs étant plus de quatre fois supérieur à celui obtenu sur la configuration rapide.

La figure 3.38 présente l'ordonnancement produit pour l'algorithme de Strassen ciblant une plate-forme hétérogène. Si cet ordonnancement est loin d'être optimal, les processeurs de C_2 restant inactifs pendant plus de 22 secondes, le gain par rapport à un ordonnancement n'utilisant que le parallélisme de données est très intéressant. En effet, le temps de complétion de notre ordonnancement est de l'ordre de 56 secondes alors que celui de l'ordonnancement en parallélisme de données est de l'ordre de 165 secondes, soit un gain d'environ 66%.

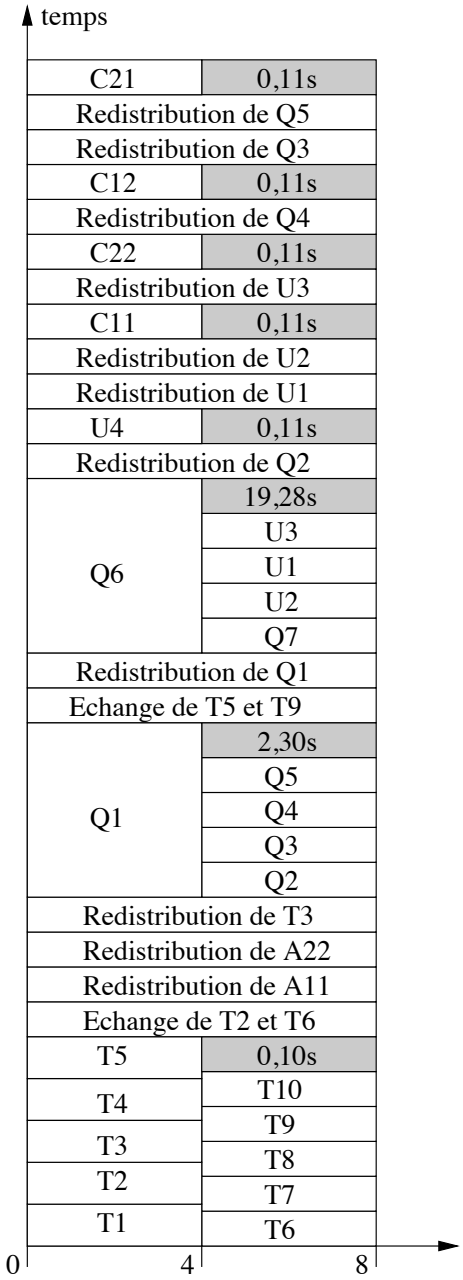


FIG. 3.38 – Ordonnancement mixte produit pour l'algorithme de Strassen sur une plate-forme hétérogène.

3.5 Conclusion

Dans ce chapitre, nous avons cherché à valider l'utilisation du paradigme de programmation, appelé parallélisme mixte, consistant à exploiter simultanément les parallélismes de tâches et de données présents dans une application. Pour cela, nous avons développé plusieurs implantations des algorithmes rapides de produit de matrices de Strassen et Winograd. Ces implantations ont été évaluées théoriquement et validées expérimentalement en les comparant à des implantations utilisant le parallélisme de données.

Nous avons également proposé dans ce chapitre un algorithme original utilisant le parallélisme mixte pour ordonnancer des applications lorsque les données ne peuvent être répliquées. Le principe fondateur de cet algorithme est d'associer à chaque nœud du graphe de tâches de l'application une liste de configurations, *i.e.*, de grilles de processeurs, pour lesquelles nous savons prédire le temps d'exécution de cette tâche. Cela nous permet alors d'effectuer simultanément le placement et l'ordonnancement de ces tâches. Nous avons appliqué cet algorithme à deux applications : la multiplication de matrices complexes et l'algorithme de Strassen. Les ordonnancements produits pour ces applications aboutissent à des temps de complétion inférieurs d'environ 15% à ceux obtenus par un ordonnancement n'utilisant que le parallélisme de données.

Nous envisageons par la suite d'améliorer notre algorithme afin de diminuer les temps d'inactivité qui peuvent apparaître sur l'une des configurations, comme lors de l'ordonnancement de la décomposition de Strassen sur une plate-forme hétérogène. Pour cela, nous pourrions, au lieu de nous arrêter à la première configuration vérifiant la condition, tester plus de configurations pour peut-être trouver une meilleure solution. Bien entendu, ce gain du point de vue de la qualité des ordonnancements produits implique une augmentation de la complexité de l'algorithme.

Nous souhaitons également mieux gérer les tâches produisant des données terminales. En effet, ce type de tâches peut être ordonnancé en utilisant le parallélisme mixte, à condition de gérer correctement la redistribution du résultat, *i.e.*, la décaler jusqu'à ce que tous les processeurs impliqués soient disponibles. Ce décalage ne devra toutefois pas empêcher le placement d'autres tâches sur cette configuration dans l'intervalle de temps libéré.

Enfin, nous voulons étendre l'utilisation du parallélisme mixte et de notre algorithme à des applications plus complexes issues du metacomputing. En effet, nous pensons pouvoir utiliser ces travaux dans le cadre du développement de l'environnement DIET, que nous allons présenter dans le chapitre suivant. Dans un environnement à base de serveurs de calcul tel que DIET, les données issues d'une résolution de problèmes peuvent rester distribuées sur la plate-forme d'exécution à la fin du calcul, et, de plus, leur réplication n'est pas autorisée. Nous nous trouvons donc exactement dans le cadre d'utilisation de notre algorithme. Son utilisation permettrait alors de déployer un calcul sur un ensemble de serveurs sans avoir à effectuer une redistribution complète des données si cela n'est pas nécessaire.

Chapitre 4

DIET : une approche hiérarchique des serveurs de calcul

Slim-Fast™ can be used indefinitely to help maintain a healthy diet.
The Slim-Fast™ FAQ.

4.1 Introduction

Comme nous l'avons vu dans le chapitre 1, l'une des manières les plus « transparentes » pour résoudre des problèmes de très grande taille issus de la simulation numérique via Internet est d'utiliser le modèle RPC en ciblant des serveurs de bibliothèques numériques. D'autres approches pour la résolution de ces problèmes existent telles que les langages orientés objet [74], les environnements à base de passage de messages [83, 92], les boites à outils [56], les environnements de calcul global [88, 122], ceux basés sur le Web [68, 71, 117], etc.

Nous n'avons cependant retenu que l'approche RPC car elle s'est avérée la plus appropriée dans le cadre de la parallélisation de SCILAB. Des environnements utilisant ce paradigme, tels que NETSOLVE [24, 87] ou NINF [85, 89], sont d'ordinaire constitués de cinq types de composants : des **clients** qui soumettent les problèmes qu'ils ont à résoudre à des **serveurs**, une **base de donn ées** qui contient des informations concernant les ressources matérielles et logicielles, un **ordonnanceur** qui choisit un serveur approprié en fonction du problème soumis et des informations contenues dans la base de données, et enfin des **senseurs** qui acquièrent des informations sur le statut des ressources de calcul. Par exemple dans l'architecture de NETSOLVE, nous retrouvons tous ces composants sous la forme de processus clients, serveurs et agent, où l'agent contient à la fois la base de données et l'ordonnanceur.

En travaillant avec NETSOLVE, nous nous sommes aperçus que des améliorations étaient possibles et que, de plus, il était nécessaire de développer un ensemble d'outils utiles au

développement d'applications de type ASP. C'est pour cela que nous développons l'environnement DIET (*Distributed Interactive Engineering Toolbox*) [17, 18, 31, 44, 75]. DIET est un ensemble hiérarchique de composants pour l'élaboration d'applications basées sur des serveurs de calcul.

Dans un premier temps, nous allons présenter NETSOLVE, les travaux effectués dans le cadre d'OURAGAN autour de ce logiciel ainsi que ses lacunes. Puis nous détaillerons l'architecture générale de la plate-forme DIET et ses principaux composants en proposant des solutions aux problèmes soulevés par NETSOLVE. Ensuite, nous présenterons diverses expériences qui nous ont permis d'étudier l'impact de la hiérarchie sur la propagation de requêtes ainsi que de dégager quelques règles pour la construction d'une hiérarchie performante. Enfin, nous donnerons quelques perspectives de développement pour DIET.

4.2 NETSOLVE

4.2.1 Présentation générale

NETSOLVE [24, 87] est un environnement à base de serveurs de calcul développé à l'Université du Tennessee, Knoxville. Cet environnement permet de démarrer des calculs à distance sur une plate-forme de grid computing en utilisant le modèle client–agent–serveurs. De nombreuses interfaces utilisateurs existent, NETSOLVE pouvant être appelé depuis des codes C, Fortran, Java, Matlab, SCILAB, ou même depuis une page web.

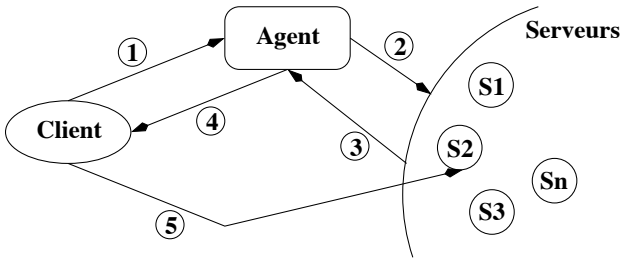


FIG. 4.1 – Architecture et fonctionnement de NETSOLVE.

Le fonctionnement d'une session NETSOLVE peut être représenté par la figure 4.1. Tout d'abord, l'agent est démarré. Puis des serveurs s'enregistrent auprès de lui, chacun envoyant une liste des problèmes qu'il est capable de résoudre ainsi que des informations concernant la vitesse et la charge de la machine sur laquelle ce serveur s'exécute ainsi que les caractéristiques du réseau entre ce serveur et l'agent, *i.e.*, latence et bande passante. Une fois cette étape d'initialisation effectuée, un client peut contacter l'agent pour lui soumettre une demande de résolution de problème (1). L'agent va alors interroger l'ensemble des serveurs proposant une méthode pour résoudre ce problème afin d'estimer le coût de cette résolution (2). Cette estimation est basée sur une formule analytique simple décrivant le comportement asymptotique de la fonction, sur la taille des données impliquées, et sur la charge de la machine et du réseau au moment de l'interrogation. L'ordonnanceur peut alors sélectionner un ensemble de serveurs adaptés au problème soumis en ne conservant que ceux dont les temps de résolutions sont les meilleurs (3). Cette liste de serveurs est ensuite envoyée au client (4). Ce dernier cherche à contacter les serveurs de cette liste et envoie ses données à traiter au premier serveur qu'il parvient à joindre (5). Le problème soumis est alors résolu sur la ressource de calcul choisie. Dans le mode de fonctionnement le plus

courant de NETSOLVE, le résultat de la résolution est systématiquement renvoyé au client dès la fin du calcul.

4.2.2 Optimisation de NETSOLVE

Dans le cadre du développement de SCILAB //, la première version basée sur des serveurs de calculs à été réalisée à partir de NETSOLVE. Comme nous l'avons vu au chapitre 1, une interface SCILAB–NETSOLVE a d'abord été réalisée. Puis en étudiant le code de NETSOLVE de manière plus précise, nous avons constaté que plusieurs optimisations étaient possibles. La première de ces optimisations concerne la persistance des données et la seconde l'évaluation des performances des serveurs.

4.2.2.1 Ajout de la persistance

Selon les séquences d'opérations effectuées, il arrive souvent que les résultats d'un calcul soient utilisés par le calcul suivant. Comme nous l'avons dit précédemment, NETSOLVE ne permet pas, dans son mode de fonctionnement normal, de laisser des données sur un serveur après un calcul ni d'effectuer des redistributions entre serveurs lorsque cela s'avère intéressant. Ce va-et-vient des données entre un client et des serveurs peut induire des surcoûts de communication importants, surtout lorsque NETSOLVE est déployé sur un réseau d'échelle nationale ou internationale où les coûts de communication sont très élevés. Ce problème a été en partie traité par l'ajout du séquencement de requêtes [4]. Grâce à cette fonctionnalité et des directives de début et de fin de séquence, un utilisateur peut indiquer à NETSOLVE que les appels compris entre ces deux directives doivent être traités d'une manière particulière. L'arbre d'appel des différentes instructions comprises entre ces deux directives est tout d'abord construit, puis une analyse statique du code est effectuée. Cette technique permet de ne pas rapatrier sur le client des données intermédiaires dont la durée de vie est limitée à cette séquence d'appels. Toutefois, cette nouvelle fonctionnalité ne permet de laisser des données persistantes que sur un seul serveur.

La notion de persistance de données telle qu'elle à été implantée dans NETSOLVE par Emmanuel Jeannot permet à l'utilisateur de distribuer ses données sur différents serveurs. De plus, une fois une donnée distribuée, celle-ci peut être déplacée entre deux serveurs pour les besoins d'un calcul. Pour réaliser cette implantation, le code de NETSOLVE à été modifié, mais de manière transparente, puisque des codes NETSOLVE existants fonctionnent toujours, à une recompilation près. Désormais, lorsqu'un serveur a terminé le calcul qui lui était assigné, il se place en attente d'ordres provenant du client. Ces ordres permettent d'indiquer au serveur de s'arrêter et donc d'effacer les données locales, ou de retourner au client une ou plusieurs données, ces données pouvant être des paramètres d'entrée ou des résultats du problème résolu. Cette fonctionnalité peut alors être utilisée par un client ou l'agent pour optimiser l'ordonnancement des tâches sur la plate-forme de grid computing. La figure 4.2 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 (resp. B_r et C_r) est la partie réelle d'une matrice complexe A (resp. B et C), A_i (resp. B_i et C_i) est la partie imaginaire d'une matrice complexe A (resp. 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 obtenues sont 1.77 fois meilleures pour des matrices de dimension 1024 qu'en utilisant la version « classique » de NETSOLVE.

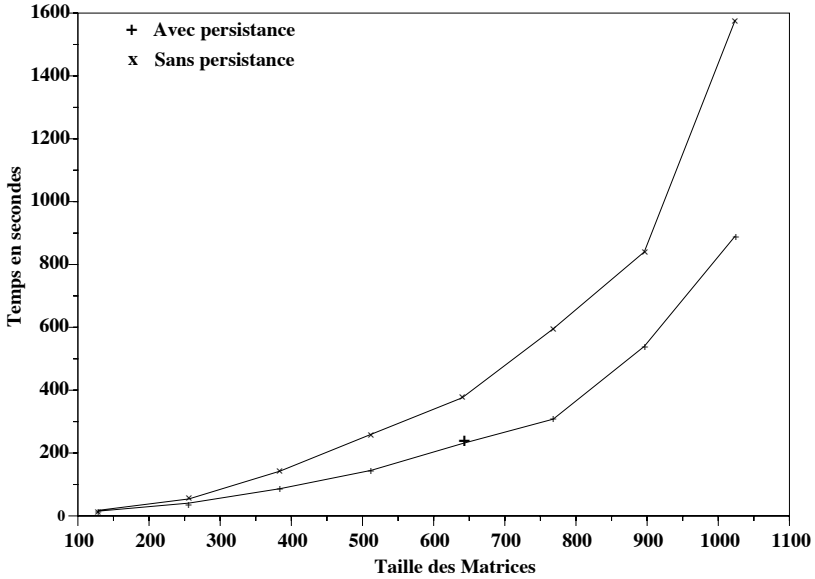


FIG. 4.2 – Produit de matrices complexes entre Nancy et Bordeaux en utilisant NETSOLVE.

4.2.2.2 Évaluation précise des performances

La seconde optimisation apportée à NETSOLVE concerne l'estimation des performances des serveurs permettant à l'ordonnanceur de construire la liste à retourner au client. En effet, NETSOLVE effectue deux simplifications qui empêchent d'avoir une vision réaliste de la plate-forme d'exécution. Tout d'abord, pour l'estimation des coûts de transfert des données entre un client et un serveur, NETSOLVE se base sur les caractéristiques du lien de communication entre ce serveur et l'agent. Or, selon la localisation de chacune de ces entités, des différences non négligeables peuvent apparaître. De plus, le temps d'exécution des routines de calcul est estimé en se basant sur une formule simple ne tenant compte que des performances théoriques de la machine et de la taille des données. Nous avons vu dans le chapitre 2 que ce type d'estimations ne permettait pas de tenir compte de la hiérarchie mémoire des machines ni des optimisations de codes. C'est pour ces raisons que l'ordonnanceur de NETSOLVE a été modifié pour utiliser FAST pour effectuer ses prédictions de performances.

4.3 DIET : Distributed Interactive Engineering Toolbox

Un des problèmes majeurs, que nous n'avons pas encore évoqué, des environnements à base de serveurs de calculs tels que NINF et NETSOLVE, est que l'agent contenant l'ordonnanceur est centralisé. Cet agent peut donc devenir un goulot d'étranglement dans une situation concrète où beaucoup de clients souhaitent accéder à plusieurs serveurs. De plus, les topologies réseau actuelles étant fortement hiérarchiques, la localisation de l'ordonnanceur à un impact important sur les performances de l'ensemble de la plate-forme de grid computing. Pour résoudre ces problèmes, nous pensons qu'une approche hiérarchique est la plus adaptée. C'est pourquoi, dans le cadre des projets GASP du Réseau National des Technologies Logicielles (RNTL)[1] et GRID-ASP

[1]URL : http://www.ens-lyon.fr/~desprez/GASP/

de l'ACI GRID[2], nous avons entreprit de proposer une approche portable et extensible pour le développement d'applications selon le modèle ASP sur une architecture distribuée de type grille de calcul. Le logiciel développé se nomme DIET. Une des idées fortes est de porter des applications de grande envergure sur un environnement développé dans le cadre de nos projets de recherche tout en réutilisant des logiciels existants, portables et « standards ». Ces applications proviennent de domaines d'applications aussi variés que la physique, la chimie, la géologie ou encore la micro–électronique.

4.3.1 Applications cibles de DIET

4.3.1.1 Modélisation numérique de terrain

Le Laboratoire des Sciences de la Terre (LST) de l'École Normale Supérieure de Lyon dispose d'un programme parallèle pour le calcul de Modèles Numériques de Terrains (MNT) à partir de deux vues stéréoscopiques, typiquement des images satellites . Cette technique consiste à mettre en correspondance le plus de points possibles effectivement présents dans les deux images. Puis, en utilisant les propriétés géométriques des deux prises de vues ainsi que la disparité de position entre les pixels représentant un même point, les altitudes sont calculées.

L'interface entre DIET et l'application de calcul de MNT est développée sous la forme d'un page Web où l'utilisateur doit saisir un certain nombre d'informations relatives à ses données. Ces informations concernent principalement la localisation des fichiers des images stéréoscopiques et des fichiers de données complémentaires. Dans le cas où l'utilisateur ne dispose pas déjà des images stéréoscopiques qui l'intéressent, il sera possible, dans une version ultérieure, de fournir une description formelle de la région à modéliser. DIET se chargera alors de localiser, en plus du serveur de calcul, un serveur d'images géographiques pour récupérer les images nécessaires.

Le résultat fourni par l'application est une série de fichiers binaires représentant le MNT à diverses résolutions ce qui permet une reconstruction pyramidale du maillage complet en minimisant les erreurs à chaque niveau de résolution.

Les besoins spécifiques de cette application qui doivent être pris en considération dans le cadre du développement de DIET sont une consommation mémoire importante, une grande puissance de calcul nécessaire et un besoin d'une interface de visualisation.

4.3.1.2 Simulation de composants électroniques

Les travaux de l'Institut de Recherche en Communications Optiques et Micro-ondes (IRCOM) de Limoges concernent la simulation de composants électroniques. Leur simulateur de circuit en équilibrage harmonique utilise le domaine fréquentiel pour décrire la partie (ou sous-circuit) linéaire du système à analyser, et n'utilise le domaine temporel que pour la description des éléments (ou sous-circuit) non-linéaires. Lors de la recherche itérative d'une solution, une transformée de Fourier permet, à chaque itération, de raccorder les variables à l'interface entre sous-circuits linéaire et non-linéaire. Les points critiques de cette application résident dans les appels successifs au simulateur physique qui constitue la majeure partie du temps de calcul du simulateur. Ce temps croît donc de manière quasi linéaire avec le nombre de transistors du circuit. D'après cette constatation, une première parallélisation à « gros grain » peut être mise en place. Le client « simulateur de circuit » résout à l'aide d'une méthode itérative de Newton-Raphson l'équation d'équilibrage harmonique provenant des lois de Kirchoff à chaque nœud du circuit

[2] URL : http://www.ens-lyon.fr/~desprez/GRIDASP/

global. Afin de calculer les valeurs des courants non-linéaires et de leurs dérivées en fonction des commandes appliquées, le client envoie des requêtes aux serveurs « simulateurs physiques » qui résolvent quant à eux les équations de transport pour chaque doigt d'émetteur et collecte les valeurs des courants ainsi que de leurs dérivées afin de construire la matrice Jacobienne nécessaire à « l'algorithme circuit » de convergence. Cette approche parallèle permet de diminuer le temps de calcul quasiment par le nombre de transistors du circuit à condition que l'on dispose d'autant de processeurs que de transistors.

Les besoins spécifiques de cette application concernent la recherche d'un solveur creux à haute performance et la gestion de codes de simulation propriétaires.

4.3.1.3 Calcul d'Hyper Surfaces d'Énergie Potentielle

L'objectif de cette application du laboratoire de Structure et Réactivité des Systèmes Moléculaires Complexes (SRSMC) de l'Université Henri Poincaré de Nancy est d'effectuer le calcul distribué des points d'une surface en chimie quantique. Cette application peut se décomposer en deux phases. Tout d'abord, le client interroge une base de données des points déjà calculés pour déterminer si les points recherchés sont d'ores et déjà connus. Si c'est le cas, ces points sont directement retournés au client. Dans le cas contraire, le client contacte, via DIET, un serveur de calcul sur lequel s'exécute un logiciel de chimie quantique. L'interface entre le client, la base de données des points connus et DIET est développée sous la forme d'une page web.

Cette application a comme besoin spécifique l'utilisation d'une base de données relationnelles, de type MySQl.

4.3.1.4 Dynamique moléculaire

La dernière application cible de DIET étudiée dans le cadre du projet GRID-ASP appartient au domaine de la dynamique moléculaire. Cette application est développée conjointement par trois laboratoires : les laboratoires de Physique et de mathématiques appliquées de l'Université Claude Bernard de Lyon, et le laboratoire de Physique de l'École Normale Supérieure de Lyon. Le but de l'application est de simuler des très grands systèmes, de l'ordre du million de particules, par dynamique moléculaire, *i.e.*, résolution microscopique des équations du mouvement des atomes en vue d'une comparaison avec les équations continues de Navier-Stokes dans des conditions où la validité de celles-ci n'est pas triviale. Le code utilisé est écrit en Fortran et utilise la bibliothèque de passage de messages MPI. Son principe est d'utiliser une décomposition spatiale du système, déjà largement testée sur de nombreuses machines parallèles. L'utilisateur spécifie la configuration de départ et les paramètres d'interaction, et le programme intègre les équations du mouvement pendant un temps prédéterminé. Les données recueillies à la fin de calcul sont les trajectoires des particules ou les champs de vitesse à des instants sélectionnés.

De même que pour l'application de micro-électronique, cette application doit gérer des codes publics et privés. De plus, il est nécessaire de générer des traces disques des trajectoires des molécules pour en réaliser une exploitation post-mortem. La gestion du stockage de données sur supports de mémoire secondaire devra donc également être étudiée dans DIET.

4.3.2 Architecture de DIET et outils associés

L'originalité de l'environnement DIET est d'être basé sur une hiérarchie d'agents plutôt que sur un seul composant centralisé. L'architecture de DIET est présentée dans la figure 4.3 et détaillée dans les paragraphes suivants.

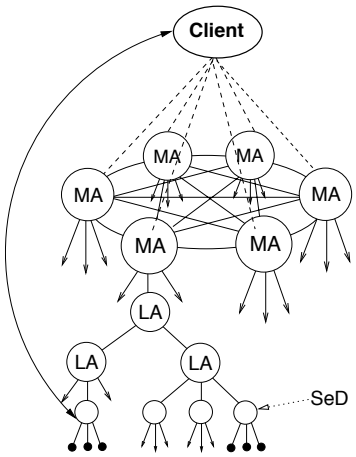


FIG. 4.3 – Architecture hiérarchique de DIET.

4.3.2.1 Hiérarchie d'agents

La particularité principale de DIET est de ne pas avoir d'ordonnanceur, et donc d'agent, centralisé comme la plupart des environnements à base de serveurs de calcul. DIET utilise une hiérarchie d'agents, à la fois pour répartir la charge et éviter le phénomène de goulot d'étranglement, mais aussi pour refléter l'aspect hiérarchique des réseaux sous-jacents. Cette hiérarchie est constituée de deux types de composants : les *Master Agents* (MA) et les *Local Agents* (LA).

Les MA se situent au sommet de la hiérarchie et sont connectés par un réseau complet. Leur nombre est par conséquent limité. Ce choix est justifié par la première plate-forme cible de DIET : le réseau à Vraiment Très Haut Débit (VTHD)[3] reliant, entre autres, les Unités de Recherche de l'INRIA. L'objectif de déploiement de DIET sur VTHD est d'avoir un MA par Unité de Recherche. Par la suite, d'autres types de connexions que le réseau complet seront étudiés. Les MA constituent les points d'entrées de DIET à travers lesquels les utilisateurs pourront accéder aux services de calcul. Chacun des MA est la racine d'un arbre de LA, les serveurs de calculs étant les feuilles de cet arbres. Le rôle des LA est de transmettre les requêtes soumises par l'utilisateur depuis un MA jusqu'au serveurs. Chaque agent a la connaissance des données distribuées dans sa descendance et, pour chaque type de problème, du nombre de serveurs sachant résoudre ce problème.

Le MA est l'entité responsable de toutes les décisions concernant le choix des serveurs. Ces décisions sont prises dès lors que toutes les informations nécessaires, concernant les capacités et les disponibilités des serveurs, sont remontées à travers la hiérarchie. Les techniques mises en œuvre pour choisir le ou les serveurs les plus adaptés à la résolution du problème soumis par un client seront présentées dans le paragraphe 4.3.3.4.

4.3.2.2 Serveurs

Dans DIET, un serveur de calcul est encapsulé dans un *Server Daemon* (SeD). Ce processus est typiquement situé sur le point d'entrée d'une machine parallèle et gère les différentes ressources de calcul et de communication du serveur. Les informations utiles à l'ordonnanceur stockées sur

[3] URL :http://www.vthd.org

un SeD sont la liste des données distribuées sur le serveur correspondant, la liste des problèmes DIET pouvant être résolus sur ce serveur et toutes les informations concernant sa charge, *e.g.*, espace mémoire disponible, quantité de ressources disponibles, etc.

Lors de son démarrage, un SeD publie la liste des problèmes qu'il est capable de résoudre auprès de son LA père. Pour chacun de ces problèmes, le SeD fournit également un mécanisme de prédiction de performances sur lequel nous reviendrons dans le paragraphe 4.3.2.4.

Un serveur de calcul DIET peut aussi bien être séquentiel que parallèle. Ainsi, dans le cas de serveurs résolvant des problèmes d'algèbre linéaire dense, les calculs pourront être effectués en appelant les bibliothèques BLAS, pour le cas séquentiel, et SCALAPACK, pour le cas parallèle. Dans ce dernier cas, plusieurs configurations, impliquant différents nombres de processeurs connectés selon diverses topologies, pourront être employées. Le choix d'utiliser ou non le parallélisme pour la résolution dépendra de la taille des données impliquées et bien entendu des coûts respectifs des différentes solutions proposées. En effet, si la somme des coûts de distribution des données sur la machine, de calcul sur cette configuration et de rapatriement des résultats est supérieure au coût d'une exécution séquentielle, le choix de configuration devient alors trivial.

Deux possibilités existent pour l'implantation de serveurs parallèles dans DIET. La première consiste à réserver un certain nombre de processeurs de calcul au démarrage du SeD, *e.g.*, sous la forme d'une commande de type `mpirun`. Les phases de distribution et de calcul peuvent alors être comprises dans le code du SeD. L'inconvénient majeur de cette solution est l'inactivité des processeurs réservés entre deux requêtes. La seconde possibilité consiste à n'effectuer la réservation des processeurs de calcul qu'au moment de la réception de la requête du client. Cela permet d'être beaucoup plus souple quant au choix de configuration, puisque la limite du nombre de processeurs devient celle de la machine parallèle. En revanche, cette technique soulève des problèmes de passage de données du SeD au processus exécutant l'ensemble du calcul.

4.3.2.3 Client

Un client DIET peut en réalité être vu comme une interface pour l'invocation de la plate-forme. À terme, plusieurs types de clients pourront se connecter à DIET : depuis une page web, un PSE comme SCILAB ou plus classiquement depuis un programme écrit en C.

4.3.2.4 Identification de problèmes et prédiction de performances

Deux modules supplémentaires sont utilisés par DIET, l'un pour identifier les problèmes soumis par les clients et l'autre pour prédire les performances des implantations correspondant à ces problèmes. Le module de prédiction de performances est l'outil FAST, présenté dans le chapitre 2. FAST et son extension parallèle permettent aux serveurs DIET d'estimer le temps d'exécution d'une routine pour chacune des configurations possibles sur ce serveur. De plus, la partie de FAST basée sur NWS rend possible l'estimation des coûts de transfert d'une donnée du client vers un serveur ou même entre deux serveurs.

Comme nous l'avons indiqué précédemment, lorsqu'un SeD s'enregistre auprès d'un agent DIET, il fournit la liste des problèmes qu'il est capable de résoudre. Dans une première implantation de DIET, cette liste correspond à un formalisme bien précis que nous allons détailler. Les offres de services proposées par les serveurs sont regroupées en classes et sous-classes. Chaque service est défini à la fois par son *nom complet* et par la description de son *profil*, ce couple devant être utilisé par le client pour soumettre ses problèmes. Le nom complet s'apparente à un chemin d'accès à un fichier. Si l'on considère, par exemple, le cas d'une multiplication de matrices denses, un SeD sur lequel la bibliothèque BLAS est installée publiera les services suivants :

matmult/ddmatmult	IN DoubleDenseMatrix, INOUT DoubleDenseMatrix
matmult/dgemm	IN Char, IN Char, IN Integer, IN Integer, IN Integer, IN Double, IN DoubleDenseMatrix, IN Integer, IN DoubleDenseMatrix, IN Integer, IN Double, IN Integer, INOUT DoubleDenseMatrix

Le service matmult/dgemm fait directement référence à la routine `dgemm` de la bibliothèque BLAS, alors que le service matmult/ddmatmult correspond à une description plus générique d'un produit de matrices denses.

Le profil d'un service, présenté par la figure 4.4, consiste en un tableau de descripteurs de données, regroupés selon leur rôle dans le profil : *in*, *inout* ou *out*. Trois variables (`last_in`, `last_inout` et `last_out`) permettent de délimiter ces ensembles au sein du tableau.

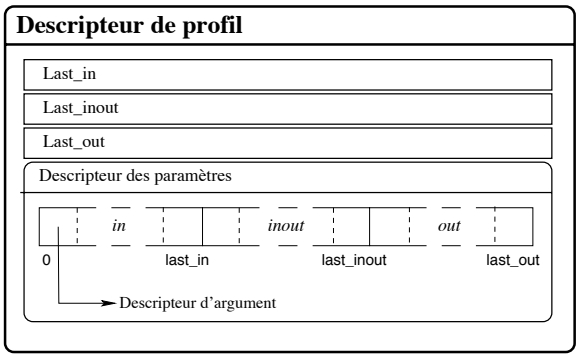


FIG. 4.4 – Structure d'un profil.

Une liste complète des services publiés par l'ensemble des serveurs DIET sera maintenue dynamiquement. C'est en consultant cette liste qu'un utilisateur de DIET pourra écrire des requêtes correspondant à des problèmes existants. L'identification de problèmes utilisera également cette liste lors de la propagation des requêtes dans la hiérarchie DIET. La mise en correspondance entre description de problème et implantation spécifique, *i.e.*, un appel de routine, sera effectuée au niveau des SeD.

Dans une évolution future de DIET, la réalisation du module SLIM (Scientific Libraries Metaserver) pourra permettre à un utilisateur de DIET de spécifier ses problèmes dans une syntaxe plus proche du langage naturel. Ce module sera alors chargé de faire la correspondance entre cette spécification et les implantations disponibles sur la plate-forme.

4.3.2.5 Couche de communication

Les environnements à base de serveurs de calcul peuvent typiquement être implantés en utilisant une couche de communication classique par sockets. NINF et NETSOLVE sont ainsi implantés de cette manière. Cependant, plusieurs problèmes de cette approche ont été soulignés dans [79], comme par exemple le manque de portabilité ou la limitation due au nombre de sockets ouvertes au même instant. Notre souhait étant d'implanter et déployer un environnement distribué efficace à une plus grande échelle, nous avons choisi une autre couche de communication.

Des environnements objets tels que *Java*, *DCOM* ou CORBA ont prouvé être une bonne base pour la construction d'applications gérant l'accès à des services distribués. Ils fournissent non

seulement des communications transparentes en milieu hétérogène, mais offrent également un cadre pour le déploiement à grande échelle.

La norme CORBA, définie par l'*Object Management Group* (OMG), propose une interface standard de haut niveau pour la programmation d'applications distribuées à base d'objets. Un système respectant cette norme est construit autour d'un bus de communication entre objets, l'*Object Request Broker* (ORB). Les communications sont réalisées sous forme d'appel de méthodes entre des objets pouvant se trouver sur différentes machines. La transparence de ces communications sur un réseau hétérogène est assurée par un compilateur *IDL* (*Interface Description Language*) fourni avec chaque ORB. Ce compilateur génère automatiquement le code permettant l'encapsulation et la restitution des types de données complexes. CORBA utilise le protocole *CDR* pour l'encapsulation des types simples. Ce protocole est plus performant que *XDR*. Les données ne subissent en effet de codage que lorsque cela est nécessaire. Le typage dynamique et le téléchargement de la description d'un type sont également supportés par le code généré. De plus, ce niveau de transparence autorisé par CORBA n'affecte pas les performances. La plupart des implantations de CORBA disposent en effet d'une couche de communication fortement optimisée dont les performances se rapprochent de celles des sockets [2].

Différentes implantations de la norme existent, chacune disponible sur de nombreuses plates-formes. L'interopérabilité entre ces implémentations est garantie par le *General Inter ORB Protocol* (GIOP). Les applications basées sur CORBA bénéficient par conséquent d'une bonne portabilité. Du point de vue du développeur, CORBA peut être utilisé avec plusieurs langages de programmation, dont C, C++ et Java.

Nous pouvons donc conclure que des systèmes CORBA constituent une des alternatives pour le développement d'un environnement à base de serveurs de calcul. Notre première implantation de DIET est basée sur *OmniORB*, une implantation libre de CORBA qui offre de bonnes performances de communication.

4.3.3 Initialisation et fonctionnement d'une plate-forme DIET

Dans ce paragraphe, nous allons détailler le processus d'initialisation d'une plate-forme DIET ainsi que le déroulement d'une session de calcul. Dans un but pédagogique, nous limiterons cette présentation à une plate-forme ne comprenant qu'un unique MA.

La figure 4.5 présente les différentes étapes de l'initialisation d'une plate-forme DIET. La construction de cette plate-forme s'effectue de manière hiérarchique, chaque composant s'enregistrant auprès de son père. Le MA est la première entité à être démarrée (1). Ce processus se met alors en attente de demande de connections issues de LA et de requêtes en provenance de clients.

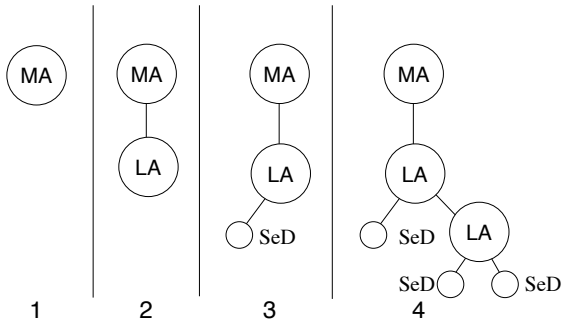


FIG. 4.5 – Initialisation d'une plate-forme DIET.

Lorsqu'un LA est lancé, il s'enregistre auprès de l'agent se trouvant au niveau supérieur de la hiérarchie. Dans (2), l'enregistrement s'effectue directement auprès du MA. À cette étape de l'initialisation, deux types de composants peuvent se connecter au niveau le plus bas de la hiérarchie : un SeD (3) ou un autre LA (4), ce qui ajoute un niveau supplémentaire dans cette branche. Lors de l'enregistrement d'un SeD, celui-ci publie la liste des problèmes qu'il est capable de résoudre. Cette liste est alors diffusée le long de la branche, jusqu'au MA.

Tout ce processus d'initialisation est effectué par des administrateurs. En effet, il est pour l'instant nécessaire que chaque composant connaisse l'agent de niveau supérieur auquel se rattacher. Cependant, la structure hiérarchique de DIET permet de répartir cette charge administrative, chaque site déployant et gérant sa hiérarchie locale. Les seules informations « globales » à récupérer ne sont alors nécessaires qu'à des niveaux élevés, donc peu nombreux, de la hiérarchie, *e.g.*, pour l'interconnexion de plusieurs sites. Cette approche hiérarchique présente également l'avantage de pouvoir apporter des changements de configuration locaux dans les parties basses de la hiérarchie sans interférer avec l'ensemble de la plate-forme.

Une fois la plate-forme initialisée, un client peut soumettre des requêtes à DIET. Pour cela, ce client doit tout d'abord interroger le *ReDirector* (ReD) pour obtenir la référence du MA auquel se connecter. Le concept de ReD peut être considéré à différents niveaux d'abstraction. Des implantations possibles sont : une simple page web répertoriant les références de tous les MA disponibles ; un processus lancé sur une machine donnée dont le fonctionnement serait très proche de celui du services des « Pages Jaunes » de CORBA ; ou enfin le déploiement d'une plate-forme *pair-à-pair* reliant les clients et les MA. Quelque soit l'implantation choisie pour ce concept de redirection, la phase de connexion doit être aussi transparente que possible pour l'utilisateur. Un simple appel, avec un fichier de configuration comme paramètre, permet ainsi à un client DIET de se connecter au MA le plus proche.

Dans la plupart des cas, la plate-forme DIET comprendra plusieurs serveurs capables de résoudre un même problème. En revanche, les données distribuées sur les serveurs ne seront présentes qu'en un unique exemplaire. La figure 4.6 présente un exemple de session DIET où le client souhaite résoudre le problème `F` impliquant les données `A` et `B`.

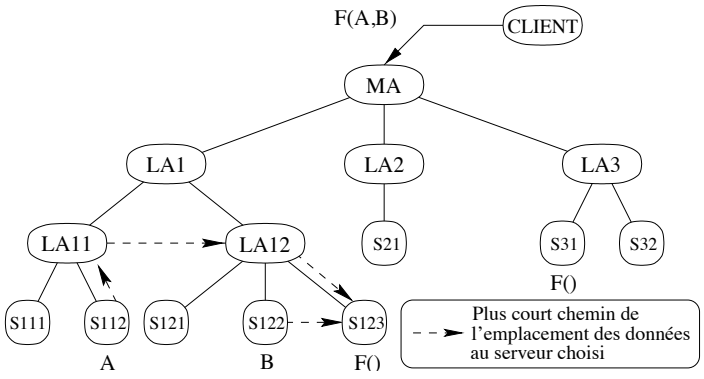


FIG. 4.6 – Exemple de soumission de problème dans DIET.

Pour pouvoir effectuer un classement des différents serveurs de la plate-forme sachant résoudre ce problème, le MA doit récupérer les prédictions de temps d'exécution pour chacun des candidats. Pour cela, les cinq étapes suivantes sont nécessaires.

4.3.3.1 Propagation d'une requête

Lorsque le MA reçoit une requête de la part d'un client, il construit une *structure de requête*. Cette structure est composée de deux champs : le `nomDeProblème`, et une liste d'`attributs` des données impliquées, qui comprennent des détails concernant leurs tailles et propriétés de manière à estimer les temps de calcul et de communication. Le MA propage alors la structure correspondant à la requête du client dans la hiérarchie jusqu'aux serveurs qui proposent la résolution du problème soumis ou qui disposent d'une des données impliquées. Un élagage est effectué à chaque niveau de l'arbre, chaque LA intermédiaire sachant dans quelle(s) branche(s) se trouvent des serveurs à même de satisfaire la requête ou possédant une donnée. La propagation ne se fait donc que dans les branches « utiles ». Pour conserver l'information relative aux branches visitées, chaque LA marque ses fils atteints par la propagation de la structure, et se met en attente de leurs réponses.

4.3.3.2 Estimations des coûts de calcul et de communication

Chaque serveur contacté initie sa réponse en construisant une *structure de reponse* qu'il renverra ensuite à son père. Cette structure se compose de trois champs :

`monNom` est l'identifiant unique du composant qui émet cette réponse ;

`données` est un tableau possédant une entrée pour chacune des données impliquées dans le calcul. Chacune de ces entrées comporte deux champs :
- `localisation` est l'identifiant du serveur qui possède la donnée s'il est connu (ce champ a une valeur nulle sinon) ;
- `tempsVersMoi` est l'estimation du temps de transfert de la donnée depuis le composant possédant la structure au composant construisant la réponse, si cette localisation est connue. Ce temps est obtenu en appelant FAST.

`calcul` est un tableau qui contient une entrée pour chaque serveur, capable de satisfaire la requête, connu à un niveau de l'arbre donné. Pour chacun de ces serveurs, trois informations sont stockées :
- `nom` est l'identifiant du serveur ;
- `tCalc` est le temps de complétion estimé de la requête. Chaque SeD concerné appelle FAST pour obtenir ces estimations. Les paramètres fournis à FAST concernant les données sont extraits de la structure de requête ;
- `tComm` est un tableau contenant l'estimation du temps pour transférer chaque donnée impliquée vers ce serveur.

4.3.3.3 Agrégation des réponses

Chaque LA intermédiaire collecte les structures de réponses provenant de sa descendance et les agrège pour construire sa propre structure de réponse. Les temps de transferts des données sont calculés dynamiquement lors de la remontée de la structure de réponse vers le MA. La précision de l'estimation de ces temps de transfert suppose que la hiérarchie d'agents soit déployée de manière à refléter la topologie du réseau sous-jacent. Sous cette condition, l'addition des valeurs contenues dans `tempsVersMoi` permet d'obtenir une précision suffisante. De plus, le chemin choisi sera le plus court parmi ceux surveillés. La figure 4.7 présente l'algorithme d'agrégation utilisé par les LA.

pour chaque donnée `D` **faire**
  si aucun de mes descendants ne possède `D` **alors**
    `TempsVersMoi` = 0
  sinon si un de mes fils référence `D` **alors**
    `TempsVersMoi` = `TempsVersMoi` pour ce fils + temps pour que ce fils m'envoie `D`
  sinon si `D` n'est connu par aucun de mes fils **alors**
    si Il existe un serveur `S` dans mon sous arbre sachant résoudre le problème **alors**
      `D` sera envoyée à `S` en passant par moi si `S` est choisi.
      ⇒ J'augmente le `tComm` de `D` pour tous les serveurs capables
    sinon
      `D` sera envoyée à un serveur `S` par un chemin où je ne figure pas.
      ⇒ **si** le calcul de `tComm` n'est pas terminé pour certains de mes descendants **alors**
      je peux le faire
    fin si
  fin si
fin pour

FIG. 4.7 – Algorithme d'agrégation des réponses par un LA.

De plus, les LA qui reçoivent une ou plusieurs réponses positives de leur descendance sélectionnent un ensemble de serveurs parmi les plus rapides et transmettent cette sélection au niveau supérieur. Ce principe est appliqué à chaque niveau, jusqu'au MA.

4.3.3.4 Choix des serveurs

Dès lors que le MA a collecté toutes les réponses provenant de ses fils, il peut effectuer l'ultime sélection parmi les serveurs proposés et renvoyer une liste de références sur ces serveurs au client. Dans sa version actuelle, le MA sélectionne les serveurs les plus rapides. Il sera possible, dans une version ultérieure, d'ajouter des contraintes à ce problème de sélection. Le MA pourra ainsi tenir compte dans son choix des moyens financiers engagés par le client, dans l'hypothèse où l'accès à certains serveurs est payant, de la possibilité pour certains serveurs d'être réservés, etc.

Il est important de considérer le cas où est un serveur est sélectionné à deux reprises et où le premier calcul n'a pas encore démarré au moment de la soumission du second. Ce cas peut se produire lorsque la pénalité affecté au serveur lors de l'affectation du premier calcul n'est pas prise en compte dans l'estimation du temps d'exécution du second calcul sur ce serveur. C'est un problème classique d'évaluation dynamique de performances, et un algorithme basé sur un système de contrats est en cours de développement. Cela permettra de vérifier si l'estimation est toujours significative à l'instant de la connexion entre le client et le serveur.

4.3.3.5 Soumission du problème

À la réception de la liste fournie par le MA, le client DIET cherche à contacter l'un des serveurs. Une fois la connexion établie, le client envoie ses données locales et spécifie si les données issues de la résolution doivent être conservées distribuées pour les opérations suivantes, ou lui être retournées. Le transfert des données persistantes, *i.e.*, déjà distribuées sur la plate-forme DIET, vers le serveur choisi est alors effectué.

4.4 Évaluation de la hiérarchie DIET

Un premier prototype de DIET à été développé, au sein du Laboratoire d'Informatique de Franche Comté, afin de valider notre approche hiérarchique et d'évaluer les performances de l'algorithme de diffusion de requêtes proposé. Toutes les expériences ont été effectuées sur une plate-forme n'impliquant qu'un unique MA et une hiérarchie de LA. Nous verrons tout d'abord comment, dans le cas où le réseau sous-jacent est lent, l'utilisation d'une hiérarchie de LA intermédiaires peut diminuer considérablement le temps de propagation d'une requête. Puis, nous étudierons le coût de l'ajout de serveurs à une plate-forme DIET ne disposant que d'un seul LA. Nous comparerons ensuite deux architectures de même profondeur, l'une linéaire et l'autre déployée selon un arbre binaire. Cela nous permettra d'évaluer l'impact de l'introduction du parallélisme dans le traitement des requêtes. Enfin, nous montrerons comment il est possible d'ajouter des serveurs à une hiérarchie existante sans augmenter les coûts de traitement. Pour cela, nous étudierons les choix à suivre pour définir correctement la structure de l'arbre.

4.4.1 Impact de la hiérarchie sur la propagation de requêtes

Cette expérience a été réalisée dans le but de montrer les bénéfices de l'approche hiérarchique. En effet, dans le cas où la plate-forme sur laquelle est déployé DIET comporte un lien de communication à très faible débit, il semble intéressant d'utiliser un LA intermédiaire afin de réduire le nombre de messages transitant par ce lien. Ceci peut être vérifié sur la figure 4.8 qui présente les deux configurations utilisées dans cette expérience. Par cette expérience, nous avons étudié l'évolution du temps de propagation d'une requête en fonction du nombre de SeD pour chacune des deux configurations de la figure 4.8.

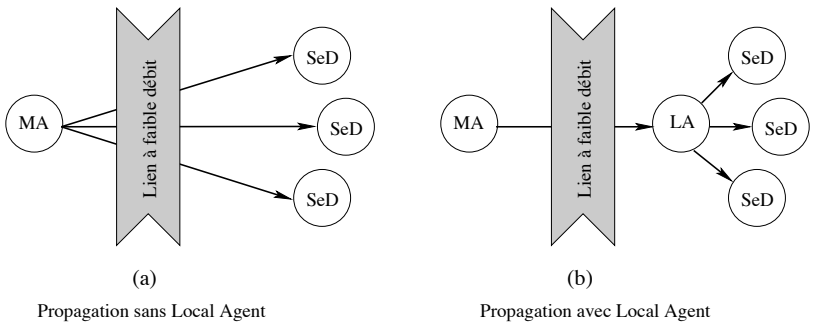


FIG. 4.8 – Impact de la hiérarchie sur la propagation de requête.

Le lien limitant a une bande passante de 2 Mo/s. De plus, cette bande passante est partagée avec d'autres applications. Le réseau local sur lequel sont déployés les SeD (et le LA de la configuration présentée dans la figure 4.8 (b)) est un réseau Fast Ethernet possédant plusieurs commutateurs. Le temps de traitement d'une requête par un SeD étant nettement inférieur que le temps de communication sur le lien à faible débit, plusieurs SeD sont co-alloués sur une même station de travail. Le MA et le client sont démarrés sur une seule machine de l'autre coté du lien limitant. La figure 4.9 montre le résultat de cette expérience. Pour chaque nombre de serveurs, le temps reporté est une moyenne des temps de propagation obtenus pour cinquante clients lancés séquentiellement.

Les résultats obtenus sont quasi linéaires pour chacune des deux configurations. Dans le cas

où un LA est utilisé, le temps de propagation d'une requête est de trois (pour 42 SeD) à quatre fois (pour 72 SeD) moins élevé que celui obtenu sans LA. Une régression linéaire montre que la pente est quatre fois plus importante lorsqu'aucun LA n'est déployé.

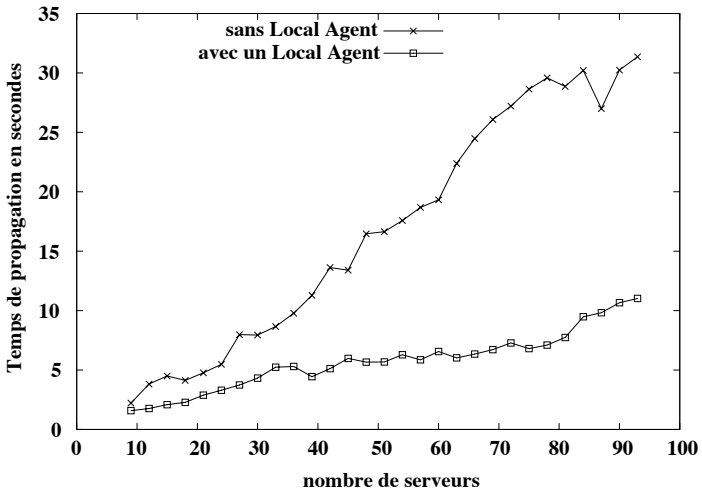


FIG. 4.9 – Temps de propagation d'une requête avec ou sans Local Agent.

Cette expérience souligne l'importance de l'utilisation d'une hiérarchie d'agents dans le cadre du développement d'environnements à base de serveurs de calculs. En effet, les environnements existants, tels que NINF et NETSOLVE, se placent dans une configuration où l'impact d'un seul lien à très faible débit peut affecter le temps de réponse de l'ensemble de la plate-forme.

4.4.1.1 DIET sur réseau local

Les expériences suivantes ont été réalisé sur un réseau Fast Ethernet commuté local. Ce réseau étant dédié, les phénomènes de contention ne sont pas pris en compte. Pour chacune de ces expériences, une hiérarchie DIET a été déployée. Il est à noter que les SeD lancés sont capables de s'enregistrer auprès d'un agent et de répondre aux requêtes, mais n'effectuent en revanche aucun calcul. Une autre hypothèse majeure est que le problème soumis par le client peut être résolu par tous les serveurs. Cette hypothèse reste réaliste puisque, par exemple, un problème d'algèbre linéaire dense pourra être résolu sur toute machine où les BLAS sont installées. La conséquence de cette hypothèse est que tous les SeD sont contactés à chaque requête. La métrique employée est la moyenne des temps de soumission, *i.e.*, le temps écoulé, du point de vue du client, entre la soumission d'une requête au MA et la réception d'une référence sur un serveur.

4.4.1.2 Ajout d'un serveur à DIET

Cette expérience a pour but de quantifier le coût de l'ajout d'un SeD à une hiérarchie DIET existante. Cette hiérarchie est la plus basique puisque constituée d'un unique MA. L'ajout « naïf » d'un SeD par connexion directe à ce MA n'introduit aucun parallélisme dans la propagation des requêtes et doit donc conduire aux pires performances. Nous avons fait varier le nombre de SeD alloués de un à huit. Le nombre de stations de travail disponibles nous a empêchés de tester de plus grandes configurations. En effet, la co–allocation de plusieurs SeD sur une machine aurait

certes permis de connecter plus de SeD, mais les mesures ainsi obtenues n'auraient plus été significatives au regard de l'expérience souhaitée. Nous pouvons donc supposer que l'ajout d'un SeD à une hiérarchie DIET existante, sans ajouter de LA intermédiaire devrait entraîner une perte de performances de l'ordre de celles mesurées au cours de cette expérience. Le tableau 4.1 présente les résultats obtenus.

Nombre de SeD	1	2	3	4	5	6	7	8
Temps de soumission (en ms.)	4,4	6,0	6,7	7,4	8,0	9,0	9,8	10.6

TAB. 4.1 – Coût de l'ajout « naïf » d'un SeD dans DIET.

Dans cette expérience, le MA contacte les SeD les uns après les autres. Or, lorsqu'un SeD reçoit une requête, il commence à la traiter immédiatement. De plus, le temps de propagation d'une requête est plus rapide que le temps de traitement. Par conséquent, un recouvrement de l'envoi par le traitement des requêtes par les SeD apparaît, ce qui explique que les temps obtenus ne soient pas linéaires.

L'utilisation de LA intermédiaires permettrait d'introduire du parallélisme dans la diffusion des requêtes, et par conséquent d'améliorer les performances. Afin de déterminer le surcoût minimal pour l'ajout de serveurs dans DIET, nous avons considéré comme temps de référence le temps de l'ajout de quatre serveurs à une hiérarchie comprenant déjà quatre serveurs. Un tel ajout augmente de $3,2\ ms$ le temps de soumission moyen, ce qui correspond à un surcoût de 43%. Les expériences suivantes visent à déterminer la meilleure topologie pour diminuer ce surcoût.

4.4.1.3 Comparaison entre deux architecture de même profondeur

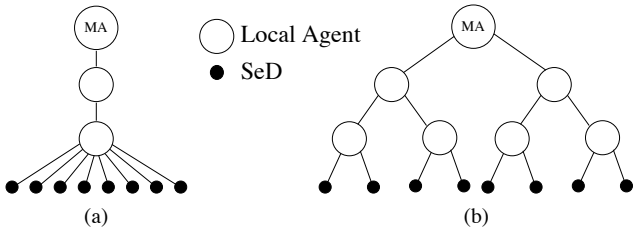


FIG. 4.10 – Comparaison entre deux hiérarchies DIET.

La figure 4.10 présente deux hiérarchies DIET ayant la même profondeur et comprenant le même nombre de SeD. Le coût d'une communication sur une branche de la hiérarchie est exactement le coût d'une communication sur le réseau Fast Ethernet. C'est pourquoi deux LA intermédiaires sont utilisés dans la hiérarchie de la figure 4.10 (a) pour obtenir une profondeur identique à celle de la hiérarchie de la figure 4.10 (b). Même si, dans la pratique, une telle topologie linéaire ne sera jamais déployée, cela nous évite de comparer deux hiérarchie de profondeurs différentes, *i.e.,* avec des coûts de communication différents entre le MA et un SeD.

Le temps de soumission moyen est de $52,2\ ms$ pour la hiérarchie de la figure 4.10 (a) et seulement $33,5\ ms$ pour celle de la figure 4.10 (b). Cela confirme le fait que, pour obtenir de bonnes performances, une hiérarchie DIET doit être correctement déployée. Les expériences du paragraphe suivant visent à établir quelques règles simples pour réaliser au mieux cette phase de déploiement.

4.4.2 Évaluation du coût de l'architecture

En évaluant le coût de l'ajout de serveurs dans la hiérarchie selon diverses stratégies, nous voulons montrer comment la propagation et le traitement des requêtes peuvent être effectués en parallèle dans DIET. Les expériences suivantes montrent qu'il est possible d'ajouter des serveurs au système pratiquement sans surcoût. Les conditions expérimentales sont les mêmes que celles détaillées précédemment.

4.4.2.1 Ajout d'une branche à une hiérarchie DIET

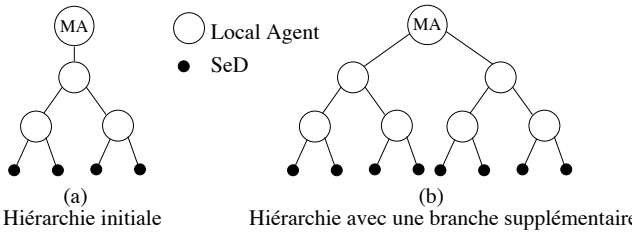


FIG. 4.11 – Ajout d'une branche à une hiérarchie DIET.

La figure 4.11 montre comment il est possible de disposer de deux fois plus de SeD en ajoutant simplement un fils au MA. Ce fils est la racine d'une hiérarchie similaire à celle d'origine, qui est un arbre binaire. Nous avons vu, lors de l'expérience précédente qu'en augmentant le nombre de SeD dans les mêmes proportions, un surcoût de $3\ ms$ apparaissait. Le temps de soumission moyen est de $32,3\ ms$ pour la hiérarchie initiale (figure 4.11 (a)) et de $33,5\ ms$ pour la hiérarchie après l'ajout d'une branche supplémentaire (figure 4.11 (b)).

Nous obtenons donc une augmentation de $3,7\%$ ($1,2\ ms$) du temps de soumission moyen. Cette augmentation est négligeable comparée à celle mesurée lors de l'ajout « naïf » de SeD (43% et $3,2\ ms$). Cette expérience simple montre qu'il est possible d'obtenir un traitement parallèle des requêtes lorsque l'on ajoute les serveurs sur une branche indépendante. La stratégie d'ajout de nouveaux serveurs doit donc être soigneusement étudiée. Dans l'expérience suivante, différentes méthodes d'ajout de serveurs ont été testées.

4.4.2.2 Comparaison architecturale pour l'ajout de serveurs

Dans cette expérience, nous n'avons considéré que des hiérarchies dont le niveau de profondeur est un. Les configurations testées peuvent être composées d'autant de LA que nécessaire, à partir du moment où il n'y a pas plus d'un LA entre le MA et un SeD. Cette restriction nous permet de comparer des hiérarchies ayant le même nombre de SeD, mais des diamètres différents. La figure 4.12 montre les configurations testées au cours de cette expérience. Le nom de chacune de ces configurations dépend du nombre de LA et de SeD qu'elle comprend. Par exemple, la configuration $(1/4)$ implique un LA et quatre SeD. À partir de cette configuration initiale, deux autres hiérarchies sont crées en déclarant quatre serveurs supplémentaires. La première, $(1/8)$, est construite sans ajouter de LA, alors que la seconde « élargit » la configuration initiale. De la même manière, trois hiérarchies impliquant douze serveurs sont déployées. Les résultats obtenus sont présentés par le tableau 4.2.

Le temps de soumission moyen pour la configuration initiale est de $58,8\ ms$. Lorsque quatre serveurs sont ajoutés à cette configuration en s'enregistrant directement au LA existant, ce temps

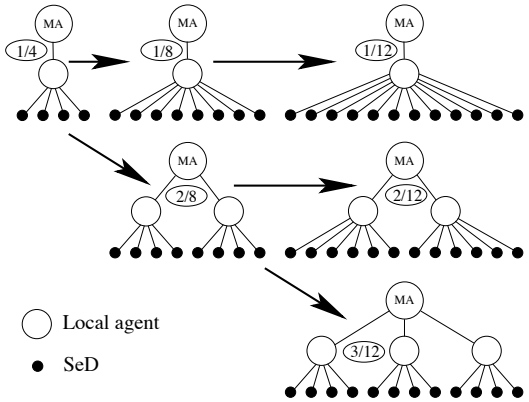


FIG. 4.12 – Configurations testées.

Configuration	1/4	1/8	2/8	1/12	2/12	3/12
Temps de soumission (en ms.)	58,8	69,6	60,9	74,4	62,6	62,9

TAB. 4.2 – Évolution du temps de propagation en fonction du nombre de SeD et du diamètre de la hiérarchie DIET.

de soumission augmente de $18,36\ \%$, soit $10,8\ ms$. En revanche, lorsque l'enregistrement s'effectue auprès d'un LA supplémentaire, le surcoût engendré est seulement de $3.5\ \%$, soit $2,1\ ms$. Ceci démontre l'intérêt de l'utilisation de plusieurs LA afin de paralléliser la propagation des requêtes.

Le temps de soumission moyen de la configuration (1/12) est de $74,4\ ms$. Mais les résultats obtenus pour chacune des deux autres hiérarchies impliquant douze SeD sont relativement similaires. Cela montre que l'ajout d'un LA supplémentaire n'est utile qu'à partir d'un certain nombre de nouveaux SeD. Par exemple, avec douze SeD, la différence de temps de propagation entre la meilleure et la pire des hiérarchies est de $11,8\ ms$, ce qui représente une augmentation de 16%. Cette différence peut cependant augmenter lorsque le nombre de SeD de la hiérarchie croît.

Cette expérience montre l'impact de la hiérarchie sur les performances d'une plate-forme DIET. Évidemment, les hiérarchies déployant des LA en parallèle obtiennent de meilleures performances que les autres. Cependant, le déploiement efficace d'une hiérarchie dépend également fortement de la connaissance du réseau sous-jacent. Ainsi, certaines branches peuvent, pour diverses raisons techniques ou administratives, connecter plus de SeD que d'autres. Toutefois, les résultats obtenus au cours de cette expérience doivent être considéré lors du déploiement d'une hiérarchie DIET.

4.4.3 Conclusions sur les expériences

L'ensemble des expériences menées dans ce chapitre nous permettent d'affirmer que les performances d'une plate-forme DIET dépendent fortement de la structure de la hiérarchie. Deux contraintes majeures doivent être prises en considération afin de déployer une hiérarchie bien adaptée : la hiérarchie DIET doit « suivre » la topologie du réseau physique sous-jacent pour permette une propagation des requêtes plus rapide. Cela peut de plus améliorer la précision des prédictions de performances réalisées par FAST ; l'utilisation d'une hiérarchie de LA a un impact sur les performances de la plate-forme en introduisant du parallélisme dans le traitement des requêtes. Cependant, le choix de la meilleure architecture peut devenir un problème critique lors-

qu'un grand nombre de serveurs est connecté. Toutes ces expériences nous ont tout de même permis de dégager quelques directives pour le déploiement d'une hiérarchie efficace. Ce problème dépend également des vitesses de processeurs, des performances du réseau, etc. Plusieurs architectures doivent donc être testées lors d'un déploiement dans le cas où les performances du système se doivent d'être bonnes.

4.5 Conclusion et perspectives

Dans ce chapitre, nous avons présenté l'architecture de DIET, un ensemble hiérarchique de composants pour l'élaboration d'applications conformes au modèle ASP. Cet environnement est basé sur une hiérarchie d'agents connectant clients et serveurs. Une série d'expériences nous a permis de valider notre approche hiérarchique et de dégager quelques règles de base concernant l'élaboration d'une arborescence performante. Nous avons également présenté les applications cibles de DIET, étudiées dans le cadre des projets GASP et GRID-ASP. Nos perspectives de développement concernent la tolérance aux pannes, la sécurité, le déploiement dynamique de la hiérarchie et enfin l'utilisation du parallélisme mixte au sein de DIET.

4.5.1 Tolérance aux pannes et sécurité

Dans un environnement distribué tel que DIET, les aspects relatifs à la tolérance aux pannes et à la sécurité sont des points cruciaux. Les développements concernant ces sujets se situent à plusieurs niveaux. En effet, la panne d'un serveur sur lequel s'exécute une résolution de problèmes entraîne un redémarrage des calculs en cours. Pour des calculs de longue durée, des mécanismes de sauvegardes intermédiaires et de reprises sur erreurs doivent alors être mis en place pour diminuer la perte de temps induite par cette panne.

Dans le cas de la panne d'un des agents de la hiérarchie, il est nécessaire de reconstruire au plus vite la hiérarchie. En effet, si des données persistantes sont conservées dans le sous–arbre dont cet agent est la racine, ces données deviennent indisponibles. Le premier point à considérer est la découverte de la panne de l'agent. Un mécanisme simple de détection consiste à décider qu'au-delà d'un certain temps de réponse à une requête, l'agent contacté est considéré comme en panne. Une fois la panne détectée, une tentative de reconstruction doit être mise en œuvre. Une des solutions possibles est de conserver dans chaque agent la liste des LA depuis la racine. Chacun des fils de l'agent en panne doit alors prendre l'initiative de contacter son « grand-père ». Enfin, si un MA tombe en panne, il n'est pour l'instant pas envisagé d'autre solution que la reconnexion des clients auprès d'un autre MA et le redémarrage des sessions de ces clients. Cela induit également la perte des calculs en cours et l'obligation d'avertir les serveurs de cette panne.

Concernant la sécurité, il est nécessaire de mener une étude plus approfondie de l'architecture de DIET en cherchant à dégager les attaques possibles, les préventions et le type de sécurité nécessaire. En effet, ce n'est qu'après une analyse des différents flux de données présents dans DIET que nous pourrons proposer une architecture générale de sécurité à mettre en œuvre au niveau de chaque composant, qu'il s'agisse d'un client, d'un agent ou d'un serveur. Du point de vue des communications, l'utilisation de la sécurité du protocole Internet [70] ou de solutions telles que SSL (*Secure Socket Layer*) peut permettre de sécuriser les échanges de données entre les différents composants de la hiérarchie. De plus, une disposition stratégique des agents en fonction des différents pare-feux peut permettre de refléter le découpage du réseau en domaines sécurisés. L'objectif de la sécurisation de notre architecture est enfin de faire en sorte qu'un utilisateur puisse être authentifié sur l'ensemble des serveurs. Or, les différentes solutions actuellement disponibles

sont conçues sur le modèle client–serveur, alors que DIET est fondé sur un modèle impliquant un ensemble de clients, d'agents et de serveurs. Cette authentification unique soulève le problème de la délégation de certificat. Tous ces aspects seront étudiés dans le cadre de la sécurisation de DIET.

4.5.2 Déploiement dynamique de la hiérarchie

Une autre perspective de développement que nous souhaiterions explorer concerne le déploiement dynamique d'une hiérarchie DIET. Cela permettra, d'une part, d'alléger les travaux d'administration et d'étude préliminaire nécessaires à l'obtention d'une hiérarchie efficace et, d'autre part, d'équilibrer dynamiquement la répartition des clients et des serveurs entre les agents en fonction de l'évolution du nombre de ceux-ci. Une première solution envisageable consiste à démarrer des agents sur l'ensemble des machines disponibles pouvant servir de relais entre les serveurs de calcul et les clients puis d'élaguer le graphe ainsi obtenu pour ne conserver que les branches les plus performantes. Enfin, grâce au déploiement dynamique, la réaction à une panne d'agent peut être gérée différemment. En effet, lors de la détection d'une panne, le déploiement d'une nouvelle branche peut être déclenché afin de pouvoir récupérer les serveurs situés sous l'agent défectueux.

4.5.3 Utilisation du parallélisme mixte

Si nos travaux relatifs à la prédiction de performances présentés au chapitre 2 sont déjà utilisés dans DIET, nous envisageons également d'intégrer l'algorithme d'ordonnancement mixte du chapitre 3 au sein de l'ordonnanceur de DIET. En effet, les données issues d'une résolution de problèmes pouvant rester distribuées sur la plate-forme d'exécution à la fin du calcul, et leur réplication n'étant pas autorisée dans DIET, nous nous trouvons exactement dans le cadre d'utilisation de l'algorithme présenté en 3.4. De plus, comme aucune supposition concernant les caractéristiques d'une machine n'a été faite lors du développement de cet algorithme, les informations nécessaires ne faisant référence qu'à des couples {routine, configuration}, il semble tout à fait adapté à une utilisation dans un environnement tel que DIET. Les seules modifications à apporter pour intégrer notre algorithme à l'ordonnanceur de DIET consistent à ajouter un système similaire à celui du séquencement de requêtes de NETSOLVE. Cela permettrait à un client de DIET de spécifier quelle partie du code est à ordonnancer en utilisant le parallélisme mixte et à DIET d'extraire le graphe de tâches correspondant à cette portion de code. Une fois ce graphe extrait, notre algorithme pourra être appliqué afin de déterminer si une exécution mixte est possible. Cette technique pose néanmoins un problème de la gestion de la dynamicité de la plate-forme. En effet, les décisions prises par l'algorithme d'ordonnancement seront basées sur une vue à un instant donné de la plate-forme d'exécution. DIET étant un environnement partagé entre diverses applications, une possibilité de rééquilibrage à l'exécution de la répartition des tâches devra être considérée.

Conclusion

> J'peux pas. J'suis sur un coup, pis j'ai des bonnes chances de conclure, là.
> Jean-Claude Dus, avec un D comme Dus.

Au cours de cette thèse, nous avons abordé plusieurs aspects du *grid computing*, *i.e.*, la connexion de machines parallèles par des réseaux hétérogènes. Nous avons tout d'abord présenté les travaux effectués au cours de l'Action de Recherche Coopérative OURAGAN autour de la parallélisation de SCILAB, un environnement de programmation de haut niveau similaire à Matlab. Deux approches ont été suivies lors de ces développements. La première consiste à répliquer SCILAB sur la nœuds d'une machine parallèle et à fournir des interfaces permettant de faire communiquer ces instances du logiciel et d'utiliser des bibliothèques numériques parallèles performantes telles que SCALAPACK. La seconde approche utilise quant à elle SCILAB comme portail d'accès à un ensemble de serveurs de calculs distribués de par le monde. Une interface avec le logiciel NETSOLVE, qui offre ce type de fonctionnalités à ainsi été développée. Cette approche étant plus extensible, plus dynamique et moins connectée à SCILAB que la précédente, nous avons poursuivi nos recherche dans cette voie.

L'un des problèmes majeurs associé aux environnements à base de serveurs de calcul concerne la capacité à estimer précisément le temps de résolution d'un problème sur un serveur donné, ainsi que les temps de transfert des données nécessaires à cette résolution. Nous avons proposé une solution par l'intermédiaire de la bibliothèque FAST et son extension pour la gestion des routines parallèles. Les modules de FAST en charge de la prédiction des besoins, en termes de puissance de calcul et de mémoire, d'une routine, et de l'acquisition des disponibilités du système et du réseau au moment de l'estimation, ont été développés par Martin Quinson. La modélisation des besoins est fondée sur un travail préliminaire consistant à exécuter une série de tests extensifs pour chaque couple {routine ; machine } considéré. Cette technique s'apparente à celles utilisées dans la génération automatique de bibliothèques à hautes performances et permet de prendre en considération toutes les optimisations apportées aux noyaux séquentiels d'algèbre linéaire, par exemple. L'acquisition des données dynamiques est quant à elle confiée à un système distribuée de

sondes logicielles qui permettent non seulement d'obtenir régulièrement des mesures, mais aussi de prédire les évolutions futures. Nous nous sommes plus particulièrement intéressés dans cette thèse à combiner les informations fournies par ces modules et des modèles issus d'une analyse de code afin d'être en mesure de proposer des estimations précises de routines parallèles. Une routine parallèle est alors décomposée en phases de calcul et de communication. Les phases de calcul sont composées d'appels à des routines séquentielles, estimables par FAST, alors que les phases de communication peuvent se ramener à un modèle classique dont les paramètres peuvent également être acquis en utilisant FAST. Cette technique permet de prendre en compte la forme de la grille de processeurs utilisée, qui a un impact non négligeable sur les performances d'une routine donnée.

Les développements liés à la parallélisation de SCILAB nous ont également permis de mettre en évidence certaines limitations des environnements à base de serveurs de calculs existants. Notamment, la présence d'un agent centralisé, en charge de la répartition des requêtes émanant des clients sur les serveurs, peut conduire à un phénomène de goulot d'étranglement qui limite le passage à l'échelle de ces environnements. Nous avons donc proposé une approche hiérarchique des serveurs de calcul en développant l'environnement DIET, dans le cadre de deux projets nationaux : les projets GASP du RNTL et GRID-ASP de l'ACI GRID. L'utilisation d'une hiérarchie d'agents permet d'une part d'assurer le passage à l'échelle de l'environnement, et de réduire le temps de propagation des requêtes d'autre part. Nous disposons ainsi d'un ensemble hiérarchique de composants pour l'élaboration d'applications conformes au modèle *Application Service Provider*. Plusieurs applications cibles, provenant de domaines aussi divers que la physique, la chimie, la géologie ou encore la micro–électronique, sont en cours de portage sur cet environnement. Nous avons également présenté une série d'expériences qui nous a permis de valider notre approche hiérarchique et de dégager quelques règles de base concernant l'élaboration d'une arborescence performante.

Du point de vue algorithmique, nous avons étudié l'exploitation simultanée des parallélismes de tâches et de données, appelée *parallélisme mixte*. Ce paradigme de programmation permet potentiellement d'exhiber plus de parallélisme qu'en utilisant seulement l'une des deux formes. Le principe de base est de considérer une application sous la forme d'un graphes de tâches dont chacun des nœuds représente un calcul potentiellement parallèle. L'objectif est alors de placer et d'ordonnancer ces tâches sur un ensemble de processeurs de manière à obtenir le meilleur temps de complétion possible. Nous avons tout d'abord développé selon ce paradigme plusieurs implantations des algorithmes rapides de produit de matrices de Strassen et Winograd. Nous avons choisi ces applications car leur graphes de tâches sont exclusivement composés de tâches parallélisables et par conséquent bien adaptés à l'application du parallélisme mixte. Ces implantations ont été évaluées théoriquement et validées expérimentalement en les comparant à des implantations utilisant le parallélisme de données. Nous avons également proposé un algorithme original utilisant le parallélisme mixte pour ordonnancer des applications lorsque les données ne peuvent être répliquées. Le principe fondateur de cet algorithme est d'associer à chaque nœud du graphe de tâches de l'application une liste de configurations, *i.e.*, de grilles de processeurs, pour lesquelles nous savons prédire le temps d'exécution. Cela nous permet alors d'effectuer simultanément le placement et l'ordonnancement de ces tâches. Nous avons appliqué cet algorithme à deux applications : la multiplication de matrices complexes et l'algorithme de Strassen. Les ordonnancements produits pour ces applications aboutissent à des temps de complétion inférieurs à ceux obtenus par une ordonnancement n'utilisant que le parallélisme de données.

Les perspectives à court terme ont été présentées au fur et à mesure des chapitres. Concernant SCILAB$_{//}$, les développements présentés dans le chapitre 1 ont pris fin en même temps que l'ARC

OURAGAN. Cependant, une série de documentations sur chacune des différentes interfaces est en cours de rédaction, et une version du logiciel incluant tous les développements effectués devrait prochainement être diffusée. Nous comptons également étendre notre extension de FAST afin de gérer plus de routines parallèles et notamment les fonctions de factorisation contenues dans SCALAPACK. En effet, la routine de factorisation *LU* de SCALAPACK n'est composée que d'appels à des fonctions similaires à celles dont nous avons proposé des modèles au chapitre 2. Il semble donc aisé d'extraire des modèles pour ce type de routines qui seront basés sur ceux déjà réalisés. De fait, plus FAST sera capable de prédire le temps d'exécution de routines de base, qu'elles soient séquentielles ou parallèles, plus il deviendra aisé d'étendre encore d'avantage cet outil. D'autres perspectives concernent le parallélisme mixte. Nous souhaitons notamment améliorer notre algorithme d'ordonnancement, détaillé dans le chapitre 3. Différents points sont à étudier afin de produire de meilleurs ordonnancements. Nous pouvons ainsi explorer d'une manière plus vaste l'espace des solutions, tout en proposant des heuristiques efficaces, ou bien modifier la gestion de certaines tâches pour lesquelles les contraintes liées aux données sont plus fortes. L'utilisation du parallélisme mixte fait également partie des perspectives de développement de l'environnement DIET que nous avons présenté au chapitre 4. Nous souhaitons également développer dans DIET les aspects relatifs à la tolérance aux pannes et à la sécurité. Ces deux points sont tout d'abord à considérer du point de vue des applications, avec la mise en place de mécanismes de sauvegardes intermédiaires et de reprises sur erreurs et d'une politique de certification de ces applications. Ces sujets devront également être traités au niveau de l'environnement lui-même, par la gestion adaptée des pannes affectant les divers points de l'architecture et la sécurisation, par des méthodes de chiffrement par exemple, de toutes les communications transitant au travers de DIET. Enfin, il devra être fait en sorte qu'un utilisateur puisse être authentifié sur l'ensemble des serveurs sans avoir à fournir un mot de passe à chaque connexion. Pour cela, une étude approfondie des problèmes d'authentification unique et de délégation de certificats sera conduite.

Comme perspectives à plus long terme, plusieurs thèmes de recherche peuvent être explorés à partir des travaux effectués au cours de cette thèse. En premier lieu, nous pouvons utiliser les compétences acquises lors du développement de l'extension parallèle de FAST pour élaborer une méthode d'estimation des redistributions de données entre machines parallèles dans le cadre du grid computing. Une première étape consisterait à étudier précisément les schémas de communications induits par différents type de modifications entre les grilles de processeurs source et destination. Cette étude pourrait reposer sur des algorithmes de redistribution existants, tel que celui de la chenille. Par la suite, l'utilisation d'un simulateur, recréant les conditions d'exécution sur une plate-forme de grid computing, en ajoutant une garantie de reproductibilité des expériences, pourrait éventuellement permettre de développer des algorithmes efficaces pour différentes grandes classes extraites de l'étude préliminaire.

Un second point qu'il serait intéressant d'aborder concerne la problématique des systèmes pair-à-pair. En effet, le modèle de calcul global pair-à-pair est en train de révolutionner le grid computing. Il s'agit d'aggréger dynamiquement des ressources de calcul disponibles à certains moments de la journée afin de résoudre des problèmes aux besoins en puissance de calcul astronomiques. Les machines impliquées dans la plate-forme peuvent donc jouer alternativement le rôle de client ou de serveur. Les applications actuelles sont surtout massivement parallèles et complètement découplées. L'exemple le plus médiatique est certainement le projet SETI@home dont le but est de trouver des traces de signaux extraterrestres grâce à un programme logé dans un économiseur d'écran. Malgré le succès de ce type de projets, le potentiel de calcul encore disponible reste gigantesque. Il est donc critique d'élargir le domaine des applications pouvant bénéficier de ce potentiel. Par exemple, le domaine de la bio-informatique possède des appli-

cations telles que l'alignement de séquences et l'identification de protéines peuvent bénéficier de l'apport d'une plate-forme de calcul global. Par ailleurs, de nombreuses autres applications existent mais ne sont pas si simples à implanter dans un environnement aussi dynamique et hétérogène qu'Internet. En effet, que ce passe-t-il cependant lorsque des dépendances existent entre les diverses tâches de l'application cible ? Aucun projet à notre connaissance ne s'occupe directement de ce problème. De plus, les données peuvent être répliquées sur n'importe lequel des autres pairs, afin de limiter les transferts et ajouter de la redondance pour la tolérance aux pannes. La gestion de ces réplicas et les transferts éventuels entre pairs doivent être gérés correctement si l'on veut conserver des performances attractives. La première tâche a effectuer dans cette optique sera de modéliser la plate-forme de calcul global et son comportement. Ceci est très lié aux travaux autour du comportement d'Internet. Il s'agira ensuite de développer différents algorithmes d'ordonnancement correspondants à divers scénarios d'applications, *e.g.*, parallélisme massif, calcul ou communication dominant, etc. et à différentes implantations de la plate-forme, *e.g.*, serveur centralisé, pur pair-à-pair, etc. Du fait de l'instabilité d'une telle plate-forme basée sur l'Internet, les heuristiques développées pourront être validées par simulation.

Table des figures

Publications personnelles

Revues internationales avec comité de lecture

• Eddy Caron, Serge Chaumette, Sylvain Contassot-Vivier, Frédéric Desprez, Eric Fleury, Claude Gomez, Maurice Goursat, Emmanuel Jeannot, Dominique Lazure, Fréderic Lombard, Jean-Marc Nicod, Laurent Philippe, Martin Quinson, Pierre Ramet, Jean Roman, Franck Rubi, Serge Steer, Frédéric Suter, and Gil Utard. Scilab to Scilab$_{//}$, the OURAGAN Project. *Parallel Computing*, 11(27) :1497–1519, Oct 2001.

• Frédéric Desprez and Frédéric Suter. Impact of Mixed-Parallelism on Parallel Implementations of Strassen and Winograd Matrix Multiplication Algorithms. *To appear in Concurrency and Computation, Practice and Experience*, 2003. Also available as INRIA Research Report RR-4482 at ftp://ftp.inria.fr/INRIA/publication/publi-ps-gz/RR/RR-4482.ps.gz.

• Eddy Caron, Frédéric Desprez et Frédéric Suter. Parallel Extension of a Dynamic Performance Forecasting Tool. *Submitted to Parallel and Distributed Computing Practices*.

Chapitre de livre

• Eddy Caron, Frédéric Desprez, Eric Fleury, Frédéric Lombard, Jean-Marc Nicod, Martin Quinson, and Frédéric Suter. *Calcul réparti à grande échelle*, chapitre « Une approche hiérarchique des serveurs de calculs ». Hermès Science Paris, 2002. ISBN 2-7462-0472-X.

Conférences internationales avec comité de lecture

• Frédéric Desprez and Frédéric Suter. Mixed Parallel Implementations of the Top Level of Strassen and Winograd Matrix Multiplication Algorithms. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS'01)*, San Francisco, April 2001.

• Frédéric Desprez, Martin Quinson, and Frédéric Suter. Dynamic Performance Forecasting for Network-Enabled Servers in a Heterogeneous Environment. In H.R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, volume III, pages 1421–1427, Las Vegas, June 2001. CSREA Press. ISBN : 1-892512-69-6.

• Eddy Caron and Frédéric Suter. Parallel Extension of a Dynamic Performance Forecasting Tool. In *Proceedings of the International Symposium on Parallel and Distributed Computing*, pages 80–93, Iaşi, Romania, July 2002.

• Eddy Caron, Frédéric Desprez, Frédéric Lombard, Jean-Marc Nicod, Martin Quinson, and Frédéric Suter. A Scalable Approach to Network Enabled Servers. In B. Monien and R. Feldmann, editors, *Proceedings of EuroPar 2002*, volume 2400 of *Lecture Notes in Computer Science*, pages 907–910, Paderborn, Germany, August 2002. Springer-Verlag

• Philippe Combes, Frédéric Lombard, Martin Quinson, and Frédéric Suter. A Scalable Approach to Network Enabled Servers. In A. Jean-Marie, editor, *Advances in Computing Science - ASIAN 2002. Internet Computing and Modeling, Grid Computing, Peer-to-Peer Computing, and Cluster Computing. Seventh Asian Computing Science Conference*, volume 2550 of *Lecture Notes in Computer Science*, pages 110–124, Hanoï, Vietnam, December 2002. Springer-Verlag.

Conférences nationales

• Eddy Caron, Dominique Lazure and Frédéric Suter. Manipulation de données de grande taille dans Scilab$_{//}$. *Douzièmes Rencontres Francophones du Parallélisme*, Besançon, Juin 2000.

• Frédéric Lombard, Martin Quinson, and Frédéric Suter. Une approche extensible des serveurs de calcul. *Treizièmes Rencontres Francophones du Parallélisme des Architectures et des Systèmes*, pages 79–84, Paris, La Villette, 24–26 Avril 2001.

• Frédéric Desprez and Frédéric Suter. Produit de matrices, Strassen et parallélisme mixte. *Treizièmes Rencontres Francophones du Parallélisme des Architectures et des Systèmes*, Paris, La Villette, Avril 2001.

• Eddy Caron and Frédéric Suter. Extension parallèle d'un outil de prédiction dynamique de performances. *Quatorzièmes Rencontres Francophones du Parallélisme*, Hammamet, Tunisie, Avril 2002.

Bibliographie

[1] Kento Aida, Atsuko Takefusa, Hidemoto Nakada, Satoshi Matsuoka, Satoshi Sekiguchi, and Umpei Nagashima. Performance Evaluation Model for Scheduling in a Global Computing System. *International Journal of High-Performance Computing Applications*, 14(3) :268–279, 2000.

[2] Jean-Luc Anthoine, Pascal Chatonnay, David Laiymani, Jean-Marc Nicod, and Laurent Philippe. Parallel Numerical Computing Using Corba. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, volume III, pages 1221 – 1228. CSREA Press, July 1998.

[3] Peter Arbenz, Walter Gander, and Michael Oettli. The Remote Computational System. *Parallel Computing*, 23(10) :1421–1428, 1997.

[4] Dorian Arnold, Dieter Bachmann, and Jack 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.

[5] David Bailey, King Lee, and Horst Simon. Using Strassen's Algorithm to Accelerate the Solution of Linear Systems. *Journal of Supercomputing*, 4(4) :357–371, January 1991.

[6] Henri Bal and Matthew Haines. Approaches for Integrating Task and Data Parallelism. *IEEE Concurrency*, 6(3) :74–84, Jul-Sep 1998.

[7] Olivier Beaumont, Vincent Boudet, Arnaud Legrand, Fabrice Rastello, and Yves Robert. Heterogeneous Matrix-Matrix Multiplication, or Partitioning a Square into Rectangles : NP-Completeness and Approximation Algorithms. In *EuroMicro Workshop on Parallel and Distributed Computing (EuroMicro'2001)*, pages 298–305. IEEE Computer Society Press, 2001.

[8] Olivier Beaumont, Vincent Boudet, Fabrice Rastello, and Yves Robert. Matrix Multiplication on Heterogeneous Platforms. *IEEE Transactions on Parallel and Distributed Systems*, 12(10) :1033–1051, 2001.

[9] Saniya Ben Hassen and Henri Bal. Integrating Task and Data Parallelism Using Shared Objects. In *Proceedings of the 10th Conference on Supercomputing*, pages 317–324. ACM Press, 1996.

[10] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and James 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, July 1997. ACM SIGARC.

[11] Petter Bjørstad, Fredrik Manne, Tor Sørevik, and Marian Vajteršic. Efficient Matrix Multiplication on SIMD Computers. *SIAM Journal on Matrix Analysis and Applications*, 13(1) :386–401, January 1992.

[12] L. Susan Blackford, Jaeyoung Choi, Andrew Cleary, Eduardo D'Azevedo, James Demmel, Inderjit Dhillon, Jack Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, Ken Stanley, David Walker, and R. Clinton Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997.

[13] Georges Bosilca, Gilles Fedak, and Franck Cappello. OVM : Out-of-Order Execution Parallel Virtual Machine. In *Proceedings of CCGRID'2001*. IEEE/ACM, IEEE press, May 2001.

[14] Julien Bourgeois. *Prédiction de performances statique et semi-statique dans les systèmes répartis hétérogènes*. PhD thesis, Université de Franche-Comté, January 2000.

[15] Eddy Caron. *Calcul numérique sur données de grande taille*. PhD thesis, Université de Picardie Jules Verne, December 2000.

[16] Eddy Caron, Serge Chaumette, Sylvain Contassot-Vivier, Frédéric Desprez, Eric Fleury, Claude Gomez, Maurice Goursat, Emmanuel Jeannot, Dominique Lazure, Fréderic Lombard, Jean-Marc Nicod, Laurent Philippe, Martin Quinson, Pierre Ramet, Jean Roman, Franck Rubi, Serge Steer, Frédéric Suter, and Gil Utard. Scilab to Scilab$_{//}$, the OURAGAN Project. *Parallel Computing*, 11(27) :1497–1519, October 2001.

[17] Eddy Caron, Frédéric Desprez, Eric Fleury, Frédéric Lombard, Jean-Marc Nicod, Martin Quinson, and Frédéric Suter. *Calcul réparti à grande échelle*, chapter Une approche hiérarchique des serveurs de calculs. Hermès Science Paris, 2002. ISBN 2-7462-0472-X.

[18] Eddy Caron, Frédéric Desprez, Frédéric Lombard, Jean-Marc Nicod, Martin Quinson, and Frédéric Suter. A Scalable Approach to Network Enabled Servers. In B. Monien and R. Feldmann, editors, *Proceedings of the 8th International EuroPar Conference*, volume 2400 of *Lecture Notes in Computer Science*, pages 907–910, Paderborn, Germany, August 2002. Springer-Verlag.

[19] Eddy Caron, Dominique Lazure, and Frédéric Suter. Manipulation de données de grande taille dans scilab$_{//}$. In *Douzièmes Rencontres Francophones du Parallélisme*, pages 107–112, Besançon, June 2000.

[20] Eddy Caron, Dominique Lazure, and Gil Utard. Performance Prediction and Analysis of Parallel Out-of-Core Matrix Factorization. In *Proceedings of the 7th International Conference on High Performance Computing (HiPC'00)*, volume 1593 of *Lecture Notes in Computer Science*, pages 161–172. Springer-Verlag, December 2000.

[21] Eddy Caron and Frédéric Suter. Extension parallèle d'un outil de prédiction dynamique de performances. In *Quatorzièmes Rencontres Francophones du Parallélisme*, Hammamet, Tunisie, April 2002.

[22] Eddy Caron and Frédéric Suter. Parallel Extension of a Dynamic Performance Forecasting Tool. In *Proceedings of the International Symposium on Parallel and Distributed Computing*, Iaşi, Romania, July 2002.

[23] Eddy Caron and Gil Utard. Parallel Out-of-Core Matrix Inversion. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS'02)*, Fort Lauderdale, April 2002.

[24] Henri Casanova and Jack Dongarra. Using Agent-Based Software for Scientific Computing in the Netsolve System. *Parallel Computing*, 24 :1777–1790, 1998.

[25] Soumen Chakrabarti, James Demmel, and Katherine Yelick. Models and Scheduling Algorithms for Mixed Data and Task Parallel Programs. *Journal of Parallel and Distributed Computing*, 47 :168–184, 1997.

[26] Barbara Chapman, Matthew Haines, Piyush Mehrotra, Hans Zima, and John Van Rosendale. Opus : A Coordination Langage for Multidisplinary Applications. *Scientific Programming*, 6(2) :345–362, 1997.

[27] Siddhartha Chatterjee, Alvin Lebeck, Praveen Patnala, and Mithuna Thottethodi. Recursive Array Layouts and Fast Parallel Matrix Multiplication. In *Proceedings of Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*, Saint-Malo, June 1999.

[28] Stéphane Chauveau and François 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.

[29] Jaeyoung Choi, Jack Dongarra, L. Susan Ostrouchov, Antoine Petitet, David Walker, and R. Clinton Whaley. A Proposal for a Set of Parallel Linear Algebra Subprograms. In Jack Dongarra, Kaj Madsen, and Jerzy Wasniewski, editors, *Proceedings of The Second International Workshop on Applied Parallel Computing, Computations in Physics, Chemistry and Engineering Science, PARA'95*, volume 1041 of *Lecture Notes in Computer Science*, pages 107–114. Springer, 1996.

[30] Chung-Chiang Chou, Yuefan Deng, and Wang Yuan. 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.

[31] Philippe Combes, Frédéric Lombard, Martin Quinson, and Frédéric Suter. A Scalable Approach to Network Enabled Servers. In A. Jean-Marie, editor, *Advances in Computing Science - ASIAN 2002. Internet Computing and Modeling, Grid Computing, Peer-to-Peer Computing, and Cluster Computing. Seventh Asian Computing Science Conference*, volume 2550 of *Lecture Notes in Computer Science*, pages 110–124, Hanoï, Vietnam, December 2002. Springer-Verlag.

[32] Javier Cuenca, Jack Dongarra, Dominguo Giménez, José González, and Kenneth Roche. Automatic Optimisation of Parallel Linear Algebra Routines with Variable Load. Technical Report UM-DITEC-2002-3, Computer Engineering Department, University of Murcia, Spain, 2002.

[33] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Eric Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP : A Practical Model of Parallel Computation. *Communications of the ACM*, 39(11) :78–95, November 1996.

[34] Joe Czyzyk, Mike Mesnier, and Jorge Moré. NEOS : The Network-Enabled Optimization System. Technical Report MCS-P615-1096, Mathematical and Computer Science Division, Argonne National Lab., 1996.

[35] Eduardo D'Azevedo and Jack Dongarra. The Design and Implementation of the Parallel Out-of-Core ScaLAPACK LU, QR, and Cholesky Factorization Routines. *Concurrency - Practice and Experience*, 12(15) :1481–1493, 2000.

[36] Luiz De Rose and David Padua. A MATLAB to Fortran 90 Translator and its Effectiveness. In *Proceedings of the 10th ACM International Conference on Supercomputing - ICS'96*, Philadephia, PA, May 1996.

[37] Frédéric Desprez, Jack Dongarra, Antoine Petitet, Cyril Randriamaro, and Yves Robert. Scheduling Block-Cyclic Array Redistribution. In E.H. D'Hollander, G.R. Joubert, F.J. Peters, and U. Trottenberg, editors, *Parallel Computing : Fundamentals, Applications and New Directions*, pages 227–234. North Holland, 1998.

[38] Frédéric Desprez, Éric Fleury, Claude Gomez, Serge Steer, and Stéphane Ubéda. Bringing Metacomputing to Scilab. In *Computer Aided Control System Design (CACSD 99)*, Hawaï, USA, August 1999.

[39] Frédéric Desprez, Éric Fleury, and Laura Grigori. Scilab$_{//}$: User Interactive Application and High Performances. In *Third World Multiconference on Systemics, Cybernetics and Informatics (SCI'99) and Fifth International Conference on Information Systems Analysis and Synthesis (ISAS'99)*, Orlando, August 1999.

[40] Frédéric Desprez, Martin Quinson, and Frédéric Suter. Dynamic Performance Forecasting for Network-Enabled Servers in a Heterogeneous Environment. In H.R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, volume III, pages 1421–1427, Las Vegas, June 2001. CSREA Press. ISBN : 1-892512-69-6.

[41] Frédéric Desprez and Frédéric Suter. Mixed Parallel Implementations of the Top Level of Strassen and Winograd Matrix Multiplication Algorithms. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS'01)*, San Francisco, April 2001.

[42] Frédéric Desprez and Frédéric Suter. Produit de matrices, Strassen et parallélisme mixte. In *Treizièmes Rencontres Francophones du Parallélisme des Architectures et des Systèmes*, pages 25–30, Paris, La Villette, April 2001.

[43] Frédéric Desprez and Frédéric Suter. Impact of Mixed-Parallelism on Parallel Implementations of Strassen and Winograd Matrix Multiplication Algorithms. *Submitted to Concurrency and Computation, Practice and Experience*, 2003. Also available as INRIA Research Report RR-4482 at ftp://ftp.inria.fr/INRIA/publication/publi-ps-gz/RR/RR-4482.ps.gz.

[44] DIET. http://www.ens-lyon.fr/~desprez/DIET/.

[45] Stéphane Domas. *Contribution à l'écriture et à l'extension d'une bibliothèque d'algèbre linéaire parallèle*. PhD thesis, École Normale Supérieure de Lyon, August 1998.

[46] Jack Dongarra, Jeremy Du Croz, Iain Duff, and Sven Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1) :1–17, 1990.

[47] Jack Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard Hanson. An Extended Set of Fortran Basic Linear Algebra Subroutines. *ACM Transactions on Mathematical Software*, 14(1) :1–17, 1988.

[48] Jack Dongarra, Peter Mayes, and Guiseppe Radicati di Brozolo. The IBM RISC System 6000 and Linear Algebra Operations. *Supercomputer*, 8 :15–30, 1991.

[49] Jack Dongarra, Antoine Petitet, and R. Clinton Whaley. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 27(1–2) :3–35, 2001.

[50] Jack Dongarra and Kenneth Roche. Deploying Parallel Numerical Library Routines to Cluster Computing in a Self Adapting Fashion. *Submitted to Parallel Computing*, April 2002.

[51] Jack Dongarra, Robert Van De Geijn, and David Walker. A Look at Dense Linear Algebra Libraries. Technical Report ORNL/TM-12126, Oak Ridge National Laboratory, July 1992.

[52] Jack Dongarra and R. Clinton 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.

[53] Peter Drakenberg, Peter Jacobson, and Bo Kägström. 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, 1993.

[54] Bogdan Dumitrescu, Jean-Louis Roch, and Denis Trystram. Fast Matrix Multiplication Algorithms on MIMD Architecture. *Parallel Algorithms and Applications*, 4(2) :53–70, 1994.

[55] Ian Foster and K. Mani Chandy. FORTRAN M : A Language for Modular Parallel Programming. *Journal of Parallel and Distributed Computing*, 26(1) :24–35, 1995.

[56] Ian Foster and Carl Kesselman, editors. *The Grid : Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1998. ISBN 1-55860-475-8.

[57] Ian Foster, David Kohr, Rakesh Krishnaiyer, and Alok Choudhary. Double Standards : Bringing Task Parallelism to HPF via the Message Passing Interface. In *Proceedings of Supercomputing'96*, 1966.

[58] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM : Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing*. The MIT Press, 1994.

[59] Claude Gomez, editor. *Engineering and Scientific Computing with Scilab*. Birkhäuser, 1999. ISBN : 0-8176-4009-6.

[60] Brian Grayson and Robert Van de Geijn. A High Performance Parallel Strassen Implementation. *Parallel Processing Letters*, 6(1) :3–12, 1996.

[61] Malay Haldar, Anshuman Nayak, Abhay Kanhare, Pramod Joisha, Alok Shenoy, Nagaraj Choudhary, and Prithviraj Banerjee. Match Virtual Machine : An Adaptive Runtime System to Execute MATLAB in Parallel. In *Proceedings of the International Conference on Parallel Processing (ICPP'00*, pages 145–152, Toronto, August 2000.

[62] Nicholas Higham. Exploiting Fast Matrix Multiplication Within the Level 3 BLAS. Technical Report TR89-984, Department of Computer Science - Center of Applied Mathematics - Cornell University, April 1989.

[63] Pascal Hénon, Pierre Ramet, and Jean Roman. PaStiX : A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. *Parallel Computing*, 28(2) :301–321, January 2002.

[64] Joel Hollingsworth, Kun Liu, and Paul Pauca. *Parallel Toolbox for MATLAB– Manual and Reference Pages*. Wake Forest University, Mathematics and Computer Science Department, PT v.1.00 edition, September 1996.

[65] Timothy Howes, Mark Smith, and Gordon Good. *Understanding and Deploying LDAP Directory Services*. Macmillian Technical Publishing, 1999. ISBN : 1-57870-070-1.

[66] Parry Husbands and Charles Isbell. The Parallel Problems Server : A Client-Server Model for Interactive Large Scale Scientific Computation. In *Procceding of the 3rd International Meeting on Vector and Parallel Processing (VECPAR'98)*, pages 156–169, Porto, Portugal, June 1998.

[67] Stephen Huss-Lederman, Elaine Jacobson, Jeremy Johnson, Anna Tsao, and Thomas Turnbull. Strassen's Algorithm for Matrix Multiplication : Modeling, Analysis, and Implementation. Technical Report CCS-TR-96-147, Center for Computing Sciences, Argonne National Laboratory, 1996.

[68] iMW. http://www-unix.mcs.anl.gov/metaneos/softtools/imw.html.

[69] Silicon Graphics Inc. Performance Co-Pilot : Monitoring and Managing System-Level Performance. http://www.sgi.com/software/co-pilot/, 2001.

[70] IPSec. http://www.ietf.org/rfc/rfc2401.txt.

[71] Nirav Kapadia, Renato Figueiredo, and José Fortes. PUNCH : Web Portal for Running Tools. *IEEE Micro*, 20(3) :38–47, 2000.

[72] Charles Koelbel, David Loveman, Robert Schreiber, Guy Steele Jr., and Mary Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994. ISBN 0-262-11185-3.

[73] Charles Lawson, Richard Hanson, David Kincaid, and Fred Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transactions on Mathematical Software*, 5 :308–323, 1979.

[74] Legion. http://legion.virginia.edu/.

[75] Frédéric Lombard, Martin Quinson, and Frédéric Suter. Une approche extensible des serveurs de calcul. In *Treizièmes Rencontres Francophones du Parallélisme des Architectures et des Systèmes*, pages 79–84, Paris, La Villette, April 2001.

[76] Alexey Malishevsky, Nagajagadeswar Seelam, and Michael Quinn. Otter : Bridging the Gap between MATLAB and ScaLAPACK. In *Proceedings of the 7th IEEE International Symposium on High Performance in Distributed Computing*, pages 114–123, August 1998.

[77] Matlab. http://www.mathworks.com.

[78] Satoshi Matsuoka and Henri 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).

[79] Satoshi Matsuoka, Hidemoto Nakada, Mitsuhisa Sato, , and Satoshi 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.

[80] Igor Milosavljevic and Marwan Jabri. Automatic Array Alignment in Parallel Matlab Scripts. In *Proceedings of the 13th International Parallel and Distributed Processing Symposium (IPDPS'99)*, Puerto Rico, April 1999.

[81] Cleve Moler. Why there isn't a parallel MATLAB. *MATLAB News and Notes*, April 1995.

[82] Greg Morrow and Robert 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/.

[83] MPICH-G. http://www.hpclab.niu.edu/mpi/.

[84] Hidemoto Nakada, Satoshi Matsuoka, Keith Seymour, Jack Dongarra, Craig Lee, and Henri Casanova. GridRPC : A Remote Procedure Call API for Grid Computing. In *Proceedings of the Workshop on Grid Computing (GRID 2002)*, volume ? of *LNCS*, page ?, Baltimore, November 2002. Springer-Verlag.

[85] Hidemoto Nakada, Hiromitsu Takagi, Satoshi Matsuoka, Umpei Nagashima, Mitsuhisa Sato, and Satoshi 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.

[86] NEOS. http://www-neos.mcs.anl.gov/.

[87] Netsolve. http://www.cs.utk.edu/netsolve/.

[88] NIMROD. http://www.csse.monash.edu.au/~davida/nimrod.html/.

[89] NINF. http://ninf.etl.go.jp/.

[90] OpenMP. http://www.openmp.org/.

[91] Salvatore Orlando, Paolo Palmerini, and Raffaele Perego. Mixed Task and Data Parallelism with HPF and PVM. *Cluster Computing*, 3(3) :201–213, 2000.

[92] PACXMPI. http://www.hlrs.de/structure/organisation/par/projects/pacx-mpi/.

[93] Victor Pan. How to Multiply Matrices Faster. In *Lecture Notes in Computer Science*, volume 179. Springer-Verlag, 1984.

[94] Paramat. http://www.alpha-data.co.uk/dsheet/paramat.html.

[95] Sven Pawletta, Andreas Westphal, Thorsten Pawletta, Wolfgang Drewelow, and Peter 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. http://www-at.e-technik.uni-rostock.de/rg_ac/dp/.

[96] PMI. ftp://ftp.mathworks.com/pub/contrib/v5/tools/PMI/.

[97] Loïc Prylli and Bernard Tourancheau. Fast Runtime Block Cyclic Data Redistribution on Multiprocessors. *Journal of Parallel and Distributed Computing*, 45(1) :63–72, August 1997.

[98] Martin Quinson. Dynamic Performance Forecasting for Network-Enabled Servers in a Metacomputing Environment. In *Proceedings of the International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS'02)*, Fort Lauderdale, April 2002.

[99] Martin Quinson. Un outil de prédiction dynamique de performances dans un environnement de metacomputing. *Techniques et Sciences Informatiques*, 21(5) :685–710, 2002. Numéro spécial RenPar'13.

[100] Andrei Radulescu, Cristina Nicolescu, Arjan van Gemund, and Pieter Jonker. Mixed Task and Data Parallel Scheduling for Distributed Systems. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS)*, San Francisco, April 2001. Best Paper Award.

[101] Andrei Radulescu and Arjan van Gemund. A Low-Cost Approach towards Mixed Task and Data Parallel Scheduling. In *Proceedings of the 15th International Conference on Parallel Processing (ICPP)*, Valencia, Spain, September 2001.

[102] Shankar Ramaswamy, Eugene Hodges IV, and Prithviraj Banerjee. Compiling MATLAB Programs to ScaLAPACK : Exploiting Task and Data Parallelism. In *International Parallel Processing Symposium (IPPS'96)*, pages 613–619, April 1996.

[103] Shankar Ramaswany. *Simultaneous Exploitation of Task and Data Parallelism in Regular Scientific Applications*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[104] Shankar Ramaswany, Sachin Sapatnekar, and Prithviraj Banerjee. A Framework for Exploiting Task and Data Parallslism on Distributed Memory Multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 8(11) :1098–1116, November 1997.

[105] Thomas Rauber and Gudula Rünger. Compiler Support for Task Scheduling in Hierarchical Execution Models. *Journal of Systems Architecture*, 45 :483–503, 1998.

[106] Victor Rayward-Smith. UET Scheduling with Unit Interprocessor Communication Delays. *Discrete Applied Mathematics*, 18 :55–71, 1987.

[107] RTExpress. http://www.rtexpress.com/.

[108] SETI@home. http://setiathome.ssl.berkeley.edu/.

[109] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI : The Complete Reference*. The MIT Press, 1996.

[110] Paul Springer. Matpar : Parallel Extensions to Matlab : version 1.2. Available at http://www-hpc.jpl.nasa.gov/PS/MATPAR/manual.1_2.pdf, December 1999.

[111] Volker Strassen. Gaussian Elimination Is Not Optimal. *Numerische Mathematik*, 14(3) :354–356, 1969.

[112] Jaspal Subhlok and Bwolen Yang. A New Model for Integrated Nested Task and Data Parallel Programming. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Pratice of Parallel Programming*, pages 1–12, Las Vegas, June 1997. ACM Press.

[113] Lixin Tao. Shifting Paradigms with the Application Service Provider Model. *IEEE Computer*, 34(10) :32–39, October 2001.

[114] Mithuna Thottethodi, Siddhartha Chatterjee, and Alvin Lebeck. Tuning Strassen's Matrix Multiplication for Memory Efficiency. In *Proceedings of Supercomputing'98*, Orlando, November 1998.

[115] Anne Trefethen, Vijay Menon, Chi-Chao Chang, Grzegorz Czajkowski, Chris Myers, and Lloyd Trefethen. MultiMatlab : Matlab on Multiple Processors. Technical Report 96-239, Cornell Theory Center, 1996. http://www.cs.cornel.edu/Info/People/Int/multimatlab.html.

[116] Robert van de Geijn and Jerell Watts. SUMMA : Scalable Universal Matrix Multiplication Algorithm. Technical Report CS-95-286, The University of Tennessee - Knoxville, April 1995. LAPACK Working Note #96.

[117] WebFlow. http://www.npac.syr.edu/users/haupt/WebFlow/.

[118] Emily West and Andrew Grimshaw. Braid : Integrating Task and Data Parallelism. In *Proceedings of the Fifth Symposium on Frontiers of Massively Parallel Computation*, pages 211–219. IEEE CS Press, 1995.

[119] Shmuel Winograd. Some Remarks on Fast Multiplication of Polynomials. In J. F. Traub, editor, *Complexity of Sequential and Parallel Numerical Algorithms*, pages 181–196. Academic Press, 1973.

[120] Rich Wolski, Neil Spring, and Jim Hayes. The Network Weather Service : A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computing Systems, Metacomputing Issue*, 15(5–6) :757–768, October 1999.

[121] Rich Wolski, Neil Spring, and Jim Hayes. Predicting the CPU Availability of Time-Shared Unix Systems. *Cluster Computing*, 3(4) :293–301, 2000.

[122] XtremWeb. http://www.lri.fr/~fedak/XtremWeb/introduction.php3.

Parallélisme mixte et prédiction de performances sur réseaux hétérogènes de machines parallèles

Avec la généralisation de l'Internet, il est désormais possible pour les utilisateurs de calcul numérique d'accéder aux machines les plus puissantes disponibles de par le monde, et ce depuis leur station de travail. A grande échelle, ce type d'accès distant est appelé *metacomputing*.

Les travaux effectués au cours de cette thèse ont tout d'abord concerné la parallélisation du logiciel SCILAB, en suivant, entre autres, une approche basée sur des serveurs de calcul. Au cours de ces développements, les lacunes des environnements de ce type ont été exhibées, notamment le problème de goulot d'étranglement posé par la présence d'un agent centralisé. Afin de pallier ce problème, et donc de proposer un environnement extensible, nous avons suivi une approche hiérarchique pour développer le logiciel DIET (*Distributed Interactive Engineering Toolbox*). Un des points cruciaux des environnements de ce type concerne la capacité à estimer le temps d'exécution d'une routine sur machine donnée et les coûts de transfert des données depuis un client ou un serveur vers le serveur choisi pour la résolution. La bibliothèque FAST (*Fast Agent's System Timer*), que nous avons étendue afin de gérer les routines parallèles, permet d'acquérir ce type d'informations.

D'un point de vue algorithmique, nous avons mené une étude à la fois théorique et expérimentale du parallélisme mixte, *i.e.*, l'exploitation simultanée des parallélismes de tâches et données. Après avoir appliqué ce paradigme aux algorithmes rapides de produit de matrices de Strassen et Winograd, nous avons proposé un algorithme d'ordonnancement en parallélisme mixte dans le cas où les données ne peuvent pas être dupliquées. Cet algorithme effectue simultanément le placement et l'ordonnancement des tâches d'un graphe en se basant sur les modèles de coûts fournis par notre extension de FAST et sur un ensemble de distributions possibles.

Mots clés : Parallélisme mixte, metacomputing, prédiction de performance, ordonnancement, algèbre linéaire, Strassen, Winograd, PSE, ASP.

Mixed parallelism and performance prediction on heterogeneous networks of parallel computers

Thanks to the Internet, numerical computation users can now access the most powerful computer from their workstation. At a large scale this remote access is named *metacomputing*.

My PhD work was first focused on the parallelization of the SCILAB tool, following a computational server based approach. Drawbacks of existing environments raised. One of these was the bottleneck problem due to the centralized agent. To solve that problem and propose a scalable environment, we followed a hierarchical approach to develop DIET (*Distributed Interactive Engineering Toolbox*). One of the critical issue of such environments is the capacity to estimate the execution time of a routine on a given computer and the communication cost of data transfers from a client, or a server, to the server chosen for the resolution. We also extended FAST (*Fast Agent's System Timer*) library to handle parallel routines.

From an algorithmic point of view we ran a theoretical and experimental study of the simultaneous exploitation of data and task parallelism, so called mixed parallelism. After applying this paradigm on fast matrix multiplication algorithms (Strassen and Winograd), we proposed a mixed data and task parallel scheduling algorithm in which data can not be replicated. This algorithm allocates and schedules tasks in one step and based on cost models given by our FAST extension and a set of possible distributions.

Keywords : Mixed data and task parallelism, metacomputing, performance forecasting, scheduling, linear algebra, Strassen, Winograd, PSE, ASP.