StarPU : un support exécutif unifié pour les architectures multicoeurs hétérogènes

Cédric Augonnet

▶ **To cite this version:**

Cédric Augonnet. StarPU : un support exécutif unifié pour les architectures multicoeurs hétérogènes. 19ème Rencontres Francophones du Parallélisme, Sep 2009, Toulouse, France. inria-00411581

HAL Id: inria-00411581
https://inria.hal.science/inria-00411581

Submitted on 28 Aug 2009

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.

RenPar'19 / SympA'13 / CFSE'7
Toulouse, France, du 9 au 11 septembre 2009

StarPU : un support exécutif unifié pour les architectures multicœurs hétérogènes

Cédric Augonnet – cedric.augonnet@inria.fr

Équipe RUNTIME – INRIA Bordeaux Sud-Ouest – LaBRI
351, cours de la Libération, 33405 Talence, France

Résumé
En conjonction avec les processeurs multicœurs, désormais omniprésents, l'utilisation d'architectures spécialisées telles que les processeurs graphiques ou le CELL est une tendance forte du calcul haute performance.
Atteindre les performances théoriques de ces architectures est un objectif difficile. Si de nombreux efforts ont d'ores et déjà été portés sur les accélérateurs, l'utilisation de toutes les ressources de calcul, simultanément, reste un véritable défi.
Nous avons donc conçu STARPU, un support exécutif original qui fournit un modèle d'exécution unifié afin d'exploiter l'intégralité de la puissance de calcul tout en s'affranchissant des difficultés liées à la gestion des données. STARPU offre par ailleurs la possibilité de concevoir facilement des stratégies d'ordonnancement portables et efficaces.
Nous avons mis en œuvre quelques stratégies d'ordonnancement sélectionnables de manière transparente lors de l'exécution. Cela nous a permis d'étudier l'impact de l'ordonnancement sur quelques algorithmes d'algèbre linéaire. Au-delà d'une réduction substantielle des temps d'exécution, STARPU obtient des accélérations *super-linéaires* grâce à sa capacité à tirer un réel avantage des spécificités des machines hétérogènes.

Mots-clés : Support exécutif, Multicœur hétérogène, Accélérateur, Ordonnancement

1. Introduction

Les architectures multicœurs sont désormais majoritaires, non seulement dans les ordinateurs de bureaux, mais aussi dans les grappes ou les super-calculateurs. Bon nombre de machines sont également équipées de cartes graphiques puissantes, et des processeurs multicœurs hétérogènes tels que le CELL sont présents aussi bien sur des consoles de jeu grand public que dans le super-calculateur ROADRUNNER.
En conjonction avec les accélérateurs de calcul tels que les cartes graphiques (GPU), l'utilisation de cœurs spécialisés s'avère très bénéfique en termes de capacité de calcul et de performance énergétique. Les architectures de demain ne comporteront donc pas simplement plus de cœurs, mais certains d'entre eux seront conçus pour des tâches bien spécifiques. Si cette approche offre une solution à certaines limites physiques, elle introduit cependant de nombreuses problématiques, que ce soit au niveau des modèles de programmation, des compilateurs, ou encore de la conception d'architectures passant à l'échelle. Il est donc crucial de disposer d'un support afin de tirer parti de ces architectures complexes.
Nous avons donc créé STARPU, un support exécutif offrant une interface très expressive qui permet aux applications de guider l'ordonnancement tout en s'affranchissant d'une gestion explicite des mouvements de données. STARPU se charge donc d'exécuter des tâches simultanément sur l'ensemble des ressources de calcul, que ce soit des processeur ou des accélérateurs tels que des GPU, ou des CELL.
Dans cet article, nous étudions quels sont les objectifs d'un tel support exécutif et comment STARPU fournit ces différents services. Nous décrivons alors la bibliothèque de gestion des données de STARPU, et comment il est possible de mettre en place des stratégies d'ordonnancement portables. Nous étudions alors les performances obtenues par STARPU sur différentes plate-formes hétérogènes. Enfin, nous comparons notre approche aux travaux apparentés avant de conclure.

2. Quels besoins pour un support exécutif destiné au multicœur hétérogène ?

Les architectures multicœurs hétérogènes nécessitent donc un support particulier : cette section a pour but d'établir quels en sont les objectifs et les contraintes.

Un modèle d'exécution unifié

Afin de satisfaire les besoins de portabilité, il est indispensable de modéliser les traitements de manière uniforme, malgré l'hétérogénéité des ressources de calcul. L'utilisation d'un paradigme du type *passage de message*, ou même de *multithreading* explicite, n'est pas vraiment adapté pour exploiter des machines dont on ne connaît pas la nature *a priori*. En revanche, le **parallélisme de tâche** permet de modéliser des applications de manière efficace et portable, dès lors qu'il existe un moyen de bien distribuer ces tâches sur les différentes plate-formes visées.
Dans le cadre de machines hétérogènes, ce paradigme peut se traduire par des tâches *multi-versionnées*, c'est à dire qu'il est possible d'implémenter la même tâche sur différentes architectures, et d'exécuter la bonne version selon l'architecture à laquelle la tâche a été affectée.

Gestion des données

L'utilisation d'accélérateurs implique une gestion explicite des mouvements de données au sein de la machine : lorsqu'un calcul est affecté à une carte graphique, il est nécessaire que ses données soient présentes sur la carte avant d'effectuer le calcul. Bon nombre de travaux autour des accélérateurs négligent encore totalement cette problématique.
Il est fréquent que les applications supposent qu'il suffit de charger les données à l'initialisation de l'application, et de récupérer le résultat à la fin. Cette approche ne permet pas d'exploiter plusieurs ressources de calculs simultanément. Quand bien même il serait possible de traiter les différentes données indépendamment, la répartition des traitements (et donc des données) reste à la charge du programmeur, ce qui est incompatible avec notre objectif de portabilité.
Une autre approche consiste à déplacer les données entre la mémoire principale et l'accélérateur avant et après chaque tâche. Bien que plus flexible, cette solution s'avère inefficace dès lors que le transfert des données n'est pas négligeable devant celui du calcul lui-même. Cette approche n'est pas viable à terme car les processeurs évoluent bien plus rapidement que les technologies de *bus*, qui deviennent un goulet d'étranglement majeur.

Optimisation des performances

Il est particulièrement difficile de choisir comment répartir les traitements sur les différentes ressources de calcul de manière statique. En revanche, le support exécutif doit être capable de prendre ces décisions de manière dynamique et transparente pour le programmeur.
Un écueil possible est de choisir *a priori* sur quelles architectures peuvent s'exécuter les différentes tâches, par exemple en implémentant certains traitements sur GPU uniquement. Une telle approche statique ne peut pas tenir compte des nombreux paramètres qui déterminent le meilleur choix, selon l'état de la machine, tout en restant portable. Dans notre modèle, le programmeur ne prend donc pas cette décision et laisse au support exécutif le soin de répartir les différentes tâches multi-versionnées de manière portable, et efficace.

3. StarPU, un support exécutif unifié

Nous présentons maintenant StarPU, un support exécutif destiné à exploiter les machines hétérogènes et à répondre aux problématiques soulevées précédemment. Dans cette section, nous montrons comment StarPU modélise et exécute des tâches de manière portable.

3.1. Interface de programmation

La notion de tâche multi-versionnée est donc une des principales structures de données que StarPU fournit. Elle se compose de deux structures distinctes avec d'une part la notion de **codelet** et d'autre part celle de **tâche** à proprement parler. Comme le montre la Figure 1, une codelet modélise un noyau de calcul multi-versionné. Il s'agit d'une structure qui précise sur quelles architectures le noyau de calcul

$$f = \begin{cases} f_{cpu} \\ f_{gpu} \\ f_{spu} \\ \dots \end{cases}$$

FIG. 1: Une codelet et ses différentes implémentations

$$\underbrace{\overbrace{f}^{\text{codelet}}\ \overbrace{(A,B)}^{\text{données}}}_{\text{tâche}}$$

FIG. 2: Une tâche : une codelet appliquée à des données

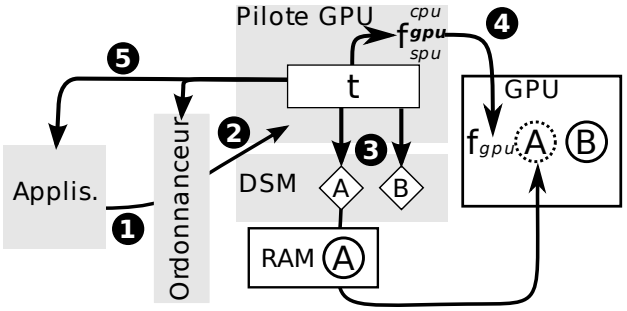

FIG. 3: Cheminement d'une tâche t calculant f(A,B)

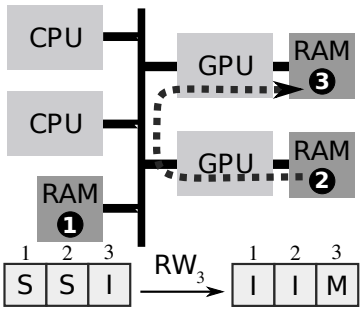

FIG. 4: Protocole de cohérence MSI

peut s'effectuer, et les implémentations associées. Une tâche STARPU consiste à appliquer une codelet sur un jeu de données. Pour chaque donnée, le programmeur précise un mode d'accès (*i.e.* lecture et/ou écriture). La structure de tâche ainsi montrée sur la Figure 2 comporte par ailleurs certains champs optionnels, qui permettent par exemple de préciser un **callback** qui sera exécuté une fois la tâche terminée, ou encore des indications pour guider l'ordonnancement tels qu'un niveau de priorité.

3.2. Structure de STARPU

Dans cette section, nous allons suivre le cheminement d'une tâche depuis sa soumission jusqu'à la notification de sa terminaison à l'application. La Figure 3 montre l'exemple d'une tâche qui consiste à appliquer la codelet *f* qui agit sur deux données A et B.

1. STARPU récupère les tâches que l'application a soumises de manière asynchrone
2. Il faut choisir où exécuter cette tâche : **l'ordonnanceur** distribue les tâches selon la politique d'ordonnancement sélectionnée. Ici, la tâche a été attribuée à une carte graphique : la description de la tâche est donc envoyée au **pilote** associé à cette carte (chaque unité de calcul est associée à un pilote spécifique). Quand le pilote est disponible, il demande une nouvelle tâche à l'ordonnanceur.
3. Avant tout calcul, il faut s'assurer que les données sont disponibles sur la carte graphique. Afin de masquer les transferts de données que cela implique, STARPU dispose d'une bibliothèque qui met en œuvre une *mémoire partagée virtuelle* (ou **DSM** pour *Distributed Shared Memory*). Ici, la tâche doit accéder aux données A et B. Le rôle de la DSM est double : elle permet de savoir que B est ici déjà présente (et à jour) sur le GPU, et que A n'est présente que dans la mémoire principale ; elle se charge par ailleurs de répliquer A (ou de la migrer dans le cas où l'on modifierait A) sur le GPU. Les mouvements de données ainsi induits sont donc transparents pour le programmeur.
4. Une fois les données présentes, le pilote déporte (*offloade*) le calcul sur le GPU, c'est-à-dire qu'il exécute la fonction correspondant à un GPU associée à la codelet.
5. Lorsque le pilote a détecté la fin du traitement, il peut le notifier à l'application si elle l'a demandé. Les éventuelles tâches qui dépendaient de celle qui vient d'être effectuée sont alors traitées par l'ordonnanceur.

4. Gestion des données

Afin d'assurer la disponibilité et la cohérence des données sur les différentes ressources de calcul, StarPU d'une dispose bibliothèque qui met en œuvre une DSM logicielle. Cette bibliothèque, que nous avons décrite en détails lors de précédents travaux [1], propose une interface de haut niveau, de telle sorte que pour StarPU, chaque donnée est un objet abstrait dont on assure la cohérence et les déplacements de manière générique.

4.1. Cohérence des données

Une première approche pourrait consister à garder toutes les données en mémoire principale, et à y accéder avant et après chaque calcul. Cette approche, dite « *write-through* », impose cependant une activité très soutenue sur le bus, qui est pourtant un goulet d'étranglement majeur. En revanche, une approche « *write-back* » effectue les transferts de données de manière paresseuse, c'est à dire lorsque qu'ils sont strictement nécessaires. Afin de maintenir des données cohérentes, il faut que StarPU garde une trace de tous les endroits où sont répliquées chacune de ces données. Ainsi lorsque l'on cherche par exemple à accéder à une matrice depuis un GPU, on vérifie si cette donnée est directement présente sur le GPU, dans le cas contraire on cherche un répliquât de la donnée puis on le copie sur le GPU. Il faut toutefois noter que ce protocole distribué n'est pas directement utilisable dans le contexte des accélérateurs : il est parfois impossible de transférer des données directement entre deux nœuds mémoires (par exemple les cartes NVIDIA ne permettent pas encore de communications inter-GPU), de telle sorte qu'il est parfois indispensable d'utiliser la mémoire principale comme intermédiaire.
D'un point de vue technique, la mise en œuvre d'un tel modèle est proche de celle d'un protocole de cohérence de type MSI. Chaque donnée est ainsi associée à un vecteur qui décrit son état sur chacun des nœuds mémoires. Trois états sont possibles, l'état *invalide* (I) indique qu'il n'y a pas de copie valide de la donnée sur le nœud, l'état *modifié* (M) indique quant à lui que la donnée est disponible sur ce nœud uniquement, et l'état *partagé* (S, pour *shared*) signifie qu'il existe plusieurs copies valides, dont une sur le nœud. Comme le montre la Figure 4, l'automate qui décrit le protocole MSI indique également quels transferts de données effectuer. L'approche de StarPU est générique puisque qu'une donnée peut tout autant représenter une matrice creuse qu'une simple image, il suffit de savoir dans quel état se trouve la donnée sur chaque nœud, et de disposer des méthodes pour effectuer les transferts de données entre les différents nœuds. StarPU propose donc différentes **interfaces mémoire** afin d'accéder aux données directement à l'aide des structures de données propres aux différentes classes d'algorithmes.

4.2. Mécanismes de mémoire virtuelle

Les ressources mémoire étant potentiellement limitées (*e.g.* 256 Ko par SPU sur un Cell), il faut donc s'assurer que lorsqu'une tâche doit s'exécuter, ses données puissent être placées dans la mémoire locale. Le principe même de la DSM est de mettre en œuvre un cache logiciel, mais il ne faut pas qu'une tâche ne soit pas exécutable parce que les données mises en cache occupent déjà trop de place. StarPU dispose donc d'un mécanisme pour libérer de la mémoire en évinçant des données du cache, tout en assurant leur cohérence et leur disponibilité ultérieure. Cela permet donc de traiter des problèmes dont la taille totale dépasse celle des accélérateurs, dès lors que la granularité de chacune des tâches le permet.

4.3. Données hiérarchiques

En général, les noyaux de calcul ne traitent pas l'ensemble des données du problème : il serait bien peu efficace de transférer une énorme matrice alors que l'on accède seulement à quelques éléments. StarPU propose donc une interface pour ne manipuler qu'une sous-partie d'une donnée (par exemple un sous-bloc dans une matrice dense). Cette interface repose sur la notion de **filtre** : un filtre est une fonction qui partitionne une donnée en sous-parties disjointes. Il est par ailleurs possible de combiner différents filtres de manière récursive, afin de représenter la structure d'une donnée sous une forme hiérarchique.
L'utilisation explicite de tels filtres résulte d'un constat simple : ces filtres ont en général une réelle signification d'un point de vue algorithmique, donc puisque le programmeur connaît *a priori* l'organisation de données, autant le laisser exprimer ce qu'il sait plutôt que de laisser StarPU utiliser un mécanisme vraisemblablement peu efficace dans le cas général (par exemple une partition des données en segments de taille fixe). Ce mécanisme est similaire à la notion de distribution de données dans HPF, même si notre approche est beaucoup plus flexible et générique puisqu'on ne manipule pas ex-

clusivement des vecteurs ou des matrices denses. De la même manière que STARPU propose différentes interfaces mémoire, divers filtres sont disponibles par défaut, et écrire un nouveau filtre demande très peu d'efforts.

5. Ordonnancement

Face à la complexité des applications et des architectures émergentes, il devient de plus en plus difficile de distribuer les tâches de manière statique. Plus qu'un simple problème d'équilibrage de charge, l'hétérogénéité nous conduit donc à considérer les techniques d'ordonnancement pour tenir compte des spécificités des différentes ressources de calcul, comme nous l'avons montré précédemment [3].

5.1. Ordonnancement dans un contexte hétérogène

Les transferts de données ont un impact très important sur les performances, une politique d'ordonnancement favorisant la localité peut donc accroître les bénéfices liés à l'utilisation d'un cache de données logiciel grâce à un meilleur taux de réutilisation des données. Étant donné que plusieurs problèmes peuvent être résolu simultanément, et que les machines ne sont pas nécessairement dédiées (*e.g.* dans le cadre du couplage de codes), il est nécessaire de considérer un ordonnancement dynamique. Dans le cadre de plate-formes hétérogènes, les performances sont très variables, aussi bien en fonction des architectures (en termes de performances brutes) que selon le type de calcul (*e.g.* un code vectoriel ou bien des accès mémoires très irréguliers). Il est donc indispensable de prendre en compte toutes les spécificités de chacune des ressources de calcul lors de l'ordonnancement.

De la même manière que la gestion des données ou le déport de calcul est rendu complexe par l'hétérogénéité, mettre en place des politiques d'ordonnancement portable est un problème qui nécessite un support exécutif adapté. STARPU offre donc une interface de haut niveau qui permet d'écrire des politiques d'ordonnancement portables en combinant des mécanismes d'ordonnancements tels que le vol de travail. La conception des politiques d'ordonnancement est ainsi totalement dissociée de l'écriture des applications. Il est donc possible de sélectionner la politique la plus appropriée au moment de l'exécution. Toutes les décisions d'ordonnancement sont typiquement prises lors de la soumission ou de la terminaison d'une tâche, mais il est aussi possible de prendre ces décisions dans d'autres circonstances (*e.g.* un cœur inoccupé) ou de manière périodique.

5.2. Écrire des politiques d'ordonnancement portables

STARPU associe à chacune des ressources de calcul une file de tâches dans laquelle aller piocher (voire soumettre) du travail. En pratique, cette file est une structure de donnée abstraite qu'il est possible d'implémenter de la manière la plus appropriée à la stratégie (*e.g.* plusieurs files de priorités différentes). Une politique d'ordonnancement peut également choisir d'organiser les files selon une topologie particulière (*e.g.* une file partagée par tout le monde ou une file par processeur). Écrire une politique d'ordonnancement consiste à choisir la structure des files et la façon de les organiser, et à écrire les méthodes appelées lorsque l'on soumet ou que l'on récupère une tâche depuis l'une de ces files.

Différentes politiques d'ordonnancement sont disponibles par défaut dans STARPU. Selon les applications, on pourra par exemple choisir une stratégie basée sur le vol de travail, ou encore sur des niveaux de priorité. Afin de prendre en compte les spécificités des différentes ressources de calcul, il est également possible d'utiliser des stratégies à base de modèles de coût, que ce soit pour modéliser les performances brutes de chaque ressource de calcul (*i.e.* en GFLOP/S) ou pour prévoir le temps d'exécution des différentes tâches selon l'architecture.

6. Évaluation : portabilité des performances

Dans cette section, nous évaluons les performances de STARPU sur différentes plate-formes hétérogènes afin de mettre en évidence la portabilité de notre approche ainsi que l'impact de l'ordonnancement.

6.1. Environnement d'expérimentation

Les expériences présentées dans ce papier ont été réalisées sous LINUX 2.6, d'une part sur un environnement hybride CPUs-GPU, et sur une PLAYSTATION 3 d'autre part. La plate-forme hybride CPUs-GPU consiste en un quad-cœur E5410 XEON cadencé à 2.33 GHZ et disposant de 4 GO de mémoire

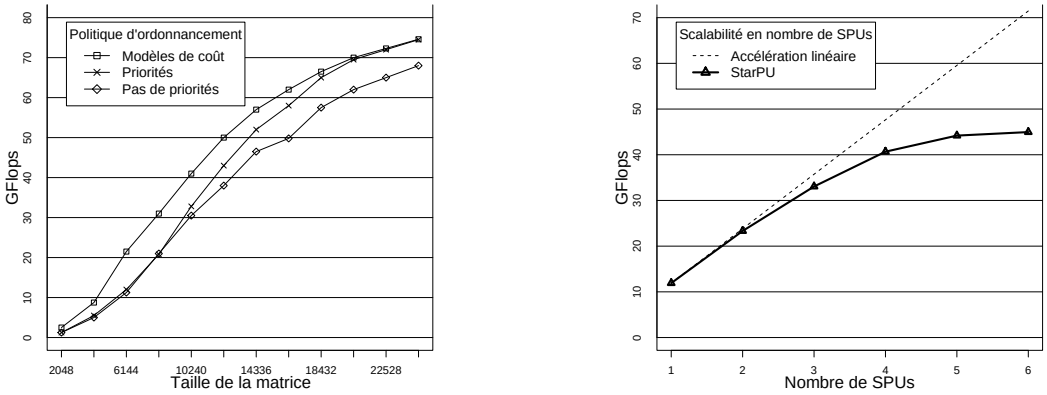

FIG. 5: Décomposition de Cholesky sur Cell (gauche) et sur CPUs/GPU (droite)

ainsi que d'une carte graphique NVIDIA QUADRO FX4600 avec 768 MO de mémoire embarquée. La PLAYSTATION 3 ne dispose que de 256 MO de mémoire, et dispose d'un processeur CELL avec seulement 6 SPUs disponibles pour les applications.

6.2. Portabilité des performances

Dans un contexte hétérogène, nous définissons l'**efficacité** comme le rapport entre la somme des capacités de calcul obtenues indépendamment sur toutes les ressources de calcul, et la capacité totale obtenue sur l'ensemble de la machine en utilisant toutes les ressources. Cette métrique permet de mesurer si STARPU réussit à tirer parti de l'intégralité de la machine.

La Table 1 montre ainsi l'efficacité obtenue sur une décomposition LU pour les politiques d'ordonnancement à base de modèles de performances présentées dans la section 5.2. Alors que l'hétérogénéité est *a priori* un frein, la stratégie qui modélise les capacités de calcul parvient à obtenir une efficacité de 99.1 %. Ce résultat est certes contrasté par la nécessité de dédier un cœur pour les accélérateurs, mais il démontre que que le faible sur-coût est plus lié à la parallélisation elle-même qu'à l'hétérogénéité.

Cette Table montre que la stratégie modélisant les temps d'exécutions de tâches, permet d'obtenir une efficacité de 101.5 %, c'est à dire une *accélération super-linéaire*. Ce phénomène s'explique par une affinité entre les tâches et les architectures : alors qu'un produit matriciel peut aller 10 fois plus vite sur un GPU que sur un cœur, une addition de matrice peut n'être accélérée que par un facteur 5. Grâce à cette politique d'ordonnancement, STARPU permet donc de distribuer les tâches là où ce sera le plus efficace, et d'éviter d'attribuer des tâches à un processeur peu adapté [3].

6.3. Portabilité

Actuellement, STARPU offre un support pour les processeurs multicœurs, les cartes graphiques et le processeur CELL comme le montre un précédent article [2]. Grâce à son interface de haut niveau, il est par exemple aisé de mettre en œuvre des algorithmes d'algèbre linéaire sur des machines hétérogènes tels qu'une factorisation de CHOLESKY comment le montre la Figure 5.

Dès lors que les *codelets* sont disponibles pour les différentes architectures, STARPU parvient donc à exploiter des machines pourtant très différentes, que ce soit au niveau du nombre de ressources de calcul, de la granularité des tâches, ou encore au niveau du paradigme de programmation sous-jacent puisque le CELL nécessite une approche totalement asynchrone alors que les cartes NVIDIA de cette génération sont totalement synchrones.

7. Travaux apparentés

En dépit des problèmes de portabilité que cela pose, la programmation des accélérateurs se fait typiquement par le biais d'interfaces constructeurs assez bas niveau. Plutôt que les APIs graphiques standards,

TAB. 1: Accélération super-linéaire sur une décomposition LU à l'aide de stratégies d'ordonnancement fondées sur des modèles de coûts. À gauche, on prédit la durée des tâches à partir de la quantité de calcul (en Flop) et de la vitesse brute des processeurs (en Flop/s) ; à droite, on utilise directement un modèle de coût pour chaque tâche et pour chaque architecture.

Ressource modélisée	Vitesse des processeurs			Temps d'exécution des tâches		
Vitesse mesurée en GFlop/s	3 CPUs + 1 GPU	3 CPUs	1 GPU	3 CPUs + 1 GPU	3 CPUs	1 GPU
	95.41	21.24	75.04	98.21	21.68	75.07
Efficacité	$95.41 = \mathbf{99.1\,\%}$	$(21.24 + 75.04)$		$98.21 = \mathbf{101.5\,\%}$	$(21.68 + 75.07)$	

ce sont maintenant FIRESTREAM (pour AMD) et surtout CUDA (pour NVIDIA) qui sont utilisées pour programmer les GPUs. Le plus souvent, la programmation du CELL se fait également avec la LIBSPE.
Alors qu'une grande partie des efforts consiste à écrire (ou générer) des noyaux de calcul efficaces, des initiatives telles qu'OPENCL montrent la nécessité d'une approche unifiée. Le standard OPENCL propose non seulement un paradigme de programmation unifié, mais également une interface bas niveau commune aux différentes technologies. Une attention toute particulière est également portée sur les standards bien établis tels que MPI [12] ou OPENMP [4]. STARPU pourrait également se placer en tant que support d'exécution pour les nombreux langages de programmation parallèles qui ont été créés (ou étendus) pour exploiter les accélérateurs [10, 6].
Un certain nombre de supports exécutifs ont été conçus pour exploiter le processeur CELL [5, 11, 14]. Bien que la plupart de ces approches proposent des interfaces similaires, peu d'entre elles visent réellement à exploiter simultanément l'intégralité des ressources de calcul des plate-formes hétérogènes. IBM ALF [5] et SEQUOIA [7] supportent à la fois le CELL et les processeurs multicœurs, mais cette approche ne se généralise pas encore à d'autres types d'accélérateurs. CHARM++ peut en revanche être utilisé aussi bien sur le CELL [9] que sur des GPUs [14], mais à notre connaissance, il n'existe pas encore d'évaluation des performances obtenues sur ces plate-formes.
Contrairement à STARPU qui propose une interface très flexible pour mettre en œuvre des politiques d'ordonnancement, SEQUOIA et CELLGEN [13] ne se concentrent pas particulièrement sur ces problématiques et se contentent de mécanismes d'équilibrage de charge très simples qui ne sont pas nécessairement suffisants pour des applications irrégulières sur des machines réellement hétérogènes. JIMENEZ *et al.* utilise également des modèles de coût pour ordonnancer des tâches entre un CPU et un GPU [8], mais leur approche ne peut pas être utilisée pour ordonnancer un graphe de tâches puisque les transferts de données restent à la charge du programmeur.

8. Conclusion et perspectives

Nous avons présenté STARPU, un support exécutif qui permet d'exploiter efficacement les architectures multicœurs hétérogènes. STARPU est disponible pour les architectures multicœures, le processeur CELL et les GPUs NVIDIA à l'adresse `http://starpu.gforge.inria.fr/`. Il offre un modèle d'exécution unifié, un environnement pour mettre en place des politiques d'ordonnancement portables, et une bibliothèque qui automatise les mouvements de données. Nous avons implémenté plusieurs stratégies d'ordonnancement et observé leur impact sur quelques problèmes classiques d'algèbre linéaire.
Au-delà du gain en programmabilité lié à son interface unifiée de haut niveau, nous avons montré que l'utilisation de politiques d'ordonnancement peut réduire les déséquilibres de charge tout en favorisant la localité des données. Étant donné qu'il n'existe pas de stratégie d'ordonnancement ultime, qui serait adaptée à tous les algorithmes, et que le choix de la meilleure stratégie est parfois complexe, le fait que STARPU permette de mettre en place et de sélectionner des politiques d'ordonnancement de manière totalement indépendante de l'application permet une très grande souplesse. Nous avons d'ailleurs montré qu'à l'aide d'une stratégie adaptée, il est possible de tirer parti de l'hétérogénéité des machines pour obtenir une efficacité super-linéaire.
Il est désormais crucial de proposer une approche unifiée pour exploiter les nombreuses technologies accélératrices, ainsi que les processeurs hétérogènes qui émergent : dans le cas contraire, il est peu probable que ces différentes initiatives transforment ce marché de *niche* avec des efforts très épars en

une approche réaliste sur le long terme. Bien que le standard OPENCL propose par exemple une interface bas niveau unifiée pour déporter du calcul sur des accélérateurs, notre travail a montré qu'il faut également qu'une telle interface soit suffisamment expressive afin que les programmeurs puissent guider l'exécution d'une application dont ils connaissent les caractéristiques.
Nous envisageons de porter STARPU sur d'autres architectures, notamment *via* un pilote pour les accélérateurs compatibles avec OPENCL. Certains environnements de compilation, comme HMPP [6], pourraient se reposer sur STARPU pour exécuter les tâches dont ils génèrent le code. Nous allons adapter certaines applications réelles ou bibliothèques, telles que les solveurs PASTIX et MUMPS, pour exploiter les architectures hétérogènes à l'aide de STARPU. Grâce à un environnement de haut niveau qui permettrait d'implémenter des stratégies d'ordonnancement complexes, il serait également intéressant d'utiliser STARPU comme une plate-forme qui permettrait de réduire le gouffre qui sépare encore les applications réelles des nombreux travaux théoriques dans le domaine de l'ordonnancement.

Remerciements

Ces travaux ont été en partie financés par l'Agence Nationale de la Recherche (ANR) au travers du programme COSINUS (projet PROHMPT n°ANR-08-COSI-013) et soutenus par NVIDIA dans le cadre d'un "Professor Partnership".

Bibliographie

1. C. Augonnet and R. Namyst. A unified runtime system for heterogeneous multicore architectures. In *Proceedings of the International Euro-Par Workshops 2008, HPPC'08*, LNCS, Las Palmas de Gran Canaria, Spain, August 2008. Springer.
2. C. Augonnet, S. Thibault, R. Namyst, and M. Nijhuis. Exploiting the Cell/BE architecture with the StarPU unified runtime system. In *SAMOS Workshop - International Workshop on Systems, Architectures, Modeling, and Simulation*, LNCS, Samos, Greece, July 2009.
3. C. Augonnet, T. Thibault, R. Namyst, and P.A. Wacrenier. StarPU : A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Proceedings of the 15th International Euro-Par Conference*, LNCS, Delft, The Netherlands, August 2009.
4. P. Bellens, J.M. Perez, R.M. Badia, and J. Labarta. Cellss : a programming model for the cell be architecture. In *Proceedings of Supercomputing*, page 86, New York, NY, USA, 2006. ACM.
5. C. H. Crawford, P. Henning, M. Kistler, and C. Wright. Accelerating computing with the cell broadband engine processor. In *CF '08 : Proceedings of the 2008 conference on Computing frontiers*, 2008.
6. R. Dolbeau, S. Bihan, and F. Bodin. HMPP : A hybrid multi-core parallel programming environment. Technical report, CAPS entreprise, 2007.
7. K. Fatahalian, T.J. Knight, M. Houston, M. Erez, D.R. Horn, L. Leem, J.Y. Park, M. Ren, A. Aiken, W.J. Dally, and P. Hanrahan. Sequoia : Programming the memory hierarchy. In *Proceedings of Supercomputing*, 2006.
8. V.J. Jiménez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *HiPEAC*, pages 19–33, 2009.
9. D. Kunzman, G. Zheng, E. Bohm, and L. V. Kalé. Charm++, Offload API, and the Cell Processor. In *Proceedings of the PMUP Workshop*, Seattle, WA, USA, September 2006.
10. M. D. McCool. Data-parallel programming on the Cell BE and the GPU using the Rapidmind Development Platform. *GSPx Multicore Applications Conference*, 2006.
11. M. Nijhuis, H. Bos, H. E. Bal, and C. Augonnet. Mapping and synchronizing streaming applications on cell processors. In *HiPEAC*, pages 216–230, 2009.
12. M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. MPI Microtask for programming the cell broadband engine processor. *IBM Syst. J.*, 45(1), 2006.
13. S. Schneider, J.S. Yeom, B. Rose, J. C. Linford, A. Sandu, and D. S. Nikolopoulos. A comparison of programming models for multiprocessors with explicitly managed memory hierarchies. In *PPoPP '09 Proceedings*, New York, 2008. ACM.
14. L. Wesolowski. An application programming interface for general purpose graphics processing units in an asynchronous runtime system. Master's thesis, 2008.