Modélisation de l'Hétérogènéité pour le Calcul Parallèle à Gros Grain

Ouissem Ben Fredj

Mémoire de D.E.A. INFORMATIQUE

Université Henri Poincaré Nancy I
U.F.R. S.T.M.I.A.
Département d'informatique

Modèlisation de l'Hétérogènéité pour le Calcul Parallèle à Gros Grain

Présenté le 1 juillet 2003 par :

Ouissem BEN FREDJ

COMPOSITION DU JURY :

Dominique	**MERY**
Didier	**GALMICHE**
Noëlle	**CARBONELL**
André	**SCHAFF**
Jens	**GUSTEDT**

Á ma famille.

Remerciements

Je tiens à remercier M. Jens GUSTEDT qui m'a accueilli au sein de son équipe ALGORILLE.

Je remercie mes encadrants Jens GUSTEDT et Emmanuel JEANNOT qui m'ont fourni la bonne ambiance de travail et m'ont facilité l'intégration au sein de l'équipe.

Je remercie Denis CAROMEL et tous les membres du projet OASIS qui m'ont accueilli dans le laboratoire SOPHIA-ANTIOPOLIS, ainsi de leurs aide continue.

Une salutation particulière pour Mohamed ESSAÏDI de ses conseils précieux dans le cadre de mon travail.

Je remercie également les membres de jury qui ont accepté de juger mon travail.

Table des matières

Chapitre 1

Introduction

1.1 Contexte général

Pour faciliter la conception et le développement des applications parallèles, plusieurs modèles ont été élaborés. Conçus récemment les modèles "*à gros grain*" ont pour objectif de fournir un réalisme aussi bien du point de vue algorithmique que du point de vue architectural. En effet, ces modèles supposent que les entrées du problème à résoudre sont de grandes tailles et une architecture parallèle à mémoire distribuée.

La faisabilité des modèles *à gros grain* a été prouvé par la conception, l'implantation et les tests d'algorithmes. Ces derniers sont développés avec des bibliothèques de support comme SSCRAP[1][1, 2] qui est une implantation de ces modèles. Pour l'instant, les modèles *à gros grain* ne sont pas en mesure de décrire ni les machines parallèles hétérogènes ni le réseau qui assure l'interconnexion des processeurs.

Par ailleurs, il existe aujourd'hui des supports de calcul qui propose un environnement de programmation flexible pour faciliter l'écriture des algorithmes parallèles. Ces supports offrent généralement un modèle de programmation qui facilite la conception et des primitives simples et flexibles qui facilitent l'implantation. Ces supports peuvent être utilisés sur des machines, réseaux et grilles hétérogènes. Des modélisations fines pour la communication et la synchronisation hétérogène ont été élaborées pour certains de ses supports exécutifs.

1.2 Objectifs

Notre travail vise à joindre les deux approches pour proposer ainsi un modèle de distribution, de communication et de synchronisation pour les algorithmes "*à gros grain*" exploitant les architectures hétérogènes. On supposera une architecture où les différents processeurs sont liés avec des réseaux/bus différents, comme dans celles obtenues en reliant des machines multiprocesseurs avec des réseaux à haut débit.

Nous partirons d'un coté des modèles à gros grain et de l'autre part de supports de communication récentes qui traitent déjà l'hétérogénéité. Ce sont en particulier les

[1]SSCRAP : Soft Synchronized Computing in Rounds for Adequate Parallelization

bibliothèques PM2[2][3], ProActive[4] et des implantations basées sur MPI. Ces bibliothèques servirons comme support pour la conception, l'implantation et l'expérimentation du modèle à proposer avec la bibliothèque SSCRAP.

1.3 Plan du manuscrit

La suite de ce mémoire sera organisée de la manière suivante :
Dans le chapitre 2, nous introduisons la notion de modèle parallèle et nous en présentons quelques modèles. Nous détaillons essentiellement, les modèles à gros grain. Le chapitre 3 sera consacré pour les supports de calculs parallèles distribués. Nous présentons les mécanismes élémentaires utilisés par les supports, puis une vue générale des supports MPI, PM^2 et ProActive. Dans le chapitre 4, nous présentons SSCRAP qui est une implantation des modèles parallèles à gros grain.

Notre contribution commence par le chapitre 5 dans lequel on analyse les méthodes adoptés pour exploiter les architectures homogènes. Dans le chapitre 6, nous proposons un modèle pour les architectures hétérogènes et les modèles de déploiement, de communication et de synchronisation considérant l'hétérogénéité. Puis nous développons les interfaces que nous avons élaborées et basées sur nos modèles et ceux de parallélisme à gros grain. L'interface basée sur PM^2 sera détaillée dans le chapitre 7 et celle basée sur ProActive dans le chapitre 8.

[2]PM2 : Parallel Multithreaded Machine

Chapitre 2

Les modèles parallèles

Généralement, lors de la conception d'une application parallèle, les concepteurs, partant de l'analyse des besoins et d'une modélisation grossière, raffinent progressivement leur approche pour aboutir à une modélisation répondant à tous les besoins initiaux et directement implantable. Le raffinement suppose la prise en compte de plusieurs niveaux d'abstraction. Ceci est fait dans le but de faciliter la conception et de pouvoir corriger, réutiliser et étendre la modélisation. Dans le domaine du parallélisme, le modèle parallèle est la principale abstraction définissant une architecture logique d'exécution et/ou des règles de fonctionnements.

Dans ce chapitre, nous commençons par définir les modèles parallèles (section 2.1). Une distinction sera faite entre les modèles à gain fin (section 2.2) et les modèles à gros grain (section 2.3). L'accent sera mis sur ces derniers parmi lesquels on detaillera les plus intéressants.

2.1 Définition

Dans la littérature, la notion de modèle parallèle ne bénéficie pas d'une définition formelle et unique. Un modèle parallèle pourrait :

- définir un cadre architectural logique ;
- définir un processus d'exécution ;
- permettre d'analyser les programmes obtenus en vu d'une étude de performances ou d'une analyse comparative.

Jusqu'à maintenant, aucun modèle parallèle ne s'est imposé comme modèle de référence comme c'est le cas du modèle Von Neumann pour la programmation séquentielle. Dans les modèles parallèles, on distingue deux grandes classes : les modèles à grain fin et les modèles à gros grain.

2.2 Les modèles à grain fin

Cette classe de modèle suppose que la taille des entrées du problème n est linéairement proportionnelle au nombre de processeur p, c'est à dire si $n = O(p)$.

Le modèle PRAM[1], est le premier modèle parallèle à grain fin. Il suppose une architecture logique comportant un ensemble de processeurs chacun muni d'une mémoire locale de taille relativement petite (de l'ordre de $O(1)$) et pouvant communiquer à travers une mémoire partagée commune. Ce modèle suppose que tous les processeurs peuvent accéder en lecture ou en écriture en temps constant à n'importe quelle zone de la mémoire partagée. Il suppose aussi une forte synchronisation entre les processeurs, c'est-à-dire que tous les processeurs sont cadencés par la même horloge.

2.3 Les modèles à gros grain

Les modèles à gros grain prennent en compte des spécificités architecturales réelles en essayant de couvrir en même temps le plus grand nombre de machines possibles. Ces modèles, supposent que le processeur dispose d'une mémoire locale de taille relativement importante et prennent en compte l'aspect communication à travers un réseau d'interconnexion. L'utilisation d'un espace mémoire important par les algorithme a pour but d'augmenter le rapport traitements locaux/communications.

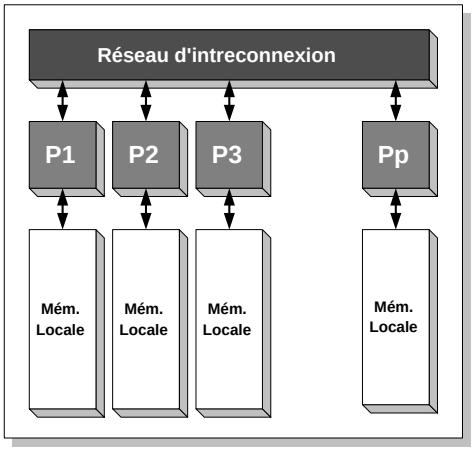


FIG. 2.1 – Architecture de référence pour les modèles à gros grain

Dans ce qui suit, nous présentons trois modèles à gros grain. Ces trois modèles considèrent le même modèle de machine abstraite : un ensemble de processeurs disposant chacun de sa propre mémoire locale communicant à travers un réseau d'interconnexion. Ils considèrent le même modèle d'exécution qui consiste à une succession de super-étapes. Chacune de ses dernières est composée d'une étape de calcul, une étape de communication et éventuellement une étape de synchronisation.

2.3.1 Le modèle BSP

Considérant le modèle de machine explicité précédemment, le modèle d'exécution de BSP[2][5] est subdivisé en une séquence de super-étapes. Une super-étape se décompose

[1]PRAM : Parallel Random Acces Memory/Machine
[2]BSP : Bulk Synchronous Parallel model

en trois sous étapes :

- Le processeur effectue des traitements sur les données locales ;
- Les processeurs effectuent des échanges pour récupérer les données non locales qui leurs sont nécessaires pour la prochaine phase de calcul ;
- Les processeurs entament une phase de synchronisation au bout de laquelle, ils se lancent dans la super-étape suivante.

Le modèle de coût du BSP tient compte des paramètres suivants : n la taille du problème, p le nombre de processeurs, L le délai minimum entre deux phases de synchronisation, s (startup) le sur-coût fixe pour le démarrage d'une communication, h la quantité maximale de données que peut émettre ou recevoir un processeur et g le rapport de la capacité de traitement totale par unité de temps sur la capacité de transmission totale par unité de temps.

2.3.2 Le modèle CGM

Contrairement au modèle BSP, le CGM[3][6, 7] ne considère que les deux paramètres n et P et s'affranchit de la phase de synchronisation. A chaque processeur est associée une mémoire locale de taille $O(\frac{n}{P})$ avec $\frac{n}{P} \gg 1$ ce qui permet de qualifier ce modèle de gros grain.

Au niveau du modèle d'exécution, CGM décrit les algorithmes comme étant une succession de super-étapes. Une super-étape est composée d'une phase de calcul et d'une phases de communication. Pendant les phases de communication chaque processeur doit envoyer ou recevoir $O(\frac{n}{P})$ données. Il peut donc échanger l'intégralité de sa mémoire locale avec un autre processeur.

Puisque la taille de données à communiquer lors d'une super-étape est constant et pour caractériser le temps de communication, le modèle CGM ne considère que le nombre de phases d'échanges. Il vise donc, pour de meilleures performances, à minimiser le nombre d'étapes de communication.

2.3.3 Le modèle PRO

Le modèle PRO[4][8] prend un algorithme séquentiel A de référence dont la taille des entrées est n. A est caractérisé par Time(n) et Space(n) qui représentent respectivement le temps d'exécution et la taille de l'espace mémoire utilisé. Les attributs du modèle sont :

- Une machine Parallèle : composée de n processeurs avec une mémoire privée de taille M=$O(\frac{Space(n)}{P})$ pour chacun.
- Une grosse granularité : on suppose que p $<$ M. C'est-à-dire que la taille de la mémoire locale d'un processeur peut contenir plus que p mots mémoire.
- Un modèle d'exécution : étant donné la fonction de granularité grain(n) qui mesure la qualité de l'algorithme, et pour une valeur p=$O(grain(n))$, un algorithme PRO se déroule au plus en $O(\frac{Time(n)}{p^2})$ super-étapes. une super-étape se décompose comme suit :

[3]CGM : Coarse grained multicomputer

[4]PRO : Parallel Resource-Optimal computation

1. envoie d'au plus un message à chacun des autres processeurs.
2. La taille de données lors d'une communication est M mots mémoire.
3. fait un calcul local.

Un algorithme PRO est considéré optimal si sa granularité grain(n) égale à $O(min(\sqrt{Space(n)}, \sqrt{Time(n)})) = O(\sqrt{Space(n)})$. Dans ce cas, il suit un rendement linéaire par rapport à sa version séquentielle.

2.4 Synthèse

Une comparaison entre les modèles existants peut être vue sous plusieurs angles. Si on les classe selon la simplicité qu'ils offrent vis à vis du programmeur, on trouve que BSP est le plus complexe puisqu'il prend en considération quatre paramètres (L,g,p,n), puis PRO et enfin CGM. Ce classement donne une idée sur le niveau d'abstraction de l'architecture qu'ils supposent. Cependant, les critères les plus importants sont le type de la mémoire, le type de communication et de synchronisation, la granularité, le rendement de l'algorithme, l'optimalité et la qualité de l'algorithme. Le tableau 2.2 ci-joint présente les résultats des comparaisons. On note que le mode d'exécution est de type SPMD[5] pour tous les modèles. C'est-à-dire que tous les nœuds exécutent le même programme mais avec des données différentes.

Pour le critère de synchronisation, on distingue deux types de synchronisation ; une synchronisation totale (lock-step) à chaque super-étape et une Bulk-Sync qui signifie qu'il peut exister des operations asynchrones entre les barrières de synchronisation.

	PRAM	BSP	CGM	PRO
synchronisation	lock-step	bulk-synch.	asynchrone	asynchrone
mémoire	partagée	distribuée	privée	privée
communication	SM	MP	MP/SM	MP/SM
paramètres	n	p, g, L, n	p, n	p, n, A_seq
granularité	grain fin	gros grain	gros grain	Grain (n)
optimalité	NA	NA	NA	rel. A_seq
quantité	temps	temps	super-étape	Grain(n)

FIG. 2.2 – Comparaison entre les modèles de programmation parallèles

Du coté mémoire, elle peut être partagée(SM) ou distribuée où une abstraction sera faite à ce niveau et on suppose que chaque processeur voit que sa mémoire privée (locale). La communication inter-processeurs peut être faite à travers une mémoire partagée(SM) ou par passage de message(MP). Ce dernier type indique que chaque processeur communique avec les autres par envoi explicite de message point à point et une abstraction sera faite sur le détail de l'acheminement des messages à travers le réseau des processeurs.

[5]SPMD : Single Program Multiple Data

Chapitre 3

Les supports de calcul distribué

Dans ce chapitre, nous nous intéressons aux supports de calcul distribué. Nous détaillons dans la section 3.1 les facteurs qui nous amènent à utiliser ces genres de support. Ensuite, nous présentons les mécanismes élémentaires sur lesquels reposent ces supports, à savoir le "multithreading" (section 3.2) et le RPC (section 3.3). On finira par présenter les supports utilisés dans notre travail : MPI (section 3.4), PM^2 (section 3.5) et ProActive (section 3.6).

3.1 Pourquoi utiliser les supports de calcul

Comme nous l'avons introduit dans le chapitre 2, les modèles à gros grain se distinguent des autres modèles par leurs "réalisme". En effet, ils prennent en compte les spécifités architecturales existantes en supposant un nombre fini de processeurs disposant chacun d'une mémoire locale de taille relativement importante et reliés par un réseau d'interconnexion offrant une communication point à point.

Ces hypothèses bien que réalistes, présentent deux limites. La première est que les modèles à gros grain ne prennent pas en charge la couche communication et ne prend en considération aucun type de réseau ni des bibliothèques bas niveaux associés. La deuxième est qu'elles supposent un haut niveau d'abstraction sur l'architecture utilisée. Cette caractéristique bien qu'elle peut être vue comme un avantage qui garantit une large échelle de portabilité, accentue le premier problème. En effet, puisque ces modèles ne se chargent pas de la communication entre les processeurs, on trouvera plus de difficultés à couvrir ce large éventail d'architectures supposées.

D'un autre coté, des supports de calcul distribué ont été élaborés. Ils proposent généralement un modèle de répartition de charges, un modèle de communication et un modèle de programmation.

Parmi les supports existants, on trouve ceux qui traitent le cas général des architectures hétérogènes. Deux exemples représentatifs de ces environnements sont PM^2 et ProActive. PM^2 se base sur le modèle de programmation et de communication client/serveur utilisant sur le mécanisme de RPC. ProActive[4] est quant à lui, basé sur le mécanisme de JAVA-RMI[1].

[1]RMI : Remote Methode Invocation

L'idée est d'utiliser ces environnements pour exploiter les architectures hétérogènes. L'avantage de cette approche est que nous serons pas obligé de réaliser un support propriétaire avant une validation de l'utilisation de ces genres de supports. L'inconvénient de cette approche est que nous serons très liée au modèle de programmation du support utilisé. En effet, il nous faut adapter le modèle de programmation du support à nos besoins, ce qui peut nous faire perdre en performance d'exécution.

3.2 Multithreading

Un thread est un flot d'exécution qui s'exécute à l'intérieur d'un processus UNIX. Le thread exécute une fonction dont le code se trouve dans le segment de code de son processus père. Le multithreading et la technique permettant la gestion de flots d'exécution multiples au sein des entités d'exécution de base fournies par le système d'exploitation. Peu de ressources mémoire sont allouées au thread, par comparaison aux ressources allouées à un processus UNIX. On distingue deux différents types de threads : threads utilisateurs (user) et les threads noyaux (kernel) :

- threads utilisateurs : la bibliothèque utilisant ces threads est composée de fonctions ne s'exécutant que dans l'espace utilisateur. Le noyau du système ne "voit" pas ces threads.
- thread noyau : dans ce cas, les threads sont gérés directement par le noyau. Il existe une norme POSIX qui assure au programmeur un portage aisé de son application multithreadée.

Les bibliothèques des threads utilisateurs bénéficient généralement de très bonnes performances. En effet, un changement de contexte entre threads d'un même processus se résume le plus souvent à sauvegarder les registres du thread que l'on retire et à restaurer ceux du thread que l'on va lancer. En revanche, pour une bibliothèque de threads noyau, c'est l'ordonnanceur du système qui s'en charge. L'intérêt des bibliothèques systèmes se situe au niveau des machines multiprocesseurs. En effet, comme le noyau connaît l'existence des threads, il est capables de les répartir équitablement sur tous les processeurs de la machine. On obtient ainsi une application réellement parallèle.

3.3 RPC et JAVA-RMI

Plutôt que d'adresser une requête (message) à un processus serveur pour obtenir en retour un résultat (par message également), le modèle du RPC[2] propose d'appeler à distance une procédure située sur le site serveur de la même manière que se ferait l'appel d'une procédure locale. En fait, l'appel d'une procédure à distance se déroule comme suit :

1. Le client envoi un message contenant son identifiant, la procédure à executer et ses paramètres vers le serveur.
2. Coté serveur, la requête entrante est désempaquetée et la procédure est exécutée ;

[2]RPC :Remote Procedure Call

3. Le résultat de la fonction est ensuite empaqueté dans un message qui est retourné à l'appelant ;
4. Une fois parvenus sur le site appelant, les résultats sont désempaquetés et celui-ci est débloqué.

Coté JAVA, il possède un RPC orienté objet intégré appelé RMI. Il permet à des objets situés dans des espaces d'adressage différents sur des machines différentes d'interagir d'une manière transparente. Un objet distribué est un objet java comme les autres. D'où la simplicité de le manipuler comme tout autre objet java.

L'avantage du modèle est double : la délégation d'un traitement à un serveur peut être réalisée de façon transparente et le code permettant le transfert des données entre les machines n'est écrit qu'une seule fois (alors qu'il est réécrit à chaque appel dans un modèle Client/Serveur classique).

Plusieurs environnements de programmation parallèle utilisent le mécanisme de RPC. Nous détaillerons les environnements les plus représentatifs : PM^2 et ProActive PDC.

3.4 MPI

MPI[3] est une spécification d'interface de passage de message et non pas une implantation particulière d'une bibliothèque de communication. MPI a été définit par MPI Forum, rassemblant des industriels, des équipes de recherche et des utilisateurs. Elle est le fruit d'une négociation qui a abouti à la spécification d'une interface aux nombreuses possibilités et adaptée à des applications diverses.

MPI assure le déploiement des processus en utilisant le modèle d'exécution SPMD. La bibliothèque fournit plusieurs mécanisme de gestion de message entre les processus. La communication peut être synchrone ou asynchrone. La synchronisation est assurée par des barrière. MPI permet aussi une gestion de groupe de groupe de processus.

3.5 PM^2

PM^2[3], conçu à ENS-Lyon, est un environnement multithread distribué développé à l'origine pour supporter de manière efficace et portable l'exécution d'applications irrégulières à parallélisme massif. PM^2 repose sur une bibliothèque de threads utilisateurs appelée Marcel[9], implanté au-dessus d'une couche de communication appelée Madeleine[10].

Marcel : Marcel fournit une interface de programmation basée sur la norme POSIX semblable à celle de la bibliothèque Pthreads. Mais elle introduit également des fonctionnalités liées permettant la migration de threads et l'exploitation efficace des machines multiprocesseurs à mémoire partagée. Marcel se décompose en trois couches. La première, est une interface utilisateur. La deuxième est un espace où se fait l'exécution concurrente des threads. L'ordonnancement de ces threads est assuré par une couche qui se décompose en un certain nombre de threads noyaux. Le noyau se charge de

[3]MPI : Message Passing Interface

repartir les threads noyau (les ordonnanceurs)ce qui implique une répartition équilibré des threads Marcel sur les différents processeurs. Actuellement, Marcel est disponible sur les architectures suivantes : Sparc, x86, Alpha, PowerPC, Mips et RS6000.

Madeleine : La bibliothèque de communications de PM^2 a été développée pour assurer la portabilité de l'environnement sur un grand nombre d'interfaces de communication. Elle fournit une interface optimisée pour les opérations du type appel de procédures à distance au dessus de protocoles et bibliothèques "classiques" (TCP, MPI) ainsi que sur les interfaces de communications des réseaux hautes performances (BIP, VIA ou SISCI). Madeleine[10] permet la transmission zéro-copie sur des réseaux à hautes performances comme Myrinet ou SCI. Actuellement, les travaux sur Madeleine s'orientent vers le support des communications dans des architectures hétérogènes.

3.6 ProActive PDC

ProActive[4] du projet Oasis à Sophia Antiopolis, est une bibliothèque Java, conçue pour la programmation parallèle, distribuée et concurrente. Elle est uniquement constituée de classes Java standards, et ne requiert aucune modification de la Machine Virtuelle Java. Elle est basée sur la bibliothèque Java RMI standard.

La motivation de ProActive est de pouvoir créer et communiquer avec un objet appelé *Objet Actif* en rendant transparent la localisation et sans se soucier la bibliothèque de communication utilisée. ProActive est interfacée avec RMI, mais aussi avec Jini[11] et Globus[12].

Chaque *Objet Actif* est manipulable comme un objet Java standard auquel est ajouté un thread Java pour l'exécution et une file d'attente pour les requêtes entrantes. Une requête est un appel externe à une méthode publique de l'*Objet Actif*. Il y a aucune restriction sur la méthode à exécuter à distance ni sur ses arguments.

Les communications entre *Objets Actifs* sont asynchrones par message (Request and reply). La synchronisation entre les objets est assurée en utilisant la notion de groupe d'objets. ProActive offre aussi le mécanisme d'*attente par nécessité* qui garantit une attente bloquante d'un objet si ce dernier n'a pas encore pris sa valeur finale. ProActive permet la migration des *Objets Actifs* entre les JVMs.

3.7 Conclusion

Le début de ce chapitre est consacré à la justification de l'utilisation des environnements de calcul distribué. Nous avons présenter trois environnements : MPI, PM^2 et ProActive PDC. Leurs mécanismes de communication, de synchronisation et de gestion des processus distribués sont variés et différents.

Dans le chapitre suivant, nous présenterons SSCRAP qui est une implantation des modèles parallèles à gros grain. Cette implantation est le support des expériences et de validation des modèles. Notre but est d'interfacer SSCRAP avec les supports cités afin de couvrir les architectures hétérogènes.

Chapitre 4

La bibliothèque SSCRAP

4.1 Introduction

Pour supporter les modèles déjà introduits dans le chapitre 2, plusieurs bibliothèques ont été développées. La bibliothèque "The Oxford BSP library"[13] proposée en 1993 fut la première bibliothèque implantant le BSP. Elle a été enrichie par plusieurs extensions et elle a abouti à une nouvelle bibliothèque la "BSPlib"[14]. Le modèle CGM, plus récent est plus simple que le modèle BSP, mais ne dispose encore d'aucune bibliothèque de communication propre. La bibliothèque SSCRAP était initialement conçue afin de fournir une interface de communication complète implantant le modèle CGM. De nouvelle considérations ont été prise en compte afin de l'adapter à une large gamme de modèles existants.

Nous présentons les motivations de SSCRAP (section 4.2). Puis, nous présentons ses principales fonctionnalités. A savoir, les communications (section 4.3) et les synchronisations de processus (section 4.4). Nous détaillons la flexibilité de SSCRAP à supporter plusieurs modèles de programmations (section 4.5).

4.2 Motivations

Les modèles parallèles à gros grain BSP, CGM et PRO possèdent de nombreux points communs. Chacun de ces modèles est constitué de l'union de trois sous-modèles : un modèle d'architecture abstraite, un modèle d'exécution et un modèle de complexité. Ils adoptent le même modèle d'architecture et le même modèle d'exécution. Ils décrivent les algorithmes comme étant une séquence de "super-étapes". Durant chaque super-étape les processeurs commencent par traiter les données disponibles dans leurs mémoires locales. Ils effectuent ensuite les échanges pour récupérer les données non locales nécessaires à la phase de calcul (traitement) de la prochaine super-étape qui ne débutera qu'au bout d'une (dernière) phase de synchronisation.

C'est l'ensemble de ces constatations qui a motivé l'élaboration de la bibliothèque SSCRAP. Elle est conçue dans le but de fournir un cadre de développement recouvrant les différents modèles gros grain et garantissant la portabilité des programmes produits sur une large gamme de plates-formes parallèles.

4.3 La communication dans SSCRAP

La fonction principale de SSCRAP est de gérer les communications qu'effectuent les processeurs lors de l'exécution d'un programme. Les plus importants types de communications discernés sont :

- chaque processeur communique avec tous les autres (all-to-allV) sans connaissance préalable des tailles des données échangées ;
- chaque processeur envoie à tous les autres (all-to-all) des messages de tailles fixes et réduites ;
- un processeur envoie à tous les autres (one-to-all) des messages de tailles fixes et réduites ;
- un processeur envoie à un autre (point-à-point) un message de taille fixe.

Le premier type de communication est de loin le plus sollicité par les algorithmes gros grain. En effet, il n'est généralement pas possible de connaître, avant l'exécution de l'algorithme, le nombre de données et quelles données à envoyer à un processeur donné. Ceci dépend en fait des entrées initiales de chaque processeur et du mécanisme de redistribution des données sur les processus à la fin de chaque phase de calcul. On nommera ce type **communications générales**.

Les principales difficultés de l'implantation des communications générales résident dans le fait que SSCRAP doit non seulement gérer un nombre important de gros messages ($p(p-1)$ ou p est le nombre de processeurs) mais aussi prendre en compte l'ordonnancement imposé par l'algorithme pour les données reçues.

L'approche adoptée par SSCRAP pour implanter les communications générales, est basée quand à elle sur l'allocation dynamique des buffers d'émission et de réception à la taille exacte des données qu'ils vont contenir.

4.4 Les synchronisations dans SSCRAP

La bibliothèque SSCRAP implante la synchronisation globale ou classique durant laquelle chaque processeur se bloque jusqu'à ce que tous les autres atteignent le même point de synchronisation.

SSCRAP fournit aussi deux autres types de synchronisations : la synchronisation d'émission et la synchronisation de réception.

La synchronisation d'émission : les processeurs se bloquent jusqu'à ce que tous les messages à envoyer soient pris en charge par la couche de communication : passage de contrôle à la couche communication.

La synchronisation de réception : les processeurs se bloquent jusqu'à ce que tous les messages venant de la couche de communication soient chargés dans leurs buffers de réception respectifs : passage du contrôle au processeur.

Á la suite d'une synchronisation d'émission, le processus SSCRAP peut réutiliser les données qui ont été émises ou encore libérer l'espace qu'elles occupaient. La synchronisation de réception est plus importante. En effet, après l'appel de la routine correspondante à ce type de synchronisation, le processus SSCRAP peut entamer une nouvelle phase de calcul utilisant les données qu'il attend des autres.

4.5 Flexibilité de SSCRAP

Dans le but de fournir un environnement de développement pour les modèles parallèles gros grain, la bibliothèque SSCRAP doit permettre l'implantation de leurs modèles d'exécution respectifs. Ainsi, tout programme s'exécutant sur SSCRAP est décrit comme une suite de super-étape. Une super-étape SSCRAP contient une phase de calcul, une phase de communication et une phase de synchronisation. Le modèle de délégation de contrôle cité précédemment, permet non seulement d'implanter les modèles d'exécution CGM et BSP mais aussi de définir de nouveaux modèles d'exécution pouvant par exemple assurer un recouvrement entre communication et calcul .

Une super-étape SSCRAP se déroule comme suit :

Modèle d'exécution BSP : calcul (algorithme), envoi/réception (communication générale SSCRAP), synchronisation (synchronisation globale SSCRAP).

Modèle d'exécution CGM et PRO : calcul (algorithme), envoi/réception (communication générale SSCRAP), synchronisation (synchronisation de réception SSCRAP).

4.6 Conclusion

Dans ce chapitre, nous avons présenté la bibliothèque SSCRAP, ces motivations et ces fonctionnalités de communication et de synchronisation. La flexibilité de la bibliothèque permet de supporter facilement les différents modèles de programmation parallèle à gros grain. Cette caractéristique fait d'elle un outil d'experimentation et donc de comparaison entre différents algorithme écris selon diffèrent modèles.

La conception modulaire en couche de la bibliothèque la rend évolutive et efficace. Ceci est prouvé en la comparant avec d'autre bibliothèques comme BSPlib ou CGMlib.Actuellement, SSCRAP est interfacée avec les deux types d'architectures parallèles homogènes : celles à mémoire partagée et celles à mémoire distribuée.

Dans le chapitre suivant nous allons présenter et analyser ces deux interfaces, afin de pouvoir entamer une modélisation de l'hétérogénéité pour le calcul parallèle à gros grain.

Chapitre 5

Analyse des solutions pour les architectures homogènes

Dans ce chapitre, on va commencer par définir les architectures homogènes qui nous intéressent (section 5.1). Puis on présentera les différentes solutions adoptées pour profiter des particularités de ces architectures (section 5.2 et 5.3). Pour chacune des solutions, nous analysons les méthodes adoptées pour la mise en œuvre de modèles parallèles à gros grain. Notre cadre d'expérimentation est la bibliothèque SSCRAP.

5.1 Les architectures homogènes

Les architectures homogènes visées par SSCRAP sont surtout les architectures MIMD. Dans ses machines, chaque processeur possède un flot d'instructions indépendant. Aucune contrainte de synchronisation n'est imposée lors des exécutions parallèles. Deux types d'architecture de machines de cette catégorie sont classiquement distingués suivant que les processeurs ont ou non directement accès à toute la mémoire.

5.1.1 Les machines multiprocesseurs à mémoire partagée(SMP)

La principale caractéristique des machines SMP[1] est le fait que les processeurs peuvent accéder directement à toute la mémoire. Si l'accès est en temps constant, la machine est dite UMA[2]. Sinon, elle est dite NUMA[3]. Les machines multiprocesseurs à mémoire partagée ont généralement un handicap lié à leur architecture : la mémoire commune est souvent accédée via un bus unique, qui peut devenir un sérieux goulot d'étranglement et augmenter fortement le coût des accès mémoire. Cet effet est d'autant plus gênant que le nombre de processeurs connectés au bus est grand. Par conséquence, ce type d'architecture ne pourra être efficace qu'avec un faible nombre de processeurs, ce qui impose une sévère limitation au degré de parallélisme réellement exploitable par la machine.

[1]SMP : Symmetric MultiProcessor
[2]UMA : Uniform Memory Access
[3]NUMA : Non Uniform Memory Access

Afin d'éviter le goulot d'étranglement de la mémoire, des caches importants sont ajoutés. Dans les premières réalisations de ce type de machines, le matériel n'assurait pas la cohérence des caches. Depuis, les machines avec cohérence de cache matériel sont apparues. Le préfixe cc (cache cohérent) est utilisé pour les reconnaître (SMP cc-NUMA).

5.1.2 Les machines à mémoire distribuée

Dans ce type d'architecture, chaque processeur n'a accès qu'à sa mémoire locale. Les processeurs communiquent entre eux par messages via un réseau pour accéder à la mémoire des autres processeurs. L'évolution de ce type de machines s'est concentrée sur l'architecture et les performances brutes du réseau. Suivant les évolutions technologiques, les machines se sont dotées d'une entité spécialisée pour la gestion des communications. L'objectif était de ne pas surcharger le processeur principal avec le coût du traitement de routage et de réception des messages. Les communications sont réalisées au niveau bas par passage de messages. Dans la plupart des implantations, ce mode de communication est directement utilisable par le programmeur.

5.2 Les machines SMP et SSCRAP

Afin d'exploiter la ressource mémoire partagée, SSCRAP utilise une interface basée sur les threads PosiX. En effet, les threads, permettant le partage de tout l'espace d'adressage se sont avérés un moyen adapté pour l'échange de messages de tailles importantes.

Pour mettre en œuvre le modèle d'exécution SPMD, il suffit de créer les threads et leur donner la même fonction à exécuter. Cette dernière commencera par une barrière de synchronisation entre tous les threads.

Un thread PosiX est ordonnancé par un thread noyau qui est créé et géré par le système. Cette caractéristique assure une parallélisation réelle de l'application. En effet, en cas de présence de plusieurs threads dans une machines multiprocesseurs, le système essaye automatiquement d'équilibrer les threads sur les différents processeurs de la machine.

l'utilisation de la norme PosiX garantit aussi la portabilité de cette interface. Pour utiliser un algorithme écrit au dessus de cette interface avec une autre architecture, l'utilisateur n'a qu'a recompiler le code source sans aucune modification.

La gestion des messages entre les processus est faite à travers un "objet partagé". Cette dernière est vue par tous les processus et contient tous les messages en attente avec leurs caractéristiques. Cette approche permet une optimisation fine de gestion des buffers. Cela est possible puisque l'interface doit gérer tous les messages, au contraire de MPI qui le fait automatiquement.

La communication est assurée par les primitives élémentaires de copie de zone mémoire. L'interface garantit le "zéro copie" intermédiaire entre la zone source et sa destination finale.

La synchronisation entre les processus est mise en place à l'aide des primitives de synchronisation offertes par les threads PosiX. En fait, ces derniers proposent comme

mécanismes les sémaphore, l'exclusion mutuelle avec une attente passive.

Enfin, l'interface garantit un accès concurrent à l'"objet partagé". Ce type d'accès fait le compromis entre l'utilisation simultanée de l'objet et la réduction de durée de verrouillage des processus.

5.3 Les machines à mémoire distribuée et SSCRAP

Pour exploiter les machines à mémoire distribuée, SSCRAP utilise une interface basée sur la bibliothèque à passage de message MPI.

L'utilisation de ce genre de bibliothèque a deux avantages :

- Codage simple : MPI offre tous les outils nécessaires pour nos besoins
- Interface SSCRAP bénéficie de toute évolution de la bibliothèque MPI. En effet, SSCRAP laisse à l'utilisateur le choix de l'implantation de MPI qu'il souhaite utiliser. Par exemple pour Origin2000, SSCRAP peut exploiter LAM, MPICH ou SGIMPI.

Toutefois, l'utilisation de MPI présente l'inconvénient majeur de gestion de mémoire. En effet, les expériences montrent qu'au delà de 16 processus, le sur-coût lié à la couche communication et gestion de message devient très coûteuse.Ce problème est dû à la gestion automatique de messages et aux copies intermédiaires générées.

5.4 Conclusion

L'implantation de SSCRAP avec la mémoire partagée, bien qu'elle aboutit à des résultats intéressants comparée à celle avec MPI, reste adaptée pour les machines SMP. Cette dernière ne supporte pas plus de 32 processeurs avec la machine Origin2000, cela est dû au goulot d'étranglement au niveau d'accès mémoire.

Construit sur MPI, SSCRAP touche une grande gamme d'architectures, mais présente des limites :

- Elle reste toujours inadaptée aux réseaux comportant des nœuds SMP, malgré les dernières motivations de MPI FORUM de faire porter efficacement MPI aux machines SMP ;
- MPI n'utilise pas au mieux les performances et les services offerts par les réseaux existants, comme le mode DMA de VIA. En plus, elle ne supporte pas les interface de communication bas niveau les plus récents et les plus performants comme BIP et SISCI ce qui provoque la non adéquation de SSCRAP aux réseaux Myrinet, SCI, etc...
- L'émergence d'une forte utilisation des architectures hétérogènes qui ne sont pas bien exploitées par les interfaces citées précédemment.

Ce sont ces raisons qui ont motivé l'extension des modèles de calcul parallèle aux architectures hétérogènes. Cela est nécessaire afin de couvrir le maximum des machines disponibles et de tester les modèles pour valider ou invalider leurs utilisations avec ce genre de machines.

Chapitre 6

Modélisation pour les architectures hétérogènes

Ce chapitre est consacré à l'analyse des architectures hétérogènes(section 6.1). Une modélisation sous forme de graphe de ces dernières est proposée(section 6.1.3). En s'appuyant sur ce modèle, nous présentons notre approche pour la répartition des processus sur les différents processeurs(section 6.2). Puis, nous proposons un modèle pour l'échange de données entre les processus(section 6.3). Enfin, une caractérisation de la synchronisation sera faite (section 6.4).

6.1 Modélisation de l'architecture

6.1.1 L'hétérogénéité

L'hétérogénéité au niveau architecture peux être vue sous plusieurs angles. Mais nous pouvons énumérer les caractéristiques sur lesquelles repose une architecture quelconque. Une machine parallèle peux être vue comme plusieurs processeurs interconnectés entre eux par un réseau. Chaque processeur est doté d'une mémoire cache et d'une mémoire centrale. Cette dernière, peut être la même pour plusieurs processeurs.

Chacun de ses éléments a ses caractéristiques :

- Processeur : un processeur est plus performant qu'un autre s'il peut exécuter un plus grand nombre d'instructions flottantes par unité de temps.
- Cache ou antémemoire : une antememoire est accédée à la même fréquence que celle du processeur, elle minimise l'accès à la mémoire centrale. Elle doit fournir un grand débit avec une taille maximum.
- Mémoire centrale : comme l'antémemoire, elle doit faire le compromis entre sa taille et la vitesse de lecture par le processeur.
- Réseau d'interconnexion : les caractéristiques les plus importantes d'un réseau sont le débit et la latence. Il doit maximiser le débit et de minimiser la latence.

Une classification possible est de différencier les machines selon que la mémoire centrale est commune au processeurs ou non. Si elle est commune, la machine sera de type SMP sinon, elle sera une machine à mémoire distribuée. Il faut noter que ses machines sont homogènes.

On parle d'hétérogénéité si une machine parallèle contient deux ou plusieurs composantes dissymétriques. Par exemple les grappes SMP, bien qu'elle soient composées de mêmes nœuds et connectés par le même réseau d'interconnexion, est qualifiée d'hétérogène parce que d'une part, il y a deux types de réseau d'interconnexion entre les processeurs, le réseau en bus et le réseau haut débit. D'autre part, deux processeurs sur le même nœud accèdent à la même mémoire alors que deux processeurs sur deux nœuds différents n'ont pas la même mémoire.

6.1.2 Analyse préliminaire

Avec les composantes élémentaires introduites dans la première section on peut prévoir plusieurs architectures. En gros il existe cinq grands types : les machines DataFlow et Systolique, les SIMD, les Vectoriels, les MIMD et les Clusters (ou Grappes).

Les quatre premières sont des machines homogènes. Un cluster est un ensemble de nœuds reliés par un réseau. Les nœuds peuvent être monoprocesseurs ou de type SMP. Dans le premier cas, le cluster peut être vue comme homogène, puisque toute communication entre les processeurs est assurée par le réseau (passage de message ou RPC). Dans le deuxième cas, le cluster est hétérogène.

Puisque les machines MIMD sont soit SMP soit à mémoire distribuée, et comme la communication dans une machine MIMD à mémoire distribuée se fait par message, on peut confondre cette machine avec un cluster de nœuds monoprocesseurs.

En conclusion, seul le cluster de nœuds SMP est une machine hétérogène. Cette machine comprend des nœuds SMP de tailles différentes éventuellement monoprocesseurs, le réseau d'interconnexion peut aussi être différent entre un couple de nœud et un autre. Ce type d'architecture est généralement nommé Grille.

6.1.3 Modélisation commune de l'architecture

Le but de cette modélisation n'est pas de classer le maximum de machines parallèles dans un minimum de classes, mais de donner un modèle d'architecture générique qui englobe le maximum de machines utilisées par le parallélisme à gros grain.

Les machines prises en compte sont les grilles avec des nœuds mono ou multiprocesseurs interconnectés par un réseau qui peut être diffèrent d'un couple de nœud à un autre.

Une modélisation réaliste de l'architecture doit dépendre du type d'applications qui vont l'exploiter. Dans notre cas, les applications sont de type gros grain. Comme déjà introduit dans le chapitre 2, un algorithme à gros grain est une succession d'étapes de calcul et de communication. Le calcul est généralement fait sur des données de grandes tailles, non consécutives et aléatoires. Donc l'utilisation de la mémoire centrale est très fréquent par rapport aux accès à l'antémemoire. On fera une abstraction de cette dernière en prenant seulement la fréquence du processeur et la taille de la mémoire centrale pour caractériser un nœud.

Dans la phase de communication, deux points sont à prendre en compte. La première est la taille importante des données à émettre. En effet, en parallélisme à gros grain la taille des données est généralement de l'ordre de la taille de la mémoire. La deuxième est

le type de communication prépondérant qui est dans notre cas un all-to-all où chaque processeur envoi un tableau de taille différent à chaque autre processeur. Dans ce cas, c'est le débit du réseau qui est déterminant.

Une représentation possible et réaliste d'une architecture hétérogène sera un graphe complet où les nœuds sont les processeurs et les arrêtes sont les interconnexions entre les processeurs. Chaque nœud aura un poids qui correspond à la fréquence du processeur associé. Le poids d'une arrête sera le débit de la connexion. La complètude du graphe garantit l'hypothèse faite par les modèles où chaque processeur admet une communication point à point avec les autres.

Une machine SMP sera représentée par un graphe complet où les nœuds sont similaires. Les poids des arrêtes sont égaux et correspondent à la vitesse du bus divisé par le nombre de processeurs. un nœud du graphe prend comme caractéristique de la mémoire la taille totale de la mémoire de la machine SMP divisé par le nombre de processeurs de la machine.

Une grappe de deux machines biprocesseurs sera représentée par un graphe. L'arête qui correspond au lien de deux processeurs de deux machines différentes a pour poids le débit de la connexion entre le deux machines divisé par le nombre de processeurs qui est quatre dans ce cas.

6.2 Le modèle de déploiement

Étant donné la modélisation présentée dans la section 6.1.3, un nombre p de processus, le but de cette section est de donner une meilleure répartition des processus sur les différents processeurs qui maximise l'efficacité de l'algorithme en utilisant les ressources machine de façon optimisée.

La répartition optimale doit garantir :

- L'utilisation de la mémoire partagée pour la communication entre les processeurs se trouvant dans le même espace d'adressage. Ceci est destiné à une utilisation efficace des machines SMP
- l'utilisation du réseau si les processus se trouvent dans deux nœuds distants.

L'idée est de faire rassembler les processus se trouvant dans un nœud en un seul processus père. L'utilisation des threads pour l'échange via une mémoire partagée localement parait l'approche la plus adaptée à nos besoins. En effet, le partage du même espace d'adressage pâlie l'inconvénient d'allocation d'une zone intermédiaire d'échanges. Cette méthode nous permet aussi de faire des échanges avec zéro-copie puisque le threads émetteur a accès à l'espace d'adressage du récepteur. Il peut donc écrire les données à transmettre directement dans leurs espaces des destinations.

Sur le graphe complet représentant l'architecture, ce modèle de déploiement correspond à identifier les nœuds SMP en utilisant le poids de l'arête (le débit) et mettre sur chaque nœud du graphe un processus. Puisque tous les processus exécutent le même algorithme, les tailles des mémoire utilisées par les processus sont souvent égaux. Par suite, la taille de la mémoire propre à chaque processus sera le minimum des tailles du sous-graphe utilisé pour le déploiement.

6.3 Modélisation de la communication

6.3.1 Routines d'échanges utilisateurs

Rappelons que pour communiquer entre les différents processus le modèle de programmation à gros grain offre à l'utilisateur les échanges one-to-one, one-to-all et all-to-all.

Le type all-to-all est un échange où chaque processus envoie des données à tous les autres et en même temps il s'attend à recevoir des données depuis tous. Cette procédure est donc décomposable en plusieurs échanges de type one-to-one. Cette décomposition ne doit pas faire de l'échange all-to-all une procédure séquentielle qui va exécuter en série plusieurs procédures one-to-one, mais elle doit maximiser le parallélisme d'échange. Une solution pour cela sera de créer autant de threads que d'échanges one-to-one.

6.3.2 Échange point-à-point

Á ce stade, nous allons confondre les échanges de données vue du coté utilisateur et les échanges de contrôle. On suppose aussi que tout échange est asynchrone. On appellera une procédure de synchronisation sur les données en cas d'échange synchrone.

Un échange est caractérisé par un émetteur, un récepteur et des données à transmettre. Chacun des deux acteurs de l'échange peut commencer sa phase d'échange à tout instant. Aucune synchronisation n'est envisagée au niveau utilisateur lors du démarrage de l'échange. Il est donc possible d'assister donc à un échange où l'émetteur commence avant le récepteur et inversement. Pour garantir un échange avec zéro-copie, une synchronisation entre les deux acteurs est nécessaire.

L'échange réel ne doit commencer que si l'émetteur est prêt à envoyer et le récepteur a déjà alloué l'espace pour recevoir les données. Ceci ne doit pas faire perdre la caractéristique d'asynchronisation de l'émission et celle de la réception.

La procédure de communication se déroulera comme suit :

- après allocation d'une zone mémoire pour stocker les données à recevoir, le récepteur notifie l'émetteur qu'il est **prêt à recevoir**. Pour une communication asynchrone, un thread est créé pour chaque notification. Deux cas sont possibles :
  1. La notification arrive à l'émetteur après qu'il ait exécuté la procédure d'émission. Dans ce cas, la notification débute l'échange effectif des données.
  2. La notification arrive avant que l'émetteur ait exécuté la procédure d'envoi. Dans ce cas, la notification initialise un drapeau qui informera la procédure d'envoi que le récepteur est prêt à recevoir.
- La procédure d'envoi étant lancée, vérifie si le récepteur a initialisé le drapeau de notification ou non. Si oui, elle commence l'envoi réel. Sinon elle initialise de sa part un drapeau qui informera le thread de notification que l'émetteur est prêt à envoyer.

6.4 Modélisation de la synchronisation

6.4.1 Synchronisation par Barrière

On peut assurer ce mécanisme de deux façons : centralisée ou hiérarchique. Dans le centralisé, la barrière est assurée comme suit :

- Chaque processeur envoie une notification à un processus élu, et attend un message pour continuer son exécution.
- le processus élu attend les notifications de tous les autres et leur envoie un message pour les débloquer.

Comme toute solution centralisée, cet approche est simple à implanter mais coûteuse. En effet, on risque d'avoir un goulot d'étranglement du coté du processus élu.

Une solution plus adaptée à nos besoins est l'approche hiérarchique :

- Une barrière de synchronisation au niveau de chaque nœud
- Une barrière de synchronisation internœuds

Cette approche assure une synchronisation globale en minimisant la charge du processus élu

6.4.2 Synchronisation de communication

Ce type de synchronisation permet d'assurer, pour un processus PRO donné, que tous les messages qu'il doit envoyer ou recevoir sont pris en compte par la couche communication.

Chaque processus doit sauvegarder toutes ses requête d'envoi et de réception. Supposons A un processus exécutant une super-étape i, B un autre exécutant une super-étape j et que A veut envoyer des données à B. Trois cas sont possible :

- j<i : Ce cas est non envisageable. En effet, l'émetteur A en i ne peut pas passer la super-étape j avant qu'il termine toutes les échange concernant cet étape.
- j=i : Dans ce cas A n'a à sauvegarder qu'une seule requête d'envoi concernant B. Ceci est garantit par les modèles de calcul à gros grain.
- j>i : Dans ce cas, B n'avait pas de requêtes de réception en attente de A pour toutes les étapes entre i et j. Sinon, il va rester bloqué jusqu'à satisfaction de la requête.

Puisque l'envoi et la réception sont symétriques, on peut conclure qu'un processus donné attend au plus une requête de réception et une autre d'envoi d'autre processus.

6.5 Conclusion

Dans ce chapitre, nous avons proposé un modèle pour les architectures hétérogène. Ce modèle est basé sur le modèle d'exécution de type SPMD. Puis nous avons présenté un modèle de communication qui prend en compte la topologie de la machine et le déphasage des processus. Le modèle de synchronisation proposé traite deux mécanismes : la barrière et la synchronisation de communication. Ce premier exploite la localisation des processus, la déphasage entre eux et la topologie de l'architecture.

Chapitre 7

PM^2 pour le calcul parallèle à gros grain

7.1 Introduction

Dans ce chapitre, on va analyser et concevoir une interface qui implante la modélisation proposé dans le chapitre précèdent, afin de porter les modèles à gros grain sur les architectures hétérogènes. Le support choisi pour l'interface est PM^2. On utilisera la bibliothèque SSCRAP comme une implantation des modèles parallèles à gros grain.

Cette interface doit faire le compromis entre le modèle de programmation à gros grain et le modèle de programmation proposé par PM^2.

La section 7.2 sera consacrée au mécanisme de répartition des processus sur l'architecture. Puis on présentera notre approche pour l'échange élémentaire point-à-point (section 7.3). Enfin, La synchronisation entre les processus (section 7.4).

7.2 Répartition des processus

L'idée retenue est de faire rassembler les processus se trouvant dans un même nœud dans un seul processus père. Le but est d'utiliser le même espace d'adressage pour la communication des données et les message de contrôle. Pour utiliser le même espace d'adressage, l'utilisation des threads est imposée. Mais le problème de choix du type des threads se pose.

Le thread noyau bénéficie d'un ordonnancement au niveau système, ce qui garantit une répartition équilibrée sur les différents processeurs de la machine SMP. Mais, cette méthode a des répercussion sur l'exécution concurrente des threads et fait perdre du temps lors d'un changement de contexte. Surtout qu'il s'agit ici d'un modèle d'exécution à gros grain centré sur les données. C'est-à-dire que le temps pris pour exécuter l'algorithme de traitement des données est plus important que celui de communication. On s'attend alors à un goulot d'étranglement au niveau du processeur et donc plusieurs changements de contexte pour une seule étape de calcul.

Les threads MARCEL regroupent les avantages de deux types de threads. Avec son ordonnancement au niveau utilisateur basé sur la préemption et l'exécution en temps partagé avec priorité, MARCEL offre les performances des threads utilisateur lors de la

création des threads et lors du changement de contexte. Comme les ordonnanceurs sont des Posix-Threads, ceci garantit une répartition équilibrée sur les différents processeurs, et engendre ainsi une exécution réellement parallèles des threads MARCEL dans une machine SMP.

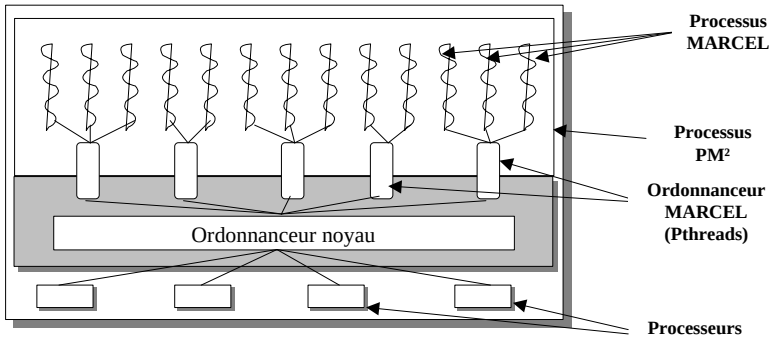


FIG. 7.1 – Ordonnancement équilibrée des processus MARCEL sur le nœud

La solution finale est donc de rassembler tous les processus SSCRAP qui doivent s'exécuter sur un même nœud dans un seul processus PM^2. Les processus SSCRAP seront des threads MARCEL communicant via l'espace d'adressage du contexte PM^2 sans copies intermédiaires.

Une optimisation intéressante sera de forcer MARCEL à placer les processus SSCRAP sur des threads systèmes différents, afin de garantir leur distribution par le noyau sur les différents processeurs du nœud SMP.

7.3 Échange point-à-point

Pour ce type d'échange de données élémentaire, plusieurs solutions ont été analysées. On a retenu la solution qui satisfait ces contraintes :

- Aucune synchronisation n'est envisagée entre le début de la réception et celle d'émission.
- Un échange avec zéro-copie.
- Asynchronisation de la routine d'échange.

Nous proposons d'utiliser les threads issus d'un RPC pour effectuer l'échange. La procédure inspiré du modèle proposé dans le chapitre précédent est la suivante :

1. le récepteur notifie l'émetteur qu'il est **prêt à recevoir**. Deux cas sont possibles :
   - La notification arrive à l'émetteur SSCRAP après qu'il ait exécuté la procédure d'émission. Dans ce cas, c'est le thread issu de la réception qui va initier l'envoi. Le thread issu de la réception est soit celui qui est lancé par le RPC si l'émetteur est distant, soit le thread qui s'est chargé de la réception si l'émetteur est dans le même contexte.

- La notification arrive avant que l'émetteur ait exécuté la procédure d'envoi. Dans ce cas, la notification initialise un drapeau qui informera la procédure d'envoi que le récepteur est prêt à recevoir.

2. La procédure d'envoi étant lancée, teste si le récepteur a initialisé le drapeau de notification ou non. Si oui, elle commence l'envoi réel. Sinon elle initialise de sa part un drapeau qui informera le thread de notification qu'il est prêt à envoyer.
3. L'envoi réel, se résume en une création d'un thread qui va faire soit un RPC, si le récepteur est distant soit une simple copie mémoire, si le récepteur est dans le même contexte.

7.4 Synchronisation

Le modèle choisit pour ce type de synchronisation est le modèle hiérarchique. Divisé en deux barrière, l'un au sein du nœud et l'autre inter-nœud.

Avec PM^2 et afin de concevoir la barrière locale, l'utilisation des sémaphores est adapté. En effet, elle garantit un temps minime pour le traitement de la routine et aussi une attente passive des processus.

La procédure se déroulera comme suit :

- chaque processus appelant la routine de synchronisation globale, incrémente une variable et se met en attente passive.
- le dernier qui arrive initie la barrière inter-nœud.
- ayant fini la barrière inter-nœud, chaque nœud reçoit une notification. Dans ce cas le thread issu de cette notification se charge de réveiller tous les processus locaux.

La barrière inter-nœud est centralisé. Le choix d'un gestionnaire est nécessaire. On va prendre par exemple le processus de plus petit numéro. Cet élu doit attendre une notification de tous les processus, de lui même aussi, afin d'envoyer une réponse à chaque nœud. L'utilisation des sémaphores au niveau du gestionnaire est aussi possible. Chaque thread issu d'un RPC de notification active la sémaphore qui doit être protégée pour une utilisation exclusive.

Après création de plusieurs threads qui se chargent des envois effectifs, la synchronisation d'émission et de réception consiste simplement à l'attente de la terminaison de ces threads. On utilisera le mécanisme de sémaphore pour les contrôler et les synchroniser.

7.5 Conclusion

Dans ce chapitre, nous détaillons les concepts de base retenus par la mise en œuvre des différentes fonctionnalités requises par SSCRAP avec les outils offerts par PM^2 ; thread MARCEL et RPC. L'implantation de cette interface a été faite sans problèmes. Cela est dû à la transparence de la conception et le modèle flexible proposé par PM^2. La phase de test a aboutit à une de-validation de cette interface. Les arguments suivants ont été pour cause :

- Une exécution non stable de l'interface. Deux exécutions avec les mêmes paramètres et dans les mêmes circonstances donnent deux durée d'exécution différentes.
- PM^2 n'est pas adapté aux types de communication all-to-all. En effet, lors de ce genre de communication, il y a un goulot d'étranglement sur le port de communication. Cela est dû au nombre important de connexion ouverte. Lors d'une communication entre un ensemble de processus se trouvant dans un nœud SMP et un autre, PM^2 ouvre autant de port de connexion que de couple de processus.
- Le problème précédent est accentué lors d'un échange de donnée de taille importante. Ceci est fréquent dans un algorithme à gros grain.

Toutefois, l'utilisation de PM^2 a été bénéfique sur plusieurs plans. En effet, à long terme, une conception d'un support de calcul parallèle propre à notre support d'expérimentation est envisageable. Pour ce support, l'utilisation du modèle de programmation en service de PM^2 sera étudiée en plus de l'approche adoptée pour exploiter efficacement les machines hétérogènes, et précisément celles comportant des machines SMP.

Chapitre 8

ProActive pour le calcul parallèle à gros grain

8.1 Introduction

ProActive est le deuxième support de calcul parallèle choisi pour mettre en œuvre les modèles proposé dans le chapitre 6. Puisque ProActive est une API écrit en Java et que notre support d'expérimentation est écrit en C++, nous avons eu des problèmes d'intégration de deux bibliothèques. La section 8.2 sera réservée pour l'analyse des différentes méthode pour utiliser ProActive dans notre cadre de travail. Dans le section 8.3, nous mettrons en œuvre le déploiement des processus. Puis, nous détaillerons notre approche pour la communication (section 8.4) et la synchronisation (section 8.5). Enfin, nous présentons notre plate-forme de test(section 8.6.1) et les interprétations de nos résultats (section 8.6.2).

8.2 Utilisation de ProActive

ProActive est une API écrite totalement en Java. Elle bénéficie donc de tous les avantages de Java : portabilité, programmation modulaire, réutilisation des composantes, programmation côté utilisateur simple, richesse en fonctionnalités et la richesse de la documentation. De plus ProActive est une extension de Java pour une utilisation simple pour les architectures distribuées. Tous ces avantages de ProActive ont motivé son utilisation pour concevoir une interface qui met en œuvre la modélisation faite dans le chapitre 6.

De l'autre côté, notre support d'expérimentation SSCRAP est une bibliothèque écrit en C++. La différence des langages pose le problème de conflit entre eux et le problème d'intégration. Plusieurs idées ont été proposées pour l'utilisation de ProActive dans le cadre de la parallélisme à gros grain :

- Intégrer SSCRAP et ProActive : cette solution consiste à écrire une interface en ProActive. Cette interface a pour but d'exploiter les ressources des machines hétérogènes, puis d'intégrer le code C++ de SSCRAP dans celui de l'interface en

Java. Pour cela Java propose JNI[1] qui est une interface d'intégration de code. Cette solution souffre de deux inconvénients l'inefficacité et l'irréalisme. Inefficacité car l'interface gaspillera le temps en basculant d'un code à un autre. Irréalisme puisqu'elle ne reflétera pas les capacités de l'architecture.
- Réécrire SSCRAP en Java. Cette solution prendra beaucoup de temps et peut faire perdre les avantages d'utilisation de C++ (l'efficacité et l'utilisation des Template C++).
- La solution adoptée dans notre travail consiste à écrire une mono-bibliothèque qui implante l'essentiel des modèles parallèles à gros grain, à savoir le déploiement des processus, le contrôle du modèle en super-étape, la communication, la synchronisation et les statistiques. Dans ce qui suit nous allons implanter le modèle à gros grain PRO.

8.3 Deploiement des *Objets actifs*

L'API ProActive est basée sur les *Objets actifs*. Chacun de ces derniers peut invoquer des méthodes sur n'importe quel objet local ou distant (un autre *objet actif*). Pour cela, il suffit de détenir une référence sur cet objet.

Notre approche consiste à créer pour chaque processus PRO un *objet actif* abstrait qui lui correspond. Cet objet a pour rôle d'assurer la gestion des messages, le contrôle de la bonne exécution du modèle en super-étape, le transfert des données avec les autres *Objets actifs*, la synchronisation entre eux, le calcul de paramètres d'exécution et de statistique et enfin d'offrir une interface utilisateur simple et flexible pour le calcul, la communication et la synchronisation.

Un algorithme utilisateur doit correspondre à une classe qui hérite de l'*objet actif*. Il n'y aura aucune restriction sur la classe utilisateur sauf qu'elle doit implanter une méthode abstraite qui remplacera la fonction *main* principale.

Pour lancer son algorithme, l'utilisateur doit lancer une application qui prend en argument le nombre des processus PRO à lancer, leurs localisations, les paramètres de sécurités des *Objets actifs*. Cette application génère un fichier XML[2] qui sera utilisé par notre chargeur des processus PRO.

Lors de chargement des processus PRO dans une machine SMP, le choix entre mettre tous les processus dans une seule JVM[3] ou de créer pour chaque processus une JVM à part est à faire. Aucun paramètre ne permettait ce choix, nous avons donc étendu notre interface pour donner à l'utilisateur cette possibilité.

Il est à noter qu'avec cette approche nous n'utilisons pas la mémoire partagée pour la communication entre les processus PRO. Cette limite est dû à ProActive qui ne donne pas la possibilité de partager des données entre les *objets actifs*.

Afin d'exploiter la mémoire partagée, nous proposons d'utiliser non plus un *objet actif* pour chaque processus PRO, mais un Thread Java pour ce dernier. En exploitant la visibilité de la mémoire commune aux threads se trouvant dans la même JVM, on peut partager et échanger des données via cette JVM. Le problème majeur de

[1]JNI : Java Native Interface
[2]XML : eXtensible Markup Language
[3]JVM : Java Virtual Machine

cette approche est le coût en temps de réalisation. En effet, pour utiliser les threads Java, l'utilisateur doit se charger de leurs ordonnancements. Il faut donc concevoir et implanter un ordonnanceur propre à notre interface.

8.4 Communication

L'échange de données entre processus PRO doit garantir l'asynchronisme des routines, la synchronisation entre les deux acteurs de l'échange, la concurrence en cas d'un échange par groupe de processus et l'envoi d'une partie d'un tableau. L'asynchronisme des routines d'envoi et de réception est assuré par ProActive lui même. En effet tout échange entre *Objets actifs* est asynchrone.

Pour garantir la synchronisation entre le récepteur et l'émetteur, nous avons adopté le mode de communication "*pull*" ou la procédure d'échange se déroule comme suit :

- Le récepteur notifie l'émetteur qu'il est "**prêt à recevoir**". La notification exécutée côté émetteur commence la réception réelle des données si l'émetteur est prêt.
- La procédure d'envoi étant lancée côté émetteur, elle teste si le récepteur a invoquer la notification ou non. Si oui, elle commence l'envoi réel.

La concurrence d'échange est assuré par la couche basse de communication RMI de ProActive. L'utilisation de *file d'attente* par ProActive permet de filtrer les requêtes en attente d'exécution selon le nom de la méthode, les valeurs des arguments et l'ordre d'arrivée des requêtes. Ce filtrage nous permet de donner des priorités aux requêtes. On peut utiliser un filtrage sur les méthodes pour satisfaire les notifications des récepteurs avant celles de synchronisation.

Pour l'envoi des tableaux et puisque Java passe tous les arguments des méthodes par copie bit-à-bit, nous étions obligé de sauvegarder une copie de chaque partie des tableaux à envoyer. Cette méthode a comme inconvénients l'utilisation excessive de la mémoire et la perte de temps pour l'allocation mémoires et la copie. Mais elle garantit une optimisation lors de l'envoi des données, surtout si le tableau initial est de grande taille.

8.5 Synchronisation

Appelant la routine de *barrière globale*, le processus PRO devrait attendre tous les autres jusqu'à ce qu'ils fassent pareil. Pour assurer ce mécanisme, notre conception est la suivante :

- Chaque processus invoque la méthode "*WaitForSignal*" du processus numéro zéro (par exemple). Puis il se bloque en attente d'une requête du processus zéro pour continuer l'exécution.
- Le processus zéro attend toutes les requêtes "*WaitForSignal*". Puis il invoque une méthode qui signale la fin de la barrière de synchronisation.

Les méthodes d'attente et de signalisation servent seulement pour la notification, leurs corps ne contiennent aucune instruction à exécuter.

La synchronisation de réception et d'émission consiste à satisfaire toutes les requêtes de communication le plus tôt possible. Lors du choix de la requête externe à exécuter, des priorités sont ajoutées à chaque type de méthode. Cela afin de satisfaire les notifications et les messages de contrôle avant celle des échanges effectifs. Cette méthode ralentit la transmission des données entre les processus mais elle prépare le maximum des message a transmettre et laisse la couche RMI gérer le débit et la latence du réseau.

8.6 Tests et Interprétation

8.6.1 Algorithme et architecture de test

La mono-bibliothèque qui met en œuvre le modèle parallèle à gros grain en utilisant l'API ProActive, a été testé sur une grappe de huit nœuds biprocesseurs AMD Athlon MP 1500+ (1333 Mhz). Le système d'exploitation est Linux 2.4.2. Chaque nœud contient 1 GO de mémoire DDR-SDRAM qui offre un débit de 2.1 Gb/s avec une latence de 6 ns. Le réseau utilisé est de Ethernet 10/100 Mb.

L'algorithme de test retenue est la multiplication de matrice par circulation de colonnes. Il consiste à repartir les lignes de la première matrice sur les différents processus. La deuxième matrice est initialement repartis par bloc de colonne. L'algorithme comprend p (le nombre de processus mis en jeu) super-étapes. chaque super-étape calcule le résultat partiel des lignes de la matrice résultat, envoi les blocs de colonne de deuxième matrice à son successeur et récupère d'autres du prédécesseur.

Le nombre de processus mis en jeu varie entre 1 et 14. Pour chaque nombre, on fait varier la taille de deux matrices carrés en entrée de 390 jusqu'à 850 avec pas de 20. Sur chaque nœud il aura au plus deux processus. Chaque exécution est répétée 10 fois. Le résultat pris en compte est le temps d'exécution de l'algorithme. Afin de prendre en compte les caractéristiques de l'architecture, on a pris comme résultat de chaque expérience le nombre de cycles processeur par un item. Ce rapport est obtenu en multipliant la durée d'exécution par le nombre d'items total et la fréquence du processeurs.

8.6.2 Interprétation des résultats

Les tests sont composés de deux parties. La première consiste, pour un nombre donné de processus PRO dans un nœud, à les grouper dans une seule JVM. C'est-à-dire, s'il y a trois processus en total, il y aura dans chacun des deux nœud une seule JVM, le premier contient deux processus et l'autre un seul.

La deuxième dégroupe au maximum les processus sur les JVMs. C'est-à-dire, s'il y a trois processus en total, il aura dans le premier nœud deux JVM et dans le deuxième une seule JVM. Chaque processus PRO s'exécute sur une JVM à part.

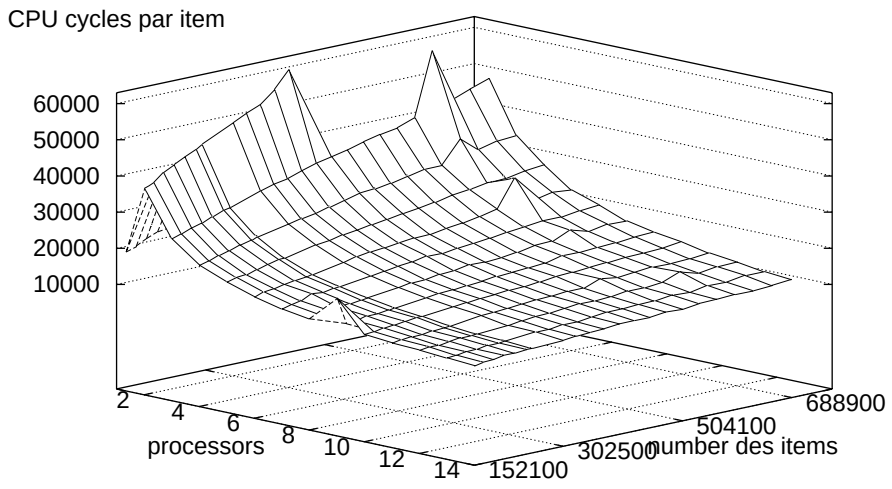
pour les figure 8.1 et 8.2, les deux axes de données sont le nombre de processus et le nombre d'items. L'axe de résultat est le nombre de cycle par item.

La figure 8.1 correspond à un groupement des processus. Elle montre que la version séquentielle est plus rapide que celle avec deux, trois et quatre processus. A partir de cinq processus, on commence à gagner en performance. Entre six et onze processus

et indépendamment du nombre d'items, l'accélération augmente progressivement. Cela montre le caractère gros grain de l'algorithme.

On remarque aussi parfois des pics dans la courbe qui montre une instabilité. Cette dernière peut être dû à l'algorithme, à la concurrence aléatoire des processus dans uns seule JVM ou à l'architecture. Dans le cas de dégroupement de processus, ces genres de pics n'existe pas. Donc, la raison de ces pics est la "mauvaise" ordonnancement des processus d'un même JVM.

Fig. 8.1 - Matrice - Groupement des processus



La figure 8.2 correspond à un dégroupement des processus. Elle montre que la version sequentielle est cette fois ci moins rapide. Entre un et cinq processus et indépendamment du nombre d'items, le nombre de cycle par item décroît rapidement, l'accélération augmente plus que celle de l'approche de groupement de processus.

Pour les deux figures 8.1 et 8.2, si le nombre de processus est supérieure à cinq dans le cas de groupement de processus et supérieure à onze dans le cas de groupement de processus et si on augmente le nombre d'items, le temps de traitement d'un item ne change plus. Le bout de la courbe devient plat. Cela signifie que le coût de calcul devient négligeable devant le coût de communication. L'accélération prend une valeur constante correspondante à la communication. Cela est plus claire si nous fixons le nombre de processus. En effet, pour un nombre fixe de processeur, si on augmente le nombre d'items l'accélération de change plus. La communication est prépondérante.

On remarque que le dégroupement des processus est nettement plus efficace et stable que la version avec groupement des processus dans la même JVM. La gestion des threads Java au sein d'une JVM n'est pas performant, notamment dans le cas des algorithmes à gros grain.

Dans la figure 8.3, nous fixons deux valeurs distinctes de nombre d'items et fait varier le nombre de processus et ceci en groupant et dégroupant les processus sur les JVMs. En groupant les processus d'un nœud dans une seule JVM, on obtient une

accélération qu'a partir de quatre processus pour 530000 items et cinq processus pour 200000. En augmentant le nombre d'items, on accélère plus rapidement.

Fig. 8.2 - Matrice - Degroupement des processus

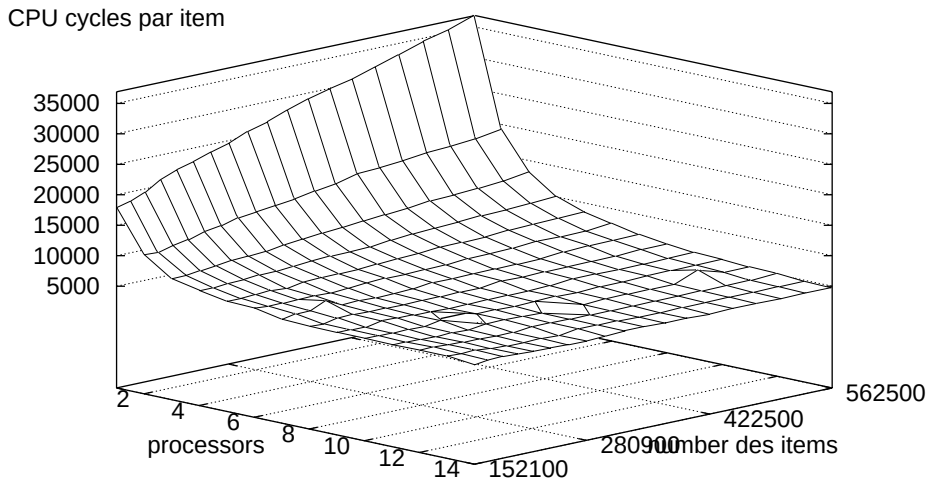
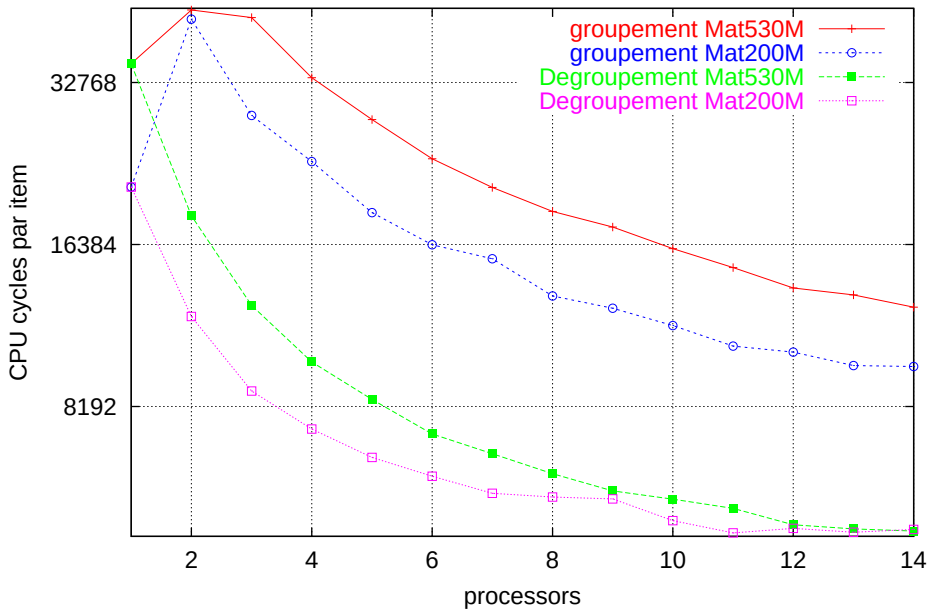


Fig. 8.3 - Matrice - Groupement et degroupement des processus



En dégroupant les processus, on obtient une accélération à partir de deux processus. En plus, à partir de onze processus, on satisfait une courbure plat qui signifie qu'on gagne plus en performance au delà de ce nombre de processus et pour ce nombre d'items. L'avantage de ce résultat est que le coût de communication se stabilise si on augmente le nombre de processus mis en jeu.

Enfin, on remarque que le deux courbes correspondants à un groupement de processus se joignent. Ceci reflete un passage à l'échelle. En effet, pour deux nombres d'items distincts, l'algorithme prend le même temps pour traiter un item.

Chapitre 9

Conclusions et perspectives

Les modèles parallèles à gros grain offre un cadre conceptuel permettant l'écriture d'algorithmes parallèles efficaces. Dans leur conception initiale, ces modèles ne sont pas en mesure de décrire les architectures hétérogènes. Pour couvrir un plus grand éventail de plates-formes, tout en gardant le même niveau d'efficacité, nous proposons un modèle permettant de couvrir ces architectures. Il est pris en compte pour concevoir des modèles de déploiement et de communication faisant le compromis entre exploitation des ressources architecturales et parallélisme à gros grain. Nos modèles proposés ont été implantés et testés en utilisant deux supports de calcul parallèle distribués PM^2 et ProActive.

Dans un premier temps nous avons étudié les différents modèles parallèle à gros grain. Dans un deuxième lieu, nous avons étudié les différents environnements permettant l'implantation des multithreads distribuées sur les architectures parallèles hétérogènes. Ceci nous a permis de proposer un modèle de déploiement des processus sur les différents processeurs. Notre modèle garantit une répartition équilibrée des ressources machine, maximise le parallélisme et minimise le temps de communication. Nous avons proposé des modèles de communication, de synchronisation globale et de synchronisation de communication. Ces derniers font le compromis entre les caractéristiques des différents modèles à gros grain, notre modèle d'architecture hétérogène et celui de déploiement.

En se basant sur nos propositions pour exploiter les architectures parallèles, nous avons conçu une interface utilisant l'environnement PM^2. Cet interface n'a pas donné des bons résultats lors de la phase de test. Cela est dû à l'incompatibilité de PM^2 avec les modèles parallèles à gros grain. En effet, PM^2 est plutôt adaptés aux algorithmes parallèles à grain fin. Cependant, ce support offre un modèle de programmation qui peut être choisi pour les travaux futurs.

Une autre interface utilisant l'environnement de calcul distribué ProActive a été conçue, implantée et testée sur une grappe de biprocesseurs. Les résultats ont validé l'utilisation de ce support pour l'algorithmique à gros grain. Mais, il s'avère coûteux en ressources mémoire et durée d'exécution.

Comme perspectives à cours terme, nous proposons de tester l'interface conçue avec ProActive avec d'autres algorithmes de différentes complexités et avec des architectures

parallèles variées. Nous pouvons ensuite la comparer avec la bibliothèque SSCRAP.

Notre modèle d'architecture couvre pour l'instant plusieurs machines du marché. Cependant, nous proposons comme perspectives à moyen terme, de compléter les hypothèses architecturales pour élargir son champ de recouvrement. Afin qu'il soit plus réaliste, nous essayons de prendre en compte le maximum de paramètres architecturaux.

Notre approche pour le déploiement, la communication et la synchronisation des processus peut être amélioré. Ceci en considérant les résultats expérimentaux obtenus afin qu'elle suit les changements du modèle d'architecture.

Nous pouvons envisager comme perspectives à long terme, le développement d'un support de calcul propriétaire exploitant directement les possibilités des réseaux. Ce support prendra en considération les spécifités des modèles parallèles à gros grain, le modèle d'architecture et les résultats expérimentaux. On peut s'inspirer des modèles de programmation des supports PM^2 et ProActive, et plus precisement de leurs approches pour assurer l'échange de données dans un milieu hétérogène ainsi que leurs méthodes pour la répartition et régulation de charge.

Bibliographie

[1] Mohamed Essaidi. Mémoire partagée pour le parallélisme à gros grain. Master's thesis, Université Henri Poincaré, Nancy I, 2000.

[2] Mohamed Essaidi, Isabelle Guérin Lassous, and Jens Gustedt. Sscrap : environnement de développement pour les modèles parallèles à gros grain. In *RenPar'14*, Hammamet, Tunisie, 2002.

[3] R. Namyst and J.-F. Méhaut. PM^2 : Parallel multithreaded machine. A computing environment for distributed architectures. In E. H. D'Hollander, G. R. Joubert, F. J. Peters, and D. Trystram, editors, *Parallel Computing : State-of-the-Art and Perspectives, Proceedings of the Conference ParCo'95, 19-22 September 1995, Ghent, Belgium*, volume 11 of *Advances in Parallel Computing*, pages 279–285, Amsterdam, February 1996. Elsevier, North-Holland.

[4] ProActive. INRIA, 1999. http ://www-sop.inria.fr/oasis/ProActive.

[5] Edson Caceres, Frank K. H. A. Dehne, Afonso G. Ferreira, Paola Flocchini, Ingo Rieping, Alessandro Roncato, Nicola Santoro, and Siang Song. Efficient parallel graph algorithms for coarse grained multicomputers and BSP. In Pierpaolo Degano, Robert Gorrieri, and Alberto Marchetti-Spaccamela, editors, *Automata, Languages and Programming, 24th International Colloquium*, volume 1256 of *Lecture Notes in Computer Science*, pages 390–400, Bologna, Italy, 7–11 July 1997. Springer-Verlag.

[6] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. Technical Report RR-1819, Inria, Institut National de Recherche en Informatique et en Automatique, December 1992.

[7] F. Dehne, C. Kenyon and A. Fabri. Scalable and architecture independent parallel geometric algorithms with high probability optimal time. In *Proc. IEEE Symposium on Parallel and Distributed Processing*, pages 586–593. IEEE, 1994.

[8] Gustedt Jens, Telle Jan Arne, Gebremedhin Assefaw Hadish, and Guérin Lassous Isabelle. Pro : a model for parallel resource-optimal computation. Technical Report RR-4319, INRIA- Lorraine, November 2001.

[9] Vincent Danjean, Raymond Namyst, and Robert D. Russell. Integrating kernel activations in a multithreaded runtime system on top of L INUX. *Lecture Notes in Computer Science*, 1800 :1160–? ?, 2000.

[10] Luc Bouge, Jean-Francois Mehaut, and Raymond Namyst. MADELEINE : An efficient and portable communication interface for RPC-based multithreaded environments. Technical Report RR-3459, Inria, Institut National de Recherche en Informatique et en Automatique, 1998.

[11] Ken Arnold, Ann Wollrath, Bryan O'Sullivan, Robert Scheifler, and Jim Waldo. *The Jini specification.* Addison-Wesley, Reading, MA, USA, 1999.

[12] I. Foster and C. Kesselman. Globus : A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2) :115–128, Summer 1997.

[13] Richard Miller. A library for bulk-synchronous parallel programming. In *British Computer Society Parallel Processing Specialist Group workshop on General Purpose Parallel Computing*, December 1993.

[14] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob Bisseling. BSPlib : The BSP programming library. Technical Report May, Oxford University Computing Laboratory, 1997.

[15] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8) :103, August 1990.

[16] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP : towards a realistic model of parallel computation. *ACM SIGPLAN Notices*, 28(7) :1–12, July 1993.

[17] Guérin Lassous I. *Algorithmes parallèles de traitement de graphes : une approche basée sur l'analyse expérimentale.* PhD thesis, Universitée Paris 7, 1999.

[18] Frank Dehne, Xiaotie Deng, Patrick Dymond, Andreas Fabri, and Ashfaq A. Khokhar. A randomized parallel 3D convex hull algorithm for coarse grained multicomputers. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 27–33, Santa Barbara, California, July 17–19, 1995. ACM SIGACT/SIGARCH and EATCS.