View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by INRIA a CCSD electronic archive server

Composabilité de codes parallèles sur architectures hétérogènes

Andra-Ecaterina Hugo

▶ To cite this version:

Andra-Ecaterina Hugo. Composabilité de codes parallèles sur architectures hétérogènes. 2011. inria-00619654

HAL Id: inria-00619654

https://hal.inria.fr/inria-00619654

Preprint submitted on 6 Sep 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Mémoire de MASTER RECHERCHE

présenté par

Andra-Ecaterina Hugo

Composabilité de codes parallèles sur architectures hétérogènes

Encadrement : Abdou Guermouche, Pierre-André Wacrenier, Raymond Namyst

Février – Juin 2011

Laboratoire Bordelais de Recherche en Informatique (LaBRI)
INRIA Bordeaux
Université Bordeaux 1

Table des matières

1 Introduction **5**
1.1 Des calculateurs toujours plus hétérogènes... 5
1.2 ...exécutant des programmes de plus en plus hybrides... 6
1.3 ... mais peu réutilisables ! 6
1.3.1 En face d'un exemple concret : une application de simulation sismique 6
1.3.2 Objectifs 7

2 À propos de la programmation hybride **8**
2.1 Environnement de programmation pour les architectures parallèles 8
2.1.1 Outils pour programmer les multicœurs 8
2.1.2 Outils pour programmer les GPUs 9
2.2 De l'optimisation des programmes hybrides 9
2.2.1 Optimisations du comportement des applications multicœurs 10
2.2.2 Optimisation des codes pour accélérateurs 10
2.2.3 Optimisation des codes hybrides 10
2.3 Relever le défi de la composition de codes parallèles 12
2.3.1 Composition au niveau de la machine - les machines virtuelles du système 13
2.3.2 Composition au niveau de l'application - les machines virtuelles du processus 13
2.3.3 Composition au niveau de la source de parallélisme 13
2.4 Discussion : hétérogénéité en Lithe ou composition en StarPU 15

3 Contribution : Une plate-forme pour composer du code parallèle **16**
3.1 Objectifs : composer pour mieux exploiter ! 16
3.1.1 Encapsuler les ordonnanceurs pour les composer 16
3.1.2 Contrôler le partage par le recouvrement 17
3.1.3 Donner le pouvoir à l'usager 18
3.2 Des contextes pour répondre à la demande 18
3.2.1 Respecter la qualité de service 18
3.2.2 Adapter les bibliothèques spécialisées 19
3.3 Limites de l'approche 19
3.4 Discussion : vers un parallélisme adaptatif 19

4 Éléments d'implémentation **21**
4.1 La structure "sched_ctx" 21
4.2 Création des contextes 21
4.3 Suppression des contextes 22
4.4 Soumission des tâches au contexte 22
4.4.1 Les stratégies d'ordonnancement dans la présence des contextes 22
4.5 Redimensionnement des contextes 22

5 Évaluation **24**
5.1 Environnement d'évaluation . . . 24
5.2 Cas du Cholesky . . . 24
5.2.1 Sur la factorisation Cholesky... . . . 24
5.2.2 Pourquoi Cholesky... . . . 25
5.3 Stratégie d'évaluation . . . 25
5.3.1 Points visés . . . 25
5.3.2 Métrique . . . 25
5.4 Performances sur les architectures homogènes . . . 26
5.4.1 Sur-coût de la plate-forme de composabilité des codes parallèles . . . 26
5.4.2 Stabilité des résultats . . . 27
5.4.3 Scalabilité sur une très grande machine . . . 28
5.5 Comportement sur une machine hétérogènes . . . 28
5.5.1 Sur-coût de la plate-forme de composabilité des codes parallèles . . . 29
5.5.2 Répartition isolée des ressources sur les contextes . . . 30
5.5.3 Recouvrement des contextes . . . 31

6 Conclusions **32**

Chapitre 1

Introduction

De plus en plus de scientifiques (biologistes, climatologues, sismologues, etc.) s'appuient de nos jours sur la simulation pour affiner hypothèses et modélisations. Ainsi, les besoins en terme de capacité de calcul intensif augmentent tous les jours, nécessitant des machines de calcul toujours plus puissantes. C'est ainsi que de pans entiers de la recherche scientifique dépend maintenant de la performance des centres de calcul.

1.1 Des calculateurs toujours plus hétérogènes...

Pour répondre à la quête de la performance les architectes ont, par le passé, utilisé des techniques de miniaturisation des composants pour augmenter la fréquence des processeurs et la taille des mémoires cache afin d'accélérer le traitement d'un flux d'instructions. Depuis 2004, la dissipation de la chaleur engendrée par les composants haute fréquence est telle qu'elle empêchent les architectes d'augmenter plus encore la fréquence. Pour continuer à proposer des machines toujours plus puissantes, les architectes ont exploité les progrès en matière de finesse de gravure des composants pour répliquer les unités de traitement sur une même puce, donnant naissance aux machines mutlicœurs. Ainsi, un processeur contemporain comporte plusieurs cœurs capables d'exécuter des traitements en parallèle tout en partageant différents niveaux de mémoire cache. De plus, plusieurs processeurs peuvent être connectés entre eux à l'aide de bus spécialisés, et, pour augmenter le débit mémoire, celle-ci est non plus centralisée mais distribuée processeurs par processeurs entraînant des temps d'accès variables selon qu'un cœur accède au banc mémoire local à son processeur ou à un autre banc mémoire.

Depuis quelques années, nous constatons une autre évolution majeure, en particulier autour du calcul intensif : l'avènement des architectures hétérogènes via l'exploitation de la puissance de calcul des coprocesseurs, plus précisément des processeurs graphiques. La simplicité des pipelines de calcul des cartes graphiques permet de les répliquer en grand nombre sur une même carte et d'obtenir des performances brutes plus importantes que celles des processeurs généralistes récents. Cependant les accélérateurs ne sont pas capables d'exécuter efficacement tous les programmes : ils sont spécialisés dans l'exécution d'un grand nombre de tâches au comportement similaires mais agissant sur des données différentes. C'est pourquoi processeurs multicœurs généralistes et cartes graphiques sont complémentaires... Du reste cette approche tend à se généraliser et va être sans doute favorisée par l'intégration des accélérateurs graphiques dans les processeurs standard (technologies Intel Sandy Bridge et AMD Fusion, en rayon depuis 3 mois dans toutes les grandes surfaces).

1.2 ...exécutant des programmes de plus en plus hybrides...

Pour répondre à ces révolutions matérielles de nombreux environnements de programmation existants ont été révisés pour mieux exploiter les machines multicœurs (OpenMP, TBB, etc.) et de nouveaux environnements ont vu le jour pour gérer les plates-formes contenant des accélérateurs (CUDA, OpenCL, etc.). Mais en raison de manque de consensus autour de la définition d'un standard pour la programmation de ce genre de plates-formes, un nombre important de développeurs d'applications haute-performance combinent manuellement différents environnements de programmation spécifiques (ex : MPI+OpenMP+CUDA). Ceci demande au programmeur un degré important d'expertise sur ces technologies complètement différentes, ce qui l'empêche à se concentrer sur les problématiques algorithmiques qui l'intéressent.

1.3 ... mais peu réutilisables !

Si la communauté maîtrise désormais un peu mieux la production de code adapté à l'exploitation des machines hétérogènes multicœurs, un défi majeur demeure : il est toujours difficile de gérer efficacement plusieurs sources de parallélisme au sein d'une même application. En effet les différents bibliothèques de calcul et environnements de programmation parallèles actuels ne coopèrent pas pour se partager les unités de calcul, pire, ils sont en compétition pour s'exécuter et peuvent finalement se gêner au détriment des performances du programme. Cette absence de coopération rend difficile la mise au point des applications parallèles du point de vue des performances et nuit à la réutilisation de multiples codes parallèles dans un programme déjà parallèle.

1.3.1 En face d'un exemple concret : une application de simulation sismique

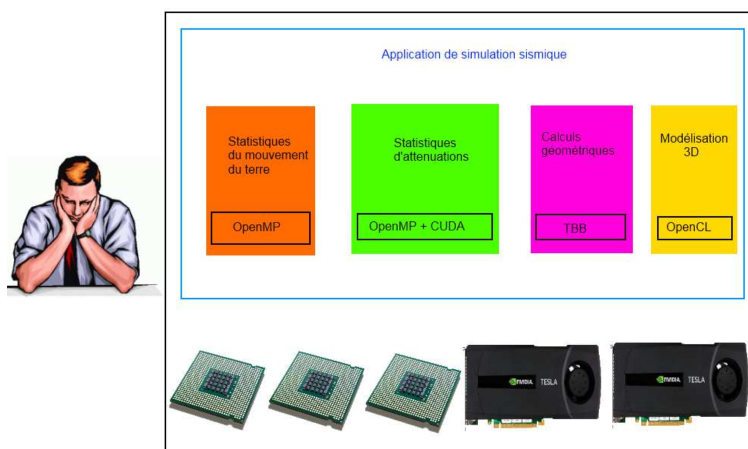

FIGURE 1.1 – Simulation sismique

Dans la figure 1.1 on décrit la situation d'un physicien qui construit une application de simulation sismique. Cette application est censée faire des calculs sur le mouvement de la terre, sur les atténuations, des calculs géométriques et de modélisation 3D de la propagation des ondes. Pour chaque type de calcul on dispose de bibliothèques parallèles qui fournissent des outils de programmation pour faciliter le travail du physicien, et qui sont optimisés pour tirer parti au maximum des machines de calcul contemporaines.

Pour tirer profit des performances des machines multi-cœurs, les bibliothèques parallèles fixent en général un thread par cœur pour contourner l'ordonnanceur du système d'exploitation. La distribution du travail est alors à la charge de la bibliothèque, ce qui lui permet de mettre en œuvre des politiques d'ordonnancements précises pour par exemple contrôler le mieux possible l'utilisation du cache. Lorsque une application utilise plusieurs bibliothèques parallèles simultanément, les threads des différentes bibliothèques vont s'ignorer, utiliser les mêmes ressources et, finalement, se gêner. Ainsi, lorsque le physicien lancera son application utilisant des bibliothèques super-optimisées sur une machine avec beaucoup de ressources de calcul il s'attendra à obtenir très rapidement ses résultats, mais malheureusement il ne peut pas tirer partie de sa machine pour améliorer son calcul parce que les bibliothèque parallèles qu'il veut utiliser ne savent pas coexister dans une même application. En effet, sur un même cœur, le comportement de deux threads apparaîtra imprévisible aux yeux du programmeur : c'est le système d'exploitation qui intervient pour les ordonnancer. Aussi le plus simple du point de vue du programmeur est actuellement de dédier une partie de la machine à chaque bibliothèque, au risque d'une sous exploitation du matériel. Par conséquent, les applications résultant de la composition des bibliothèques parallèles manifestent le plus souvent de mauvaises performances.

Dans un domaine totalement diffèrent d'expertise, on se confronte aussi avec le problème de la composabilité de codes parallèles, mais cette fois-ci au niveau des applications. Les domaines comme la météo veulent composer de très grandes applications parallèles existantes (par exemple des application qui simulent le comportement de l'atmosphère avec celles qui simulent le comportement de l'océan). Ce type de problème est géré à travers le couplage de code, un procédé qui s'occupe de limiter l'exécution de chaque application sur certains noeuds et de les faire communiquer à travers un couche spécialisé. Par contre, cette situation est adapté aux clusters, au niveau d'un même noeud généralement on essaye de tire profit de la mémoire partagée et éviter le coût de la communication. Alors, même si le couplage de code présente des solutions au niveau de la composition des applications, pour les bibliothèques on cherche encore comment gérer la coexistence des codes parallèles différents.

1.3.2 Objectifs

Mon objectif à long terme est de résoudre ce problème de composition de codes parallèles en proposant de nouveaux mécanismes au niveau des supports exécutifs ainsi que de nouvelles stratégies d'allocation des ressource dans les bibliothèques parallèles afin de permettre à ces dernières d'être réellement réutilisables, composables, dans le cadre de l'exploitation des architectures de calcul hétérogènes.

Dans le cadre de ce mémoire, je propose ainsi de construire une plate-forme au niveau exécutif qui optimise l'exécution simultanée de plusieurs noyaux de calcul parallèles. Plus précisément, mon travail concerne l'extension dans ce sens du support exécutif StarPU, logiciel développé par l'équipe projet Runtime. Pour atteindre cet objectif, j'ai défini la notion de contexte d'ordonnancement qui abstrait le fait de partager des unités de calcul entre différentes bibliothèques parallèle tout en leur donnant du contrôle et des garanties sur l'ordonnancement de leurs tâches. Dans ce cadre une bibliothèque n'a plus à fixer ses threads, mais s'exécute dans un contexte d'ordonnancement géré par le support d'exécution chargé de la répartition des ressources de calcul. La support d'exécution permet en effet de partager ou non une ressource et de la redistribuer dynamiquement afin d'adapter de façon naturelle le nombre et le type de ressources à la demande.

Chapitre 2

À propos de la programmation hybride

Aujourd'hui les constructeurs proposent des machines dont la puissance de calcul brute se compte en Téra flops (10^{15} instructions par seconde), ce sont des machines composées d'une à quelques dizaines de coeurs organisés hiérarchiquement et de quelques accélérateurs matériels. Cependant tirer parti de ces architectures demande un effort de programmation conséquent. Dans ce chapitre nous passons en revue les techniques de programmation parallèles adaptées à ces machines hétérogènes et présenterons le problème de la composition de codes parallèles.

2.1 Environnement de programmation pour les architectures parallèles

Manipuler efficacement le parallélisme pour ces architectures reste encore un véritable problème mais souvent on dispose des outils qui peuvent faciliter la programmation d'un noyau de calcul.

2.1.1 Outils pour programmer les multicœurs

Même si les machines multicœurs sont désormais majoritaires on ne peut pas dire qu'on détient la solution providentielle au problème de leur exploitation. On manipule le parallélisme explicitement en utilisant soit directement des threads soit des extensions de langage ou des bibliothèques offrant un support pour le parallélisme comme OpenMP, TBB, Cilk et MPI.

La bibliothèque Posix Thread

PThread est une interface de programmation système, de bas niveau, qui permet de lancer, placer et de synchroniser directement des flots d'exécution communiquant par mémoire partagée. Cette interface demande des connaissances conséquentes de l'architecture de la machine. Comme pouvoir rime avec devoir, le programmeur généralement doit gérer les problèmes de concurrences liés au parallélisme (priorité, protection des variables partagées, deadlock).

Les environnements de programmation à mémoire commune

OpenMP[14] est une extension des langages C, C++ et Fortran qui permet d'écrire rapidement un programme parallèle : à l'aide d'annotations le programmeur OpenMP indique au compilateur quelles sont les portions de code à paralléliser et comment. Outre sa simplicité le point fort d'OpenMP est de faciliter la parallélisation progressive du code et de pouvoir revenir à

tout moment à une exécution séquentielle. Son point faible est de ne pas proposer d'abstractions pour aider le programmeur à bien structurer les données privées ou partagées par les threads.

Cilk [3] est une extension du langage C qui s'appuie sur une gestion de tâches parallèles extrêmement efficace. Le modèle d'exécution d'un programme Cilk consiste à avoir quelques processus légers qui détiennent chacun des files de tâches à exécuter et dans l'absence des tâches ils peuvent en voler depuis les files des autres. Cilk permet une parallélisation très efficace mais il s'applique qu'à des algorithmes récursives de type "Diviser pour régner".

Intel TBB [18] est une bibliothèque développée par Intel qui, en quelque sorte, unifie OpenMP et Cilk au sein de C++. Cette bibliothèque propose de puissantes abstractions (graphe de tâches par exemple) mais souffre d'une certaine lourdeur dans son utilisation.

La bibliothèque de communication MPI

L'interface MPI est basée sur le paradigme de la communication par message. Si cette technologie logicielle demande un effort important de programmation, elle est actuellement l'outil majeur de la programmation parallèle scientifique y compris sur les machines à mémoire commune. Outre l'effort de programmation demandé au programmeur, son point faible est la consommation mémoire supplémentaire qu'elle demande.

2.1.2 Outils pour programmer les GPUs

Les précurseurs du calcul scientifiques sur GPU détournaient les API graphique telles que DirectX et OpenGL et traitait leur problème sous la forme d'opérations sur des textures. C'était un procédé fastidieux et limitant. Depuis l'apparition des langages dédiés au calcul sur GPU a permis de rendre plus accessible la programmation des cartes graphiques. En particulier on peut citer les langages Cuda et OpenCL. Cuda (NVidia)[20] est un langage très proche de C qui fournit une abstraction du matériel (d'assez bas niveau), OpenCL est un standard ouvert, inspiré de Cuda, pour la programmation des GPUs et des CPUs. De fait OpenCL a des objectifs plus larges et ambitieux que Cuda qui lui évolue plus vite et permet de mieux exploiter les GPU.

La programmation des GPU étant assez différente de celle des CPU et donc des habitudes de programmation de la communauté, des environnements de programmation ont vu le jour pour réduire cet écart. Ces environnements sont basés sur le même principe que OpenMP, à l'aide de directives ajoutées au programme on indique quelles parties du code doivent s'exécuter sur le GPU. Dans l'absence de directives le code s'exécutera sur le CPU sans aucune modification.

HMPP[11] fait partie d'une de ces environnements de programmation qui propose une solution capable de gérer l'exécution concurrente de plusieurs codelets (fonctions adaptés à l'accélérateur visé). StarSs[8] présent une solution similaire, qui permet la gestion simultanée de plusieurs cartes graphique à travers un système de cohérence de données entre leurs mémoires.

Ces modèle de programmation apporte une facilité significative pour la portabilité du code et ils évitent au programmeur de s'occuper des transferts mémoire et synchronisation entre les CPUs et les accélérateurs. Mais comme dans le cas de OpenMP, tous ces avantages viennent au détriment des performances.

2.2 De l'optimisation des programmes hybrides

La révolution des architectures parallèles reporte désormais sur les programmeurs les contraintes rencontrées par les constructeurs. Ces nouvelles contraintes mettent en péril la qualité des bibliothèques de calculs jusqu'alors très performantes. Si l'optimisation des programmes destinés aux processeurs multi-cœurs commence à être maîtrisée, elle reste quand même complexe

à mettre en oeuvre. Quant à l'optimisation des codes GPU, outre les problèmes algorithmiques posés, les environnements de programmation des GPU évoluent sensiblement tous les six mois, offrant de nouvelles techniques à tester et à mettre en oeuvre.

2.2.1 Optimisations du comportement des applications multicœurs

Pour programmer les machines multicœurs on utilise des threads qui travaillent de manière concurrente sur différents processeurs et qui permettent l'exécution de plusieurs portions de code en parallèle. Ces threads peuvent être créés explicitement par le programmeur à l'aide de bibliothèques de threads (ex : POSIX) ou générés automatiquement par des langages de programmation parallèle (ex : OpenMP, TBB). La première approche est plus puissante mais elle demande un niveau d'expertise conséquent. La seconde est plus simple à utiliser, mais elle n'apporte pas les outils pour adapter le parallélisme du programme aux architectures hiérarchiques, telles les NUMA[9]. Pour certain calculs, des threads travaillent par groupe sur les mêmes données, pour optimiser l'exécution de ce type de calcul il est nécessaire de faire en sorte que les threads qui travaillent ensemble soient exécutés sur des cœurs voisins sur la machine. Pour d'autres calculs certains threads auront besoin de beaucoup de bande passante mémoire ce qui requiert une dispersion de ces threads sur la machine. Par conséquent, chaque calcul a des besoins différents et trouver une solution qui peut être appliquée sur tous les programmes est très difficile voire même impossible.

De plus, avec l'augmentation du nombres de cœurs et la hiérarchisation de la mémoire il devient de plus en plus difficile de maintenir une mémoire cohérente sur l'ensemble des processeurs. Dans l'absence d'une mémoire globale partagée, il revient au programmeur d'assurer une cohérence du cache passant à l'échelle en évitant les problèmes de false sharing, par exemple, ce qui n'est pas toujours évident lorsqu'on travaille à l'aide de threads (par opposition à MPI).

2.2.2 Optimisation des codes pour accélérateurs

Tout d'abord il s'agit de faire rentrer le problème dans la carte en cherchant à optimiser le rendement des pipeline de calculs : pour cela le programmeur doit trouver le meilleur équilibre entre le nombre de threads et consommation par thread mémoire locale (qui joue le rôle des registres et des caches). En effet il faut chercher à ne pas être pénalisé par la latence de la mémoire globale (600 cycles) en utilisant un grand nombre de threads (pour recouvrir la latence) et la mémoire dite partagée (d'accès rapide, comme des registres, mais de taille limitée). Pour cela il s'agit de réaliser des accès coalescés à la mémoire et d'utiliser au mieux la mémoire partagée (shared memory ou registres) en optimisant le volume de mémoire partagée occupé par thread (plus ce volume est petit plus on peut exécuter efficacement de threads en parallèle).

Une fois que l'on dispose des noyaux de calcul performants il s'agit de les alimenter en données puis de les lancer. Ici l'absence de mémoire globale oblige le programmeur à gérer la cohérence mémoire entre la mémoire commune des CPU et les mémoires embarquées des multiples accélérateurs. Cette gestion mémoire est d'autant plus importante que la mémoire embarquée sur un GPU est de taille réduite (comparée à la mémoire commune) et, surtout, que les GPU ont des capacités de calcul bien supérieures à leur débit mémoire.

2.2.3 Optimisation des codes hybrides

Les machines équipée d'un grand nombre de coeurs et de quelques accélérateurs laissent espérer aux programmeurs l'obtention de performances importantes, sous réserve que le problème s'y prête. Une difficulté supplémentaire apparaît lorsqu'on cherche à optimiser un code sur machine hétérogène : celle du placement d'un calcul. D'un point de vue statique, il est assez

facile de décider s'il vaut mieux placer un calcul sur un accélérateur ou sur un ensemble de coeurs, il suffit de faire quelques expériences pour déterminer la bonne décision à prendre et d'en tirer des règles simples : "il est avantageux d'exécuter le calcul A sur la carte graphique à partir de taille T_A". Le problème est autrement plus difficile d'un point de vue dynamique. En effet imaginons que nous ayons à exécuter en parallèle deux tâches A de même taille, la règle que nous avions déterminée statiquement n'est sans doute plus vérifiée car pour terminer au plus tôt la meilleure solution pourrait être d'exécuter une tâche sur un GPU et l'autre sur un CPU. C'est à ce problème là que Cédric Augonnet s'est attaqué dans sa thèse en proposant le logiciel StarPU.

StarPU

Pour que le programmeur puisse se concentrer sur la construction des noyaux de calcul, plutôt que sur les détails techniques spécifiques à chaque plate-forme, StarPU propose une abstraction des ressources capable d'ordonnancer et de répartir les tâches de calcul en prenant en compte les avantages et les limitations de ces ressources. StarPU[4] représente un outil qui facilite la programmation et l'exécution des noyaux des calcul en automatisant les interactions entre les tâches ainsi que le mouvement de données sous-jacents.

StarPU est structuré en plusieurs niveaux : à travers l'interface de programmation on a accès aux outils pour soumettre les noyaux de calcul à StarPU et ensuite ce support exécutif va gérer les données et les codelets[5] fournis par l'utilisateur. L'ordonnanceur est celui qui prend les décisions par rapport à l'exécution des tâches et les pilotes s'occupent de récupérer les tâches attribuées pour les exécuter.

FIGURE 2.1 – Pile Logicielle de StarPU

En respectant cette structure[4], on assure la portabilité des stratégies, qui ne connaissent pas la nature des ressources qui vont exécuter les tâches, mais seulement leurs listes de tâches en attente. De l'autre coté les pilotes ne sont pas censés connaître par quel algorithme les tâches arrivent dans leurs files.

Gestion des données Pour automatiser le mouvement de données entre la mémoire des différentes ressources, StarPU implémente une DSM (Distributed Shared Memory)[6]. C'est-à-dire, pour éviter les transferts inutiles de mémoire, il garde des données là où elles étaient

utilisés la dernière fois, même si celle-ci ont été modifiées, en évitant de mettre à jour la mémoire principale de manière anticipée. En plus, StarPU garantit la cohérence des données à travers l'implémentation du protocole MSI, en permettant d'avoir plusieurs copies de la même donnée sur plusieurs PUs tant qu'elle n'est pas modifiée.

Ce système facilite la programmation des noyaux de calcul, parce que l'application doit juste enregistrer les données en fournissant leur adresse et leur taille dans la mémoire principale.

De plus, StarPU propose un mécanisme de filtres[6] qui permet de partitionner facilement les données. On peut adapter la tailles des blocs au besoin de l'algorithme du noyau de calcul. En appliquant les filtres dynamiquement on peut permettre à la DSM de transférer exactement le morceau de donnée nécessaire.

Algorithmes d'ordonnancement Un seul algorithme d'ordonnancement universel, qui apporterait les meilleures performances pour tout type d'application ou matériel, n'existe pas. Aussi, StarPU propose un système qui permet de choisir parmi plusieurs stratégies d'ordonnancement et aussi de construire ses propres stratégies.

Il y a plusieurs stratégies prédéfinies : glouton, vol de travail et HEFT (Heterogeneous Earliest Time Strategy)[7]. Cette dernière est basé sur des prédictions de performances, construites à travers le feedback de l'utilisateur, l'historique et les spécificités de chaque ressource. En effet, pour soumettre une tâche avec la stratégie "heft", on calcule pour chaque ressource la taille du travail en attente et on prévoit le temps que celle-ci peut prendre pour y être exécuté (y compris le temps de transfert de données nécessaires). Avec cette information l'ordonnanceur peut affecter la tâche sur la ressource où elle se terminera au plus tôt.

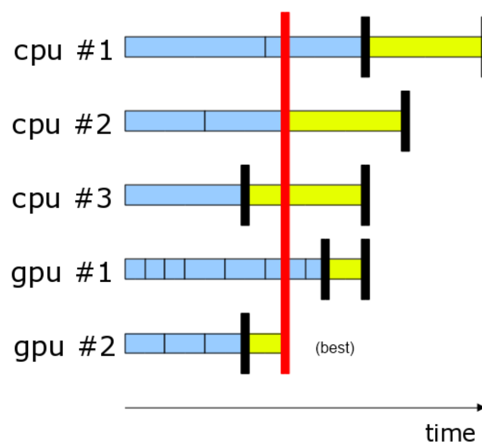

FIGURE 2.2 – La stratégie Heft

2.3 Relever le défi de la composition de codes parallèles

Aujourd'hui l'objectif est de mettre à profit tous ces outils pour arriver à exploiter au maximum les ressources de calcul dont on dispose.

Les premières démarches dans ce sens ont été fait lors de la construction du premier système à mémoire virtuelle par IBM, démontrant l'utilité d'avoir plusieurs programmes qui tournent simultanément tout en leur laissant croire que chacun dispose de l'intégralité de la machine physique. Les machines virtuelles poursuivent le même objectif : tirer partie simultanément

d'une même machine physique, en fournissant des ressources virtualisées pouvant accueillir des systèmes d'exploitation différents.

2.3.1 Composition au niveau de la machine - les machines virtuelles du système

La virtualisation des ressources permet de faire tourner plusieurs systèmes d'exploitation, chacun ayant une vision locale réduite aux ressources sur lesquelles il s'exécute sans pour autant être conscient de l'existence d'autre ressources ou des autres instances de systèmes d'exploitation. Une couche logicielle permet aux machines virtuelles de partager des ressources physiques de la machine entre eux. Cette couche logicielle assure la coexistence de plusieurs systèmes d'exploitation de façon complètement isolée au prix d'une réduction de l'efficacité.

2.3.2 Composition au niveau de l'application - les machines virtuelles du processus

La virtualisation apporte encore un autre grand avantage : la portabilité. Le langage JAVA est né à partir de cette idée. Le but est d'avoir une machine virtuelle java (JVM) [12] pour chaque application qui abstrait les détails du matériel et du système d'exploitation et qui assure une indépendance par rapport à la plate-forme. Ainsi, lorsqu'on lance un programme java, la JVM est lancée avec le processus et à la fin elle est détruite. De nouveau il y a des avantages et des inconvénients, la portabilité est au prix d'un certain ralentissement du programme.

2.3.3 Composition au niveau de la source de parallélisme

L'apparition des puces de centaines de cœurs et l'augmentation des ressources de calcul en général rend difficile voire même impossible, l'exploitation d'une telle puissance de calcul par un seul noyau parallèle. Par conséquent, les applications seront obligées de tirer partie de multiples sources de parallélisme et les faire coexister. Cependant de nos jours les applications essayant d'appeler simultanément plusieurs bibliothèques parallèles obtiennent de mauvaises performances parce que chaque bibliothèque ignore l'utilisation des ressources des autres. Ainsi, les "threads" crées par des bibliothèques parallèles distinctes sont en compétition pour accéder aux mêmes ressources.

Dans l'absence d'un support exécutif qui gère la composition, le système d'exploitation n'est pas informé de l'existence de plusieurs bibliothèques parallèles. Le système doit ordonnancer de nombreux "threads" (dont le nombre excède le nombre de cœurs) à l'aveugle, ce qui peut entraîner une importante perte d'efficacité (due aux changements de contexte intempestifs, au pourrissement des caches, à des attentes actives, etc.).

Impact sur l'évolution des applications parallèles

Dans ce contexte, il y a très peu d'applications HPC qui utilisent simultanément plusieurs codes parallèles. Des bibliothèques parallèles développées par la même entreprise rencontrent des problèmes de composition. Par exemple, Intel préconise l'utilisation de la version séquentielle de sa bibliothèque MKL dans des programmes utilisant OpenMP. Cette situation est alarmante, parce que les principes de programmation tels que la modularité, la composition et la réutilisation ne peuvent actuellement pas être appliqués au monde du calcul parallèle. Ce problème deviendra de plus en plus critique dans le futur proche lorsque l'augmentation de nombres de cœurs va demander un grand degré de parallélisme.

Outils existants : Lithe

Lithe[16] propose une solution similaire aux machines virtuelles de niveau applicatif (comme la JVM) mais appliquée aux bibliothèques parallèles utilisées au sein d'une même application. Ainsi, on peut voir les bibliothèques parallèles comme des machines virtuelles complètement isolées qui communiquent entre eux à travers Lithe pour partager les ressources.

Avec Lithe les threads ne représentent plus l'abstraction virtuelle des CPUs et les ressources de calcul sont représentées par les "harts". Ainsi, une bibliothèque parallèle peut ordonnancer des tâches sur un nombre fixe de "harts" gérés par Lithe. A la place de voir le système comme un fournisseur de ressources illimités, Lithe offre un nombre de "harts" échangés de manière coopérative entre les bibliothèques composées hiérarchiquement (voire figure 2.3).

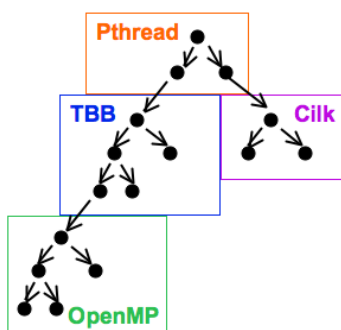

FIGURE 2.3 – L'ordonnancement hiérarchique avec Lithe

Chaque bibliothèque portée sur Lithe[15] génère des tâches qui représentent en fait des contexte (pile et registres) ordonnancées sur les "harts" et dont les changements de contexte se font au sein de la bibliothèque.(voire figure 2.4).

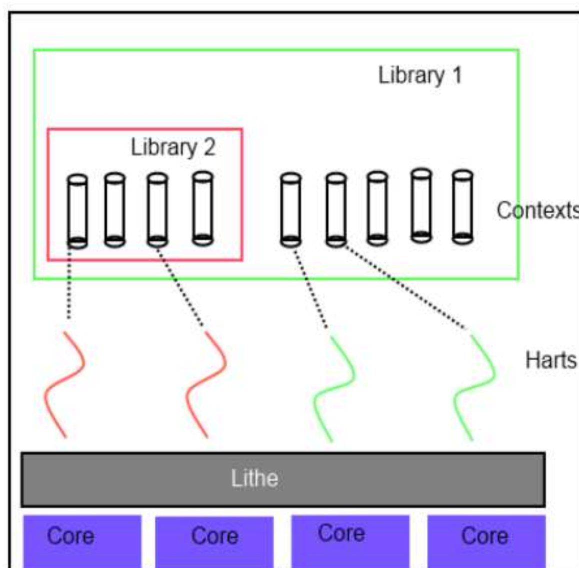

FIGURE 2.4 – Les contextes et les harts en Lithe

2.4 Discussion : hétérogénéité en Lithe ou composition en StarPU

Actuellement, Lithe représente le seul support exécutif capable de gérer la composition. A travers une virtualisation des CPUs, Lithe rend possible le partage efficace des ressources entre plusieurs bibliothèques parallèles. Par contre, Lithe limite les applications a s'exécuter sur CPUs et nécessite de coûteuses synchronisation entre les bibliothèques et ne permet pas redimensionner facilement les contextes en fonction des besoin de calculs et des ressources disponibles.

Plutôt que rajouter ces fonctionnalités en Lithe il est plus avantageux de profiter de la gestion performante de la hétérogénéité de StarPU et d'y introduire le concept de composition. StarPU propose déjà un environnement de programmation qui se charge d'exécuter les noyaux de calcul de manière transparente et qui permet la cohabitation des plusieurs types de plates-formes de calcul. Néanmoins, avec StarPU on dispose déjà de diverses stratégies d'ordonnancement ainsi que d'un système d'autotuning qui pourrait rendre plus facile le dimensionement des contextes.

Chapitre 3

Contribution : Une plate-forme pour composer du code parallèle

Notre travail est centré sur la construction d'une plate-forme d'expérimentation capable de gérer de manière efficace plusieurs sources de parallélisme au sein d'une même application. Contrairement à Lithe, qui propose une virtualisation du matériel pour permettre l'exécution de plusieurs codes parallèle, nous proposons de construire un API au centre de StarPU qui permet de cloisonner de manière flexible les ressources entre plusieurs bibliothèques parallèles. Le principe de cloisonnement est définit à travers la notion de contexte d'ordonnancement qui abstrait le fait d'isoler certains ressources de calcul pour les attribuer à un noyau parallèle. Ainsi, une bibliothèque parallèle qui s'exécute dans un contexte d'ordonnancement a du contrôle et des garanties sur l'ordonnancement de ses tâches sur les ressources de calcul que le support d'exécution lui a attribué.

3.1 Objectifs : composer pour mieux exploiter !

L'objectif principal de ce travail est centré sur la possibilité d'exploiter plusieurs contextes d'ordonnancement capable chacun de gérer l'exécution d'une bibliothèque parallèle sans pour autant perturber l'exécution des autres parties parallèles de l'application. Avec des stratégies d'ordonnancement différentes et des gestions indépendants du travail, les contextes coexistent sur la machine sans globalement se perturber les uns les autres (voir figure 3.1).

On se propose d'illustrer l'importance de la connaissance de l'utilisateur de l'application et de la plate-forme sur la gestion des contextes d'ordonnancements. En effet, bien qu'un des objectifs de StarPU est de masquer à l'utilisateur la complexité de plate-forme sous-jacente, le fait de pouvoir exprimer simplement à l'aide des contextes d'ordonnancement des contraintes liées à l'application peut grandement améliorer le comportement de cette dernière.

3.1.1 Encapsuler les ordonnanceurs pour les composer

StarPU fournit une interface d'ordonnancement que l'utilisateur peut exploiter pour concevoir simplement des stratégies d'ordonnancement grâce à une vision simplifiée de la plate-forme sous-jacente. L'utilisation des contextes décrits dans les sections précédentes implique que les ordonnanceurs spécifiques à chaque contexte ne doivent "voir" que le contexte en question. Pour autant, lorsqu'il est nécessaire de partager des ressources entre différents contextes (voir la section 3.1.2), il est nécessaire que les différents ordonnanceurs aient une vue globale de ce qui se passe au niveau des ressources partagées.

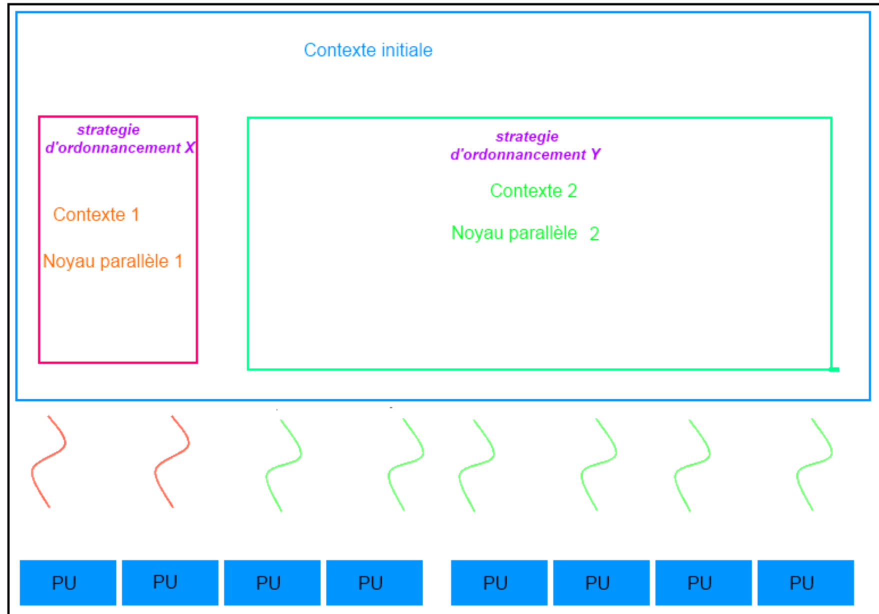

FIGURE 3.1 – Vision d'ensemble de StarPU avec contextes d'ordonnancement

Les bibliothèques parallèles qui représentent les briques de base des applications parallèles peuvent avoir des besoins très différents en terme d'ordonnancement, de granularité des calculs ou encore de volume de données traitées. Ordonnancer des tâches correspondant aux différentes bibliothèques en essayant de satisfaire tous leurs besoins à l'aide d'un unique ordonnanceur global peut être très compliqué et difficile de passer à l'échelle. L'approche la plus naturelle dans ce cas est d'ordonnancer les tâches associées à chaque bibliothèque en utilisant la stratégie la plus appropriée, ce qui correspond à ce que fournissent les contextes d'ordonnancement. De plus, il est important de signaler que les contextes apportent une sécurité par rapport au fait que les différents ordonnanceurs ne vont pas se perturber les uns les autres.

3.1.2 Contrôler le partage par le recouvrement

Le recouvrement des contextes, c'est à dire le fait que certaines ressources puissent appartenir à plus d'un contexte, permet d'exploiter des ressources particulières tels que les GPUs au maximum de leur potentiel lorsque les bibliothèques parallèles s'exécutant dans chaque contexte n'ont pas suffisamment de calcul pour pouvoir nourrir à elles seules le/les GPUs associés au contexte. L'approche basée sur l'utilisation de contextes ainsi que leurs recouvrement permet de gérer simplement le cas d'une hiérarchie de bibliothèques parallèles. Par exemple, le contexte associé à une bibliothèque A peut inclure deux sous-contextes qui vont exécuter les bibliothèques parallèles B et C que A utilise et ce avec très peu d'interactions entre les différents contextes. Ce dernier point est une des différences principales avec l'approche suivie dans le projet Lithe. En effet, contrairement à Lithe dans lequel une ressource est attribué à un seul contexte à un instant donné, dans notre approche une ressource associée au contexte de la bibliothèque B peut continuer à être utilisé par la bibliothèque A.

Enfin, les ressources qui appartiennent à plus d'un contexte utilisent, pour le moment, un algorithme "Round-Robin" pour sélectionner le contexte sur lequel elles vont être actives (i.e. exécuter une tâche appartenant au contexte) à un instant donné. Bien entendu, des techniques plus sophistiquées seront envisagées par la suite pour par exemple ne pas avoir un partage

complètement équitable des ressources partagées.

3.1.3 Donner le pouvoir à l'usager

L'efficacité de la plate-forme proposée réside principalement dans la possibilité qu'elle offre de manipuler simplement et de manière dynamique les ressources. Ceci implique de connaître la machine ainsi que le schéma d'exécution de l'application de manière à pouvoir dimensionner raisonnablement les différents contextes d'ordonnancement. L'utilisateur peut, pendant l'exécution de l'application, redistribuer les ressources parmi les contextes pour adapter la taille de ces derniers aux besoins des différentes bibliothèques parallèles utilisées par l'application au cours de son exécution. Ainsi, des opérations telles la suppression, l'ajout ou encore la fusion de contextes sont prévues.

D'autre part, le cloisonnement offert par les contextes proposés permet d'utiliser différentes stratégies d'ordonnancement exploitant chacune une partie de la machine. Ainsi, l'utilisateur peut, en plus de découper logiquement la plate-forme, l'exploiter avec les algorithmes d'ordonnancement adaptés aux calculs effectués par chaque contexte.

Enfin, dans le cas de plates-formes composées de ressources de calcul hétérogènes (CPU/GPU par exemple), les contextes d'ordonnancement peuvent permettre d'exprimer une notion de partage de ressources tels que les GPUs pour améliorer leurs utilisation (ce qui est particulièrement utile lorsque les grains des calculs effectués dans les différents contextes sont très irréguliers).

3.2 Des contextes pour répondre à la demande

Les contextes d'ordonnancement fournissent principalement un support logiciel pour l'implémentation des applications parallèles sur plates-formes hétérogènes. Dans l'absence d'un tel outil les programmeurs sont souvent amenés à faire un compromis entre l'effort nécessaires à reconcevoir certaines parties de leur applications et la performance voulue. Vu la complexité croissante des architectures modernes, il n'est clairement pas envisageable (du point de vue des développeurs d'applications parallèles) de remettre en cause l'existant à chaque évolution architecturale.

La plate-forme présentée dans ce rapport vise à améliorer la portabilité des performances des applications sur les architectures hétérogènes. Ceci impliquera des contributions au niveau des supports d'exécution parallèles ainsi que des travaux pour adapter des bibliothèques spécifiques telles que les bibliothèques d'algèbre linéaire parallèles à ce nouveau contexte. Les outils ainsi obtenus pourront être utilisés par la suite par les développeurs pour concevoir des applications parallèles portables en terme de performance.

3.2.1 Respecter la qualité de service

La composabilité s'adresse aussi aux systèmes embarqués[10] qui ont besoin d'une gestion différente du temps réel dur et du temps réel mou. Ces systèmes ont des besoins très différents allant de contraintes temporelles très strictes à la nécessité d'économie l'énergie. Il est donc important de réussir à répondre à tous les types de contraintes simultanément. L'isolation des aspects dans des contextes différents pourrait permettre de préserver dans un des contexte la satisfaction de l'usager à travers le temps de réponse rapide alors que dans un autre l'accent est mis sur l'économie d'énergie. Les contextes permettent ainsi, d'exprimer assez simplement du point de vue du programmeur des contraintes plus ou moins fortes et éviter les effets de bords induits par des politiques d'ordonnancement globales.

3.2.2 Adapter les bibliothèques spécialisées

Un grand nombre d'applications dans le domaine du calcul scientifique et de la simulation numérique repose sur des bibliothèques d'algèbre linéaire qui fournissent des "noyaux" (opérations matricielles par exemple) efficaces garantissant ainsi de très bonnes performances aux dites applications. Un exemple de telles bibliothèques sont les bibliothèques d'algèbre linéaire dense telles que PLASMA (Parallel Linear Algebra Software Multi-core Architectures)[17] qui se spécialise dans les architecture multi-cœurs ou encore MAGMA (Matrix Algebra on GPU and Multi-core Architectures)[13] qui propose des solution hybride à la fois pour des plates-formes hétérogènes à base de GPUs. Une application utilisant ce type de bibliothèques doit, pour assurer leurs bonnes performances, les confiner dans un sous-ensemble de ressources (ce qui était impossible avant l'introduction des contextes). D'autre part, une même application peut être amenée à utiliser plus d'une bibliothèque parallèle spécialisée et très optimisée. Cette situation ne peut être gérée proprement et de manière portable qu'en utilisant des contextes.

Par ailleurs, il est important de signaler que cette composition du parallélisme peut se faire sur plus d'un niveau. Par exemple, une application parallèle peut utiliser un solveur de systèmes linéaire creux [1], ce dernier utilisant différentes bibliothèques spécialisées d'algèbre linéaire dense pour être performant. Ainsi, dimensionnant les contextes d'ordonnancement pour que toutes ses bibliothèques cohabitent entre elles sans se perturber, il est possible d'avoir un bon comportement quelque soit la plate-forme sous-jacente.

3.3 Limites de l'approche

Cette plate-forme de gestion de l'isolation flexible de plusieurs contexte d'ordonnancement est un premier pas dans l'étude de la composabilité des bibliothèques parallèles. Bien que ce premier prototype est fonctionnel, il reste perfectible sur différents aspects. Premièrement, cette plate-forme a pour vocation de fournir des mécanismes pour le dimensionnement automatique des contextes, aspect qui pose sans doute de nombreux difficultés liées d'une part aux métriques à utiliser pour évaluer un partitionnement et d'autre part au coût d'un tel dimensionnement. D'autre part, un aspect qui devra être approfondi concerne le recouvrement des contextes. Le choix des ressources à partager est fait à partir des instructions fournies par l'utilisateur sans informations repris du support exécutif sur l'état de la machine. Une amélioration dans ce sens est utile pour prendre en compte l'état de la plate-forme lors d'une possible redistribution de ressources.

Généralement la quantité de travail des codes parallèles est souvent irrégulière dans le temps. Ainsi, il serait intéressant d'adapter la taille des contextes pour réagir à ces variations de charge ainsi que d'être capable d'avoir un partage de ressources qui ne soit pas forcément équitables entre les différents contextes d'une même application pour arriver à tirer la quintessence des plates-formes.

3.4 Discussion : vers un parallélisme adaptatif

Cette première étape pour la mise en oeuvre de la composabilité de différents types de parallélisme propose des outils pour la distribution de ressources sur plusieurs contextes d'ordonnancement mais l'utilisateur à besoin d'intervenir pour décider de la répartition des ressources sur les contextes. Il est nécessaire à long terme d'être capable de combiner à la fois la connaissance que l'utilisateur a de son application, ainsi que les information que le support d'exécution peut avoir concernant l'état des ressources.

Dans l'avenir nous proposons d'améliorer cette plate-forme pour aider les applications à s'adapter aux architectures hétérogènes des calculateur parallèles contemporains. À travers un environnement d'évaluation des applications, nous pourrions construire un système capable de fournir un retour précis sur l'exécution de l'application et utiliser ces informations pour prévoir le comportement futur. Avec des prédictions sur les besoins des calculs en terme de ressources il est possible d'adapter la distribution de la puissance de calcul sur les contextes.

Enfin, nous envisageons d'aborder la gestion de la composabilité au sein des architectures hiérarchique (i.e. architectures à accès mémoire non-uniforme). L'obtention de bonnes performances sur de telles plates-formes implique la prise en compte de l'affinité entre calculs et données, la prise en compte de la contention mémoire [19]. D'autre part, le recouvrement des contextes pourra représenter une solution pour contraindre à la fois l'exécution d'un calcul sur une partie de la machine assurant ainsi une bonne localité.

Chapitre 4

Éléments d'implémentation

Au sein de StarPU j'ai implémenté et adapté des fonctionnalités pour permettre l'existence de plusieurs contextes d'ordonnancement. Premièrement, j'ai ajouté une structure de données qui va regrouper plusieurs informations concernant les contextes. Ensuite, j'ai introduit des fonctions qui permettent au programmeur des noyaux de manipuler les contextes. En dernier lieu, j'ai adapté le fonctionnement actuel du moteur de gestion des tâches de manière à prendre en compte l'existence des contextes. Concernant les stratégies d'ordonnancement, elles ont subi des modifications censées à garder une vision locale aux ressources du contexte.

4.1 La structure "sched_ctx"

Cette structure regroupe des données utiles au cloisonnement des ressource :

- un tableau d'indices des "workers" appartenant au contexte
- un pointeur vers la stratégie d'ordonnancement
- un compteur de tâche soumises au contexte et des structures de synchronisation associées
- des tableaux de structure de synchronisation pour les listes de tâches des "workers" appartenant au contexte

Cette structure construit une couche d'abstraction des "workers". En lui fournissant les identifiant des "workers", elle maintient un tableau contenant ces indices qui sera utilisé lorsqu'il faut parcourir la liste des ressources associées au contexte . De plus, elle n'est pas accessible par l'utilisateur bien que son identifiant est utilisé par l'utilisateur pour soumettre une tâche au contexte.

Chaque "worker" a une liste de pointeurs vers la structure car plusieurs ressources peuvent appartenir à différents contextes. De même, chaque tâche connaît l'identifiant du contexte qui va la traiter, car elle ne peut être gérer que par l'ordonnanceur associé.

4.2 Création des contextes

A travers les contextes, l'utilisateur peut gérer la distribution de ses calculs sur les ressources de la machine. En connaissant les "workers" lancés et les stratégies d'ordonnancement disponibles, l'utilisateur peut créer un ou plusieurs contextes d'ordonnancement.

Lors de la création d'un contexte, une barrière de synchronisation sur laquelle tous les "workers" demandés vont s'endormir est positionnée de manière à pouvoir les informer de leur appartenance au nouveau contexte pour ensuite les relancer. Il est essentiel que les "workers" ne soient pas en train d'exécuter ou de récupérer une tâche lorsqu'on modifie leur tableau de contextes

(l'apparition d'un nouveau contexte impliquent la modification et l'addition des "callbacks" pour la gestion des tâches).

La barrière a été implémentée en faisant un pont entre le mode utilisateur et le moteur d'ordonnancement. Plus concrètement, on crée les contexte en mode utilisateur, mais les pilotes ne prennent en compte cette information que lorsqu'ils sont libres. Ainsi, on envoie une notification aux "workers" depuis le mode utilisateur et ceux-ci se bloquent dès qu'ils ont fini leur tâche courante et prennent ensuite en compte la notification. De même, on envoie une notification censée redémarrer les "workers" et on attend que les notifications soient prises en compte.

4.3 Suppression des contextes

La suppression des contextes fonctionne sur le même principe que la création, c'est-à-dire on endort tous les "workers" du contexte pour mettre à jour leur status et ensuite on les redémarre. Mais avant de déclencher ce procédé il faut s'assurer qu'il n'y a plus de tâches appartenant au contexte à détruire. Par conséquent, on attend que la liste associée à chacun des "workers" et correspondant au contexte soit vide.

4.4 Soumission des tâches au contexte

À la création d'un contexte, l'utilisateur récupère un identifiant associé au contexte qui sera utilisé pour spécifier le contexte auquel on doit soumettre une tâche. Ainsi, la tâche sera ajoutée sur une pile d'exécution appartenant à la stratégie d'ordonnancement du contexte et ensuite traitée par le "worker" choisi. Si on ne spécifie aucun contexte la tâche sera soumise au contexte globale et gérée par sa stratégie.

4.4.1 Les stratégies d'ordonnancement dans la présence des contextes

J'ai adapté les stratégies d'ordonnancement pour qu'elles puissent prendre des décisions concernant uniquement les ressources du contexte auquel elles appartiennent.

Lors de la création des contexte, on se retrouve dans la situation dans laquelle plusieurs stratégies d'ordonnancement sont utilisées en même temps sur la machine. Ceci à demandé plusieurs modifications à l'implémentation existante de StarPU. De même, le système devient de plus en plus complexe quand plusieurs contextes peuvent partager des ressources. Dans cette situation, un "worker" peut être manipulé à travers plusieurs stratégies d'ordonnamcent. La difficulté intervient lorsqu'on fait "pop" (i.e. extraire une tâche) parce que il faut savoir laquelle choisir. Actuellement, on utilise un algorithme Round-Robin sur l'ensemble des stratégies d'ordonnancement qui gèrent le "worker".

4.5 Redimensionnement des contextes

Une fois un contexte construit, il est possible de le redimensionner en ajoutant ou en supprimant des ressource dynamiquement. Toutes les modifications apportées à un contexte sont faites en respectant le même principe : les "workers" concernés sont bloqués, les modification appliquées puis ils sont redémarrés.

Cette fonctionnalité est extrêmement utile pour adapter la taille du contexte à la taille du problème traité par ce dernier. En effet, La seule façon de connaître le nombre de ressources nécessaires a l'exécution d'un noyau de calcul (hormis la connaissance que l'utilisateur a de son noyau) est de calibrer ce dernier. Ainsi, lors de l'exécution, si on détecte un mauvais débit ou temps de calcul on peut redistribuer les ressources entre les contextes.

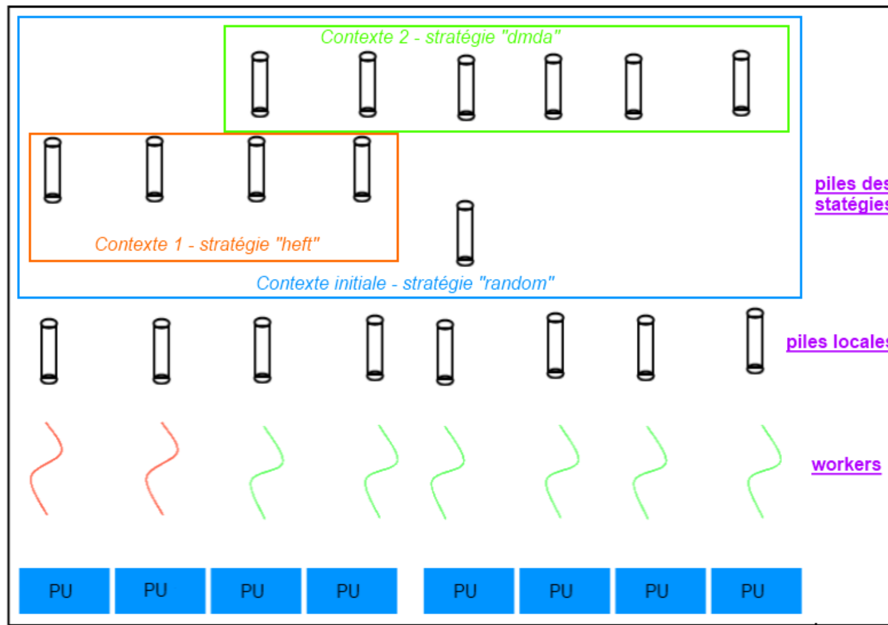

FIGURE 4.1 – Contextes superposés

Actuellement, le système que j'ai construit est basé sur une adaptation statique du nombre de ressources par rapport à la taille du problème, mais il est envisagé de le faire évoluer vers une version plus dynamique.

Chapitre 5

Évaluation

5.1 Environnement d'évaluation

Les expériences présentées dans cette section ont été faites principalement sur les machines suivantes :

- bertha : contient 16 processeurs Intel Xeon X7460 de 6 cœurs, cadencé à 2,66 GHz, 192 Go de RAM ;
- attila : contient 2 processeurs Intel Xeon X5650 de 6 cœurs, cadencé à 2,66 GHz, trois cartes graphiques NVIDIA Tesla C2050, 48 Go de RAM ;
- riri : contient 4 processeurs Intel Xeon E7-4870 de 10 cœurs, cadencé à 2,4 GHz, 1 TB de RAM

Comme routines BLAS nous avons utilisé MKL pour les noyaux exécutés sur CPU et CUBLAS pour les noyaux exécutés sur la carte graphique NVIDIA. Les trois machines s'appuient sur un noyau Linux version 2.6.

Pour évaluer la répartition du travail sur les ressources et faire du "profiling post-mortem" nous avons utilisé le logiciel Vite version 1.2.

5.2 Cas du Cholesky

Nous allons étudier le comportement de la plate-forme de composabilité de codes parallèles sur une implémentation de la factorisation Cholesky.

5.2.1 Sur la factorisation Cholesky...

La factorisation Cholesky[2] est une méthode utilisée pour la résolution des systèmes linéaires $Ax = b$, où A est une matrice symétrique définie positive. Ce genre de résolutions sont souvent utilisées dans la modélisation numérique des phénomènes physiques. Bien qu'en général la matrice A est creuse (c'est à dire qu'elle contient beaucoup d'éléments nuls), la factorisation à laquelle nous allons nous intéresser est la factorisation de matrices denses. En effet, les opérations sur matrices denses bien que n'apparaissant pas dans la simulation numérique, sont une brique de base pour un grand nombre de méthodes de résolution. D'autre part, les opérations matricielles en algèbre linéaire dense sont souvent caractérisées par leur régularité ce qui les rend plus facile à analyser pour une première évaluation.

5.2.2 Pourquoi Cholesky...

La factorisation Cholesky représente un bon benchmark pour l'évaluation de cette plate-forme de composabilité parce que le coté algorithmique est suffisamment régulier pour pouvoir gérer facilement le parallélisme. D'autre part, les autre types de factorisation dense fonctionnent plus ou moins sur le même principe (i.e. les graphes de tâche ont des structures similaires). Notons de plus qu'il est possible de fixer la taille de la matrice ainsi que la granularité des tâches pour pouvoir nourrir des machines de plus ou moins grande taille. Enfin, il est important de signaler que le comportement de ce genre d'algorithmes est bien connu ce qui rend l'analyse des résultats plus simples surtout lorsqu'il s'agît d'évaluer de nouveaux mécanismes au niveau des supports d'exécution.

5.3 Stratégie d'évaluation

Évaluer la plate-forme de composabilité des codes parallèles a représenté sans doute un point délicat. La difficulté se trouve plus précisément dans la comparaison du comportement de l'application avec et sans l'utilisation des contextes d'ordonnancement. Notre scénario expérimental va consister à lancer deux applications simultanément. Ces dernières sont constituée chacune d'une série de factorisations de Cholesky indépendantes. D'autre part, nous avons fait en sorte que la taille des matrices associées à chacune des deux applications soient différentes (i.e. la première traitera des matrices de taille importante alors que la seconde sera confrontée à des matrices de taille intermédiaire). Dans le cas où la composabilité n'est pas gérée, StarPU ordonnance les tâches associées à chacune des deux applications avec le même ordonnanceur ce qui rend l'évaluation des performances de chacune de nos applications de test compliquée. Ainsi, dans les points suivantes on va proposer une métrique qui permet une comparaison pertinente entre les deux version de StarPU.

5.3.1 Points visés

Le premier aspect que nous avons eu à évaluer est le sur-coût du développement de cette plate-forme au sein de StarPU. En effet, il est important de quantifier la pénalité induite par la gestion des contextes au sein de StarPU. Ensuite, il est intéressant d'obtenir la taille la plus adaptée d'un contexte en fonction de la taille du problème et des ressources disponibles. De tels résultats sont importantes pour la future mise en place d'une stratégie de dimensionnement automatique des contextes. Néanmoins, il faut évaluer l'isolement des contextes, observer le débit des contextes dans la présences d'autre contextes et observer jusqu'à quel point les contexte ne se perturbent entre eux. Même s'ils ne partagent pas des ressources de calcul, généralement ils partagent la mémoire et la contention du bus mémoire peut engendrer de mauvaises performances.

D'autre part, nous nous sommes intéressés à l'évaluation des la stabilité des performances sur différents types de plates-formes allant d'un "petit" noeud avec des GPU à une machine à 96 cœurs.

Enfin, nous allons évaluer le recouvrement des contextes vis à vie du non partage de ressource (isolement des contextes). Il est intéressant de pouvoir mettre en avant le type des ressources les plus appropriées au partage ainsi que leur nombre.

5.3.2 Métrique

Premièrement, notons que les contextes peuvent cloisonner des calculs parallèles de tout type, qui reçoivent des données d'entrée variées et sans doute ne vont ni commencer ni terminer

en même temps. Ainsi une métrique qui concerne le temps total d'exécution ne peut pas être suffisamment précise. Ceci est particulièrement vrai lorsqu'on fait varier le nombre des ressources.

L'approche retenue est de calculer l'efficacité de la machine en prenant en compte la quantité de travail totale (i.e. la somme des quantité de travail de chacune des applications) et de considérer le temps nécessaire au traitement de l'intégralité de ce travail. L'idée est de pouvoir calculer la vitesse à laquelle la quantité de calcul a été faite et de comparer cette dernière au cas ou la plate-forme de composabilité n'a pas été utilisée. Bien entendu, dans le cas des plates-formes homogènes, la vitesse obtenue peut être ramenée à celle d'un processeur afin de pouvoir étudier le comportement des applications lorsque le nombre de processeurs varie.

Pour résumer, nous avons lancé deux flux d'exécution qui effectuent chacun une série de factorisations de Cholesky sur des matrices de taille différentes. Dans la version sans gestion de la composabilité les deux flux vont s'exécuter sur toute la machine alors que dans la version avec contextes chaque flux est exécuté dans un contexte différent.

5.4 Performances sur les architectures homogènes

5.4.1 Sur-coût de la plate-forme de composabilité des codes parallèles

La première étape pour évaluer la plate-forme a été de vérifier si l'introduction de la plate-forme de composabilité au sein de StarPU a eu un impact négatif sur les performances du support exécutif. Ainsi, nous avons lancé le noyau de calcul de la factorisation Cholesky sur le système sans utiliser de contextes et nous en avons fait de même en utilisant un seul contexte. Les résultats obtenus sont présentés dans la figure 5.1. On peut observer que les valeurs des deux expériences sont comparables et que le sur-coût induit par l'utilisation d'un contexte reste très marginal.

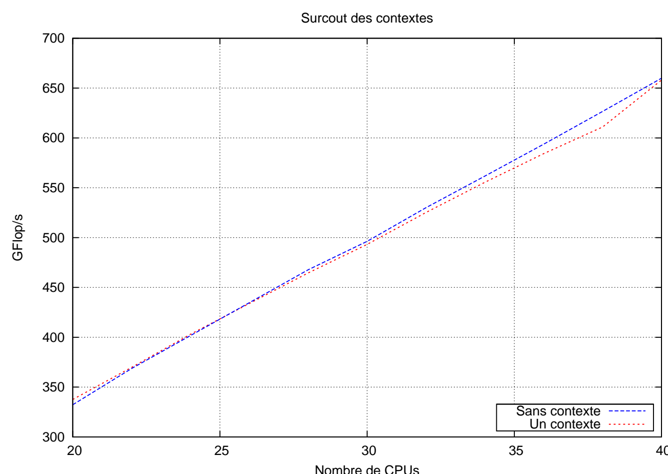

FIGURE 5.1 – Évaluation du sur-coût des contextes sur une machine homogène

5.4.2 Stabilité des résultats

Pour évaluer les performances obtenues par la prise en compte de la composabilité sur une machine homogène, nous avons choisi la plate-forme "riri" pour lancer l'exécution de deux séries de factorisations de Cholesky, la première sur des matrices de 50000x50000 et l'autre sur des matrices 20000x20000, chaque matrice ayant été découpée par blocs de 500x500 éléments.

Dans la figure 5.2, nous présentons une courbe normalisée (par rapport à l'efficacité observée sur la meilleure configuration de StarPU) de l'efficacité d'une exécution utilisant deux contextes. Nous pouvons observer que sur ce type de machines le support exécutif StarPU se comporte mieux lorsque les contextes sont utilisée. Bien entendu, cette remarque ne tient plus lorsque la répartition des ressources sur les deux contextes n'est pas raisonnable. Nous pouvons voir que les gains sont de l'ordre de 5% et atteignent un maximum de 7%.

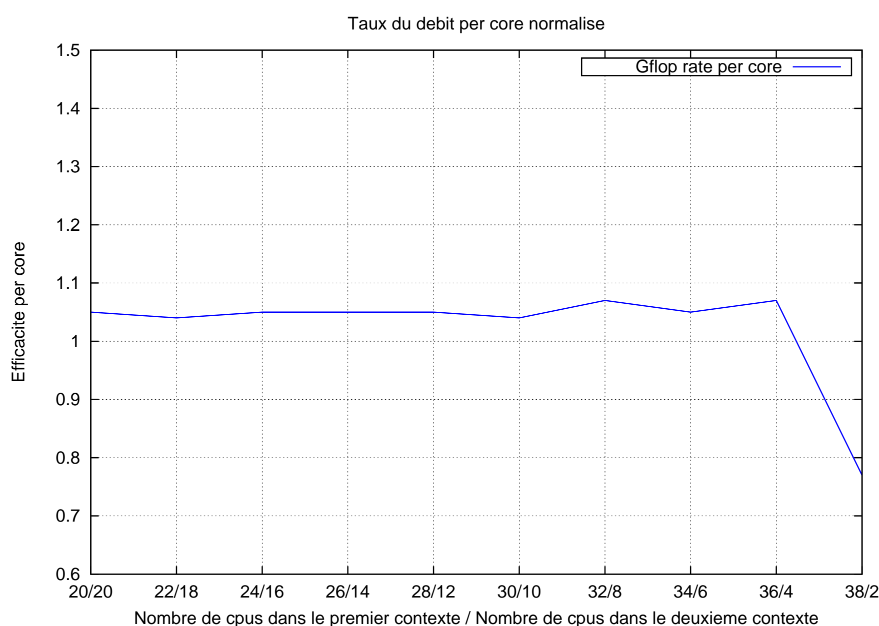

FIGURE 5.2 – Efficacité des coeurs sur Riri

On peut de plus observer sur ce graphique l'importance de la dimension des contextes pour l'obtention d'un bonne performance. Pour autant, le mécanisme de redimensionnement dynamique présenté dans les chapitres précédents peut permettre de corriger ces problèmes. Dans l'expérience correspondant à la figure 5.2 le redimensionnement n'apparaît qu'a la fin de la petite application impliquant que le contexte correspondant à l'application de grande taille se répand sur toute la machine. Ceci explique en partie la grande stabilité des performances. Enfin, on peut voir que la meilleure performance est obtenue lorsque la distribution des ressources sur les deux contextes est raisonnables par rapport à leur quantité de travail respective.

Dans le futur, il serait intéressant de traiter le redimensionner des contextes à chaud durant l'exécution pour adapter la taille de ces derniers aux caractéristiques des applications qu'ils gèrent.

5.4.3 Scalabilité sur une très grande machine

Pour évaluer la scalabilité de la plate-forme de composabilité nous avons effectué les mêmes expériences que précédemment sur la plate-forme“bertha”, une machine complexe qui est caractérisée par des mécanismes de gestion de la cohérence des caches très particulière ainsi qu'un débit mémoire limité.

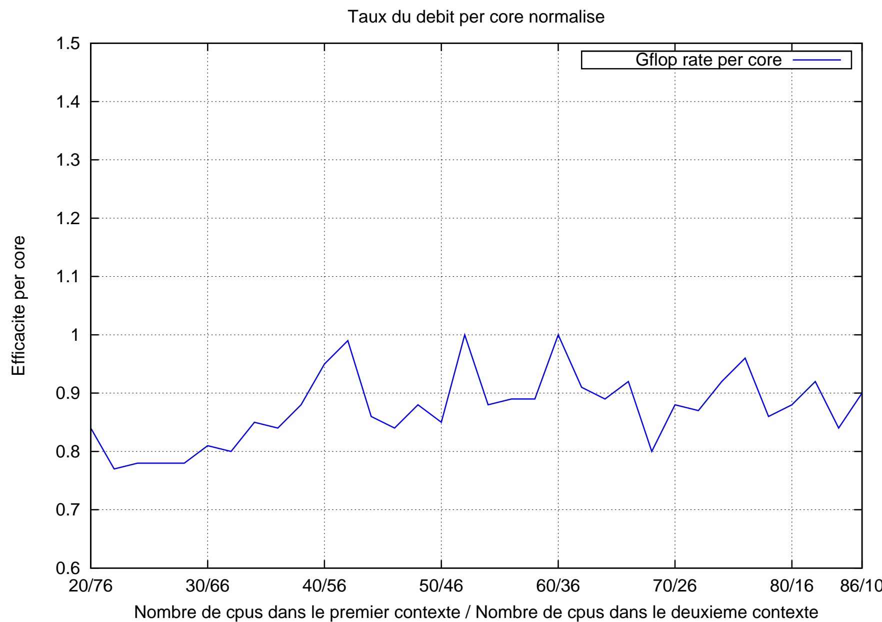

Figure 5.3 – Efficacité des coeurs sur bertha

Dans la figure 5.3 on peut voir un rapport plutôt défavorable pour la version de StarPU avec contextes. Ceci illustre une certaine difficulté du moteur de gestion des contextes à exploiter une telle machine. Il est important sur ce genre de plates-formes fortement NUMA d'étudier les limitations des deux versions pour améliorer les comportement du support d'exécution. En effet, sur une telle plate-forme, aucune des deux version de StarPU n'a un comportement stable et satisfaisant. Nous pensons qu'il serait important de prendre en compte les aspects liés à la localisation des données sur ce genre de plates-formes (i.e. prendre en compte les pénalités liées aux accès à des données distantes).

5.5 Comportement sur une machine hétérogènes

Afin d'évaluer le comportement de la plate-forme de composabilité sur une architecture hétérogène nous avons choisi la machine “attila” pour lancer l'exécution de deux séries de factorisations de Cholesky, la première sur des matrices de 20000x20000 et l'autre sur des matrices 10000x10000.

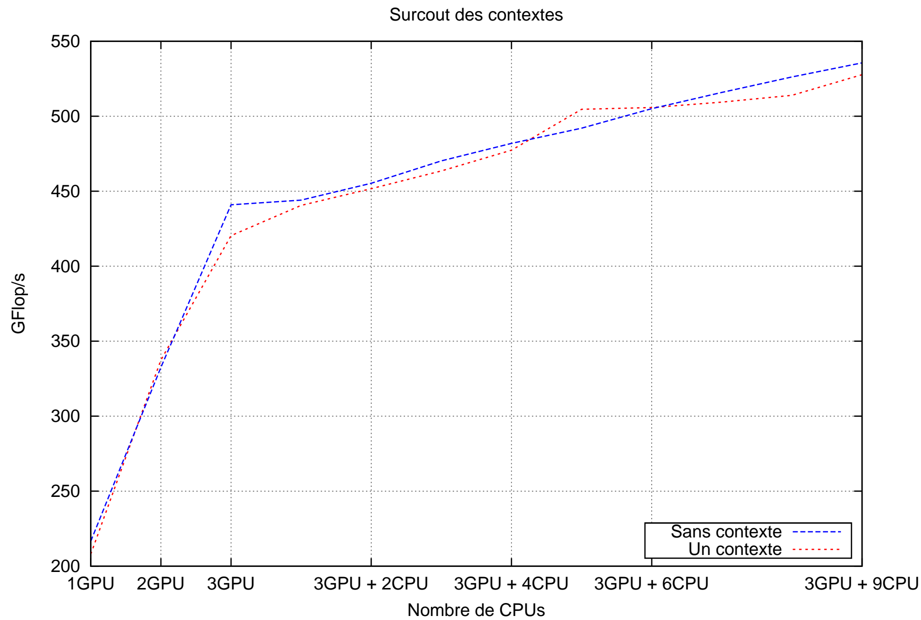

FIGURE 5.4 – Évaluation du sur-coût des contextes sur une machine hétérogène

5.5.1 Sur-coût de la plate-forme de composabilité des codes parallèles

Premièrement nous avons centré notre intérêt sur le sur-coût du système de gestion des contextes et pour faire une comparaison pertinente, nous avons lancé un seul noyau de calcul sur le système sans contexte et l'avons comparé avec une version contenant un seul contexte. Dans la figure 5.4, on peut observer un très faible affaiblissement des performances qui est du à l'introduction mécanismes de gestion des contextes.

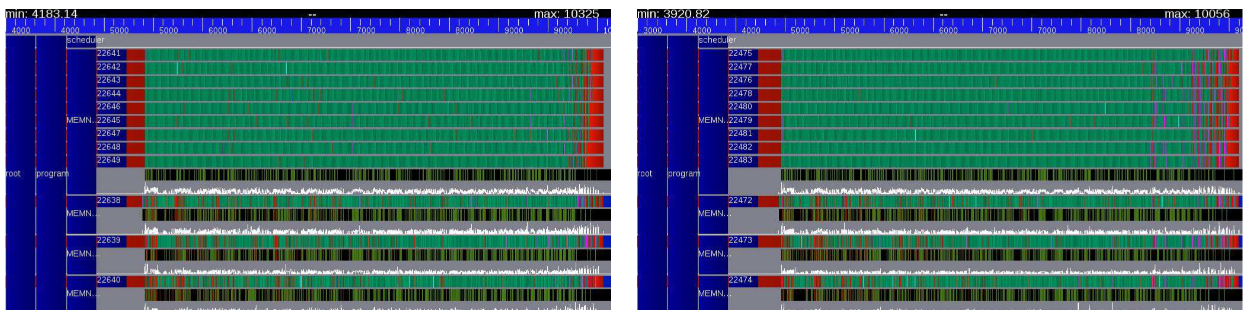

FIGURE 5.5 – a. Sans contexte b. Un contexte

Cependant on peut constater grâce aux traces d'exécution générées grâce à l'outil vite (voir figure 5.5) une exploitation des ressources similaires dans les deux cas.

5.5.2 Répartition isolée des ressources sur les contextes

En exécutant une application "composée" sur une machine hétérogènes on observe l'aspect critique lié à la distribution des ressources sur les contextes. Lorsqu'on utilise les GPU, il faut prendre en compte le fait que le grain des calculs effectués sont de tailles plus importantes que celui utilisé pour les processeurs. La figure 5.6 donne une comparaison entre les performance de la version basée sur les contextes et la version standard à travers une courbe normalisée par rapport à la meilleur performance de la version standard sur cette plate-forme. Ainsi on peut observer dans la figure 5.6 que l'efficacité de la machine augmente lorsqu'on attribue plus de GPU, ou encore lorsqu'on donne plus de GPU au premier contexte qui correspond à la factorisation de la grande matrice. De façon similaire à l'architecture homogène, attribuer la majorité des ressources au calcul le plus gros engendre généralement une chute dans les performances lorsqu'on fourni très peu de ressource de calcul à l'autre contexte.

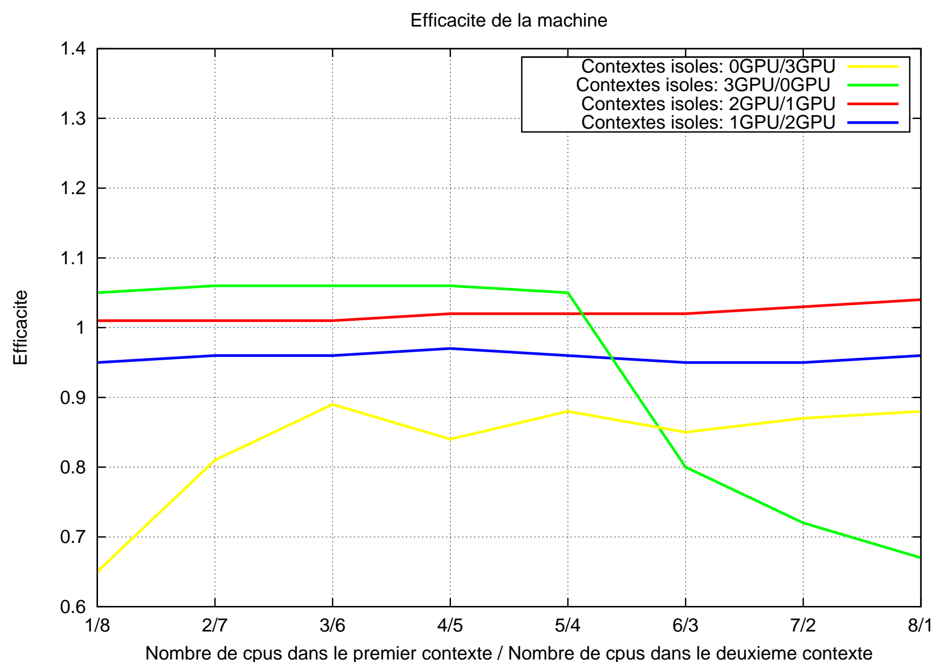

FIGURE 5.6 – Efficacité de "attila"

À travers cette expérience on arrive à évaluer les besoins des calculs en terme de ressources. En comparant les résultats de notre plate-forme à ceux de la version standard de StarPU, on observe un gain lorsque la distribution des cartes graphiques sur les contextes est pertinente à l'égard de la quantité de travail des contextes. La figure 5.6 met en évidence l'importance d'un dimensionnement correct et à chaud des contextes, contrairement à un lors de la terminaison d'un des contextes, qui est déjà tardive sans laisser la possibilité de rattrapé le temps d'inactivité des autres calculs.

5.5.3 Recouvrement des contextes

Lorsqu'on traite des architecture hétérogènes, il est souvent difficile de répartir les ressources de manière à obtenir de bonnes performances. La différence en terme de granularité de calcul rend compliqué un partitionnement rigide de la machine. Le recouvrement peut être une alternative pour partager équitablement ou non des ressources qui dans notre cas seront les GPU.

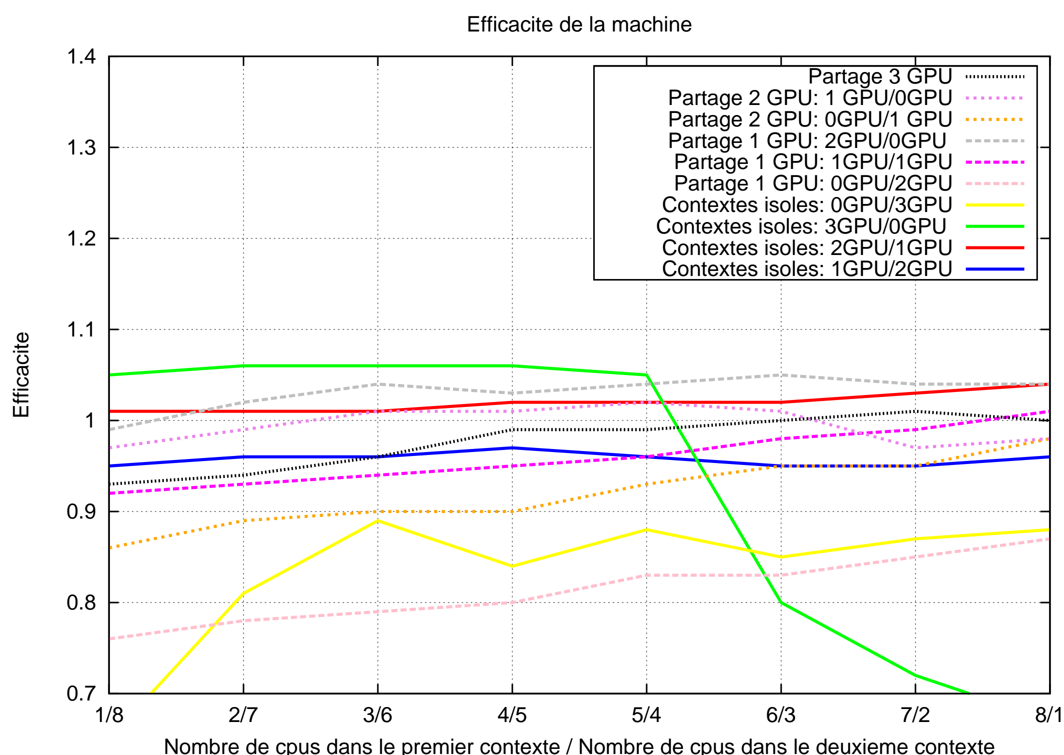

Figure 5.7 – Efficacité de "attila" lors du partage de ressources

La figure 5.6 présente des résultats montrant l'efficacité du partage de ressources. Nous pouvons observer qu'il est avantageux de partager un GPU entre les deux contextes et d'attribuer les deux autres GPU au plus gros des deux contextes. Ce résultat indique que le deuxième calcul n'a pas suffisamment de travail pour exploiter efficacement 1 GPU à plein temps.

Bien que la courbe correspondant à 3 GPU attribués au gros contexte domine la figure, on peut voir qu'elle n'est pas très flexible et qu'elle répondrait difficilement à une redistribution de ressources lors d'un dimensionnement adaptatif.

Chapitre 6

Conclusions

Dans ce mémoire, nous nous sommes intéressés à la gestion efficace de plusieurs sources de parallélisme simultanées au sein d'une même application. La motivation de l'étude réside dans le besoin croissant de pouvoir réutiliser plusieurs bibliothèque parallèle dans un même programme. Dans l'état actuel des choses, les bibliothèques parallèles sont incapables de se partager des unités de calcul. Pire, elles entrent en compétition de manière aveugle pour s'exécuter sur la totalité de la machine et peuvent finalement se gêner au détriment des performances du programme.

Notre contribution à cette problématique est d'avoir conçu et développé une plate-forme au niveau exécutif capable de gérer efficacement l'exécution simultanée de plusieurs noyaux de calcul parallèles. Cette plate-forme s'appuie sur le support d'exécution StarPU capable d'exploiter des configurations hétérogènes CPU + GPU.

Nous avons introduit la notion de contexte d'ordonnancement qui abstrait le fait de partager des unités de calcul entre différentes bibliothèque parallèles tout en leur donnant du contrôle et des garanties sur l'ordonnancement de leur tâches.

Ce travail s'est principalement focalisé sur l'allocation dynamique des ressources avec l'objectif de maximiser les performances de plusieurs calculs parallèles s'exécutant simultanément. La redistribution des ressources s'effectue de manière transparente pour les bibliothèques, en redimensionnant les contextes de manière dynamique.

Cette plate-forme permet donc une certaine flexibilité quant aux critères utilisés pour déclencher des redistributions : ressources les mieux appropriées, taux d'occupation des processeurs, nombre d'opérations effectives par seconde, etc. Elle fournit les outils pour choisir entre l'isolation totale des contextes et le partage des ressources. Des ressources comme les cartes graphiques sont en effet parfois plus efficaces lorsqu'elles sont partagées parmi plusieurs calculs parce qu'elles peuvent être difficilement exploitables à 100% par une seule bibliothèque.

Disposant des outils pour optimiser facilement la composabilité dans une application, avec un effort minimal de programmation, le programmeur peut facilement améliorer les performances de son calcul. Même si cette plate-forme souffre encore de quelques limitations elle fournit déjà des bons résultats et elle montre du potentiel pour les développements futurs.

Nous avons évalué l'apport des contextes au sein de StarPU à travers l'exécution d'une application composée de plusieurs noyaux de calculs parallèles. Nous avons constaté des bons résultats et, en outre, un comportement favorable à la redistribution des ressources ce qui confirme l'intérêt dans l'avenir de gérer le dimensionement automatiquement des contextes. D'ailleurs, nous avons observé qu'il pourrait être avantageux de permettre aux contextes de partager les même ressources, lorsqu'ils n'arrivent pas à fournir assez de travail à ces ressources pour les rendre efficaces.

Pour construire les contextes on a pris en compte les indications fournies par le programmeur, en considérant qu'il connaît très bien les spécificités de son calcul. Cependant, le support exécutif

peut améliorer l'exécution du programme là où le programmeur ne dispose pas d'informations suffisantes (efficacité du code sur chaque type de processeur/accélérateur).

Ce travail représente ainsi le premier pas vers la construction d'un environnement capable de s'adapter aux architectures hétérogènes des calculateurs parallèles contemporains. La flexibilité de ce système de gestion de la composabilité permet la construction des contextes en fonction des indications de l'usager aussi bien qu'en fonction des informations fournis par le support exécutif. Nous envisageons mettre en place un système d'auto-calibrage des paramètres applicatifs qui pourrait obtenir au cours de l'exécution et avec précision les besoins de l'application et déterminer la distribution des ressources parmi les contextes. Les perspectives de ce travail s'orientent également vers la gestion des architectures hiérarchiques. Le recouvrement des contextes pourra représenter une solution pour répondre aux besoins de ces machines lors de la gestion de la composabilité.

Finalement, nous sommes persuadés que cette plate-forme représentera un outil important pour les applications de simulations d'aujourd'hui utilisant un nombre important de bibliothèques parallèles spécifiques. A cause de la nature de plus en plus composite des applications et bibliothèques, sans doute cette plate-forme servira à plus d'un niveau de parallélisme et permettra ainsi d'importer le concept de réutilisation et modularité dans le monde de Calcul Haute Performance.

Bibliographie

[1] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Hatem Ltaief, Samuel Thibault, and Stanimire Tomov. Qr factorization on a multicore node enhanced with multiple gpu accelerators. *25th IEEE International Parallel and Distributed Processing Symposium*, 2011.

[2] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Jean Roman, Samuel Thibault, and Stanimire Tomov. Dynamically schedule cholesky factorization on multicore architectures with gpu accelerators. *Symposium on Application Accelerators in High Performance Computing*, 2010.

[3] Cédric Augonnet. The implementation of the cilk-5 multithreaded language. *Literature Study*, 2008.

[4] Cédric Augonnet. Vers des supports d'exécution capable d'exploiter les machines multicoeurs hétérogènes. *Mémoire de Master Recherche*, 2008.

[5] Cédric Augonnet and Raymond Namyst. A unified runtime system for heterogeneous multicore architectures.

[6] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. Starpu : a runtime system for scheduling tasks over accelerator-based multicore machines. *Rapport de Recherche*, 2010.

[7] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu : a unified platform for task scheduling on heterogenenous multicore architectures. *Concurency and Computation : Practice and Experience*, 2010.

[8] Eduard Ayguade, Rosa M. Badia, Francisco D. Igual, Jesus Labarta, Rafael Mayo, and Enrique S. Quintana-Orti. An extension of the starss programming model for platforms with multiple gpus.

[9] François Broquedis, Nathalie Furmenton, Brice Goglin, Raymond Namyst, and Pierre-André Wacrenier. Dynamic Task and Data Placement over NUMA Architectures : an OpenMP Runtime Perspective. *International Workshop on OpenMP*, 2009.

[10] Dazhi Chen and Pramod K. Varshney. Qos support in wireless sensor network : a survey. 2007.

[11] Romain Dolbeau, Stéphane Bihan, and François Bodin. Hmpp : A hybrid multi-core parallel programming environment.

[12] Brian Goetz. Java theory and practice : Dynamic compilation and performance measurement. 2004.

[13] Magma. http ://icl.cs.utk.edu/magma/.

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

[15] H. Pan. Cooperative hierarchical resource management for efficient composition of parallel software. *Phd Thesis*, 2010.

[16] H. Pan, B. Hindman, and K. Asanovie. Composing parallel software efficiently with lithe. *PLDI*, June 2010.

[17] Plasma. http ://icl.cs.utk.edu/plasma/.

[18] Intel TBB. http ://threadingbuildingblocks.org/documentation.php.

[19] S. Williams, K. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. De l'exécution structurée d'applications scientifiques openmp sur architectures hiérarchiques. *Thèse*, 2010.

[20] Cuda Zone. http ://developer.nvidia.com/category/zone/cuda-zone.