Programmation des architectures hétérogènes à l'aide de tâches divisibles ou modulables

Terry Cojean

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Terry Cojean**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Programmation d'architectures hétérogènes et manycore à l'aide de tâches moldables

---

Date de soutenance : 26 mars 2018

Devant la commission d'examen composée de :

Julien LANGOU	Professeur, University Colorado Denver	Rapporteur, Président
Jean-François MÉHAUT .	Professeur, Université Grenoble Alpes	Rapporteur
Laurent COLOMBET ..	Directeur de Recherche, CEA DAM	Examinateur
Thomas HERAULT	Directeur de Recherche, University of Tennessee	Examinateur
Abdou GUERMOUCHE .	Maître de Conférences, Université de Bordeaux	Directeur
Raymond NAMYST ...	Professeur, Université de Bordeaux	Directeur
Pierre-André WACRENIER	Maître de Conférences, Université de Bordeaux	Directeur

– 2017 –

Résumé Les ordinateurs équipés d'accélérateurs sont omniprésents parmi les machines de calcul haute performance. Cette évolution a entraîné des efforts de recherche pour concevoir des outils permettant de programmer facilement des applications capables d'utiliser toutes les unités de calcul de ces machines. Le support d'exécution StarPU développé dans l'équipe STORM de INRIA Bordeaux, a été conçu pour servir de cible à des compilateurs de langages parallèles et des bibliothèques spécialisées (algèbre linéaire, développements de Fourier, etc.). Pour proposer la portabilité des codes et des performances aux applications, StarPU ordonnance des graphes dynamiques de tâches de manière efficace sur l'ensemble des ressources hétérogènes de la machine. L'un des aspects les plus difficiles, lors du découpage d'une application en graphe de tâches, est de choisir la granularité de ce découpage, qui va typiquement de pair avec la taille des blocs utilisés pour partitionner les données du problème. Les granularités trop petites ne permettent pas d'exploiter efficacement les accélérateurs de type GPU, qui ont besoin de peu de tâches possédant un parallélisme interne de données massif pour « tourner à plein régime ». À l'inverse, les processeurs traditionnels exhibent souvent des performances optimales à des granularités beaucoup plus fines. Le choix du grain d'une tâche dépend non seulement du type de l'unité de calcul sur laquelle elle s'exécutera, mais il a en outre une influence sur la quantité de parallélisme disponible dans le système: trop de petites tâches risque d'inonder le système en introduisant un surcoût inutile, alors que peu de grosses tâches risque d'aboutir à un déficit de parallélisme. Actuellement, la plupart des approches pour solutionner ce problème dépendent de l'utilisation d'une granularité des tâches intermédiaire qui ne permet pas un usage optimal des ressources aussi bien du processeur que des accélérateurs.

L'objectif de cette thèse est d'appréhender ce problème de granularité en agrégeant des ressources afin de ne plus considérer de nombreuses ressources séparées mais quelques grosses ressources collaborant à l'exécution de la même tâche. Un modèle théorique existe depuis plusieurs dizaines d'années pour représenter ce procédé: les tâches parallèles. Le travail de cette thèse consiste alors en l'utilisation pratique de ce modèle via l'implantation de mécanismes de gestion de tâches parallèles dans StarPU et l'implantation et l'évaluation d'ordonnanceurs de tâches parallèles de la littérature. La validation du modèle se fait dans le cadre de l'amélioration de la programmation et de l'optimisation de l'exécution d'applications numériques au dessus de machines de calcul modernes.

Title Programmation of heterogeneous and manycore architectures using moldable tasks

Abstract Hybrid computing platforms equipped with accelerators are now commonplace in high performance computing platforms. Due to this evolu-

tion, researchers concentrated their efforts on conceiving tools aiming to ease the programmation of applications able to use all computing units of such machines. The StarPU runtime system developed in the STORM team at INRIA Bordeaux was conceived to be a target for parallel language compilers and specialized libraries (linear algebra, Fourier transforms, ...). To provide the portability of codes and performances to applications, StarPU schedules dynamic task graphs efficiently on all heterogeneous computing units of the machine. One of the most difficult aspects when expressing an application into a graph of task is to choose the granularity of the tasks, which typically goes hand in hand with the size of the blocs used to partition the problem's data. Small granularities do not allow the efficient use of accelerators such as GPUs which require a small amount of task with massive inner data-parallelism in order to obtain peak performance. Inversely, processors typically exhibit optimal performances with a big amount of tasks possessing smaller granularities. The choice of the task granularity not only depends on the type of computing units on which it will be executed, but it will in addition influence the quantity of parallelism available in the system: too many small tasks may flood the runtime system by introducing overhead, whereas too many big tasks may create a parallelism deficiency. Currently, most approaches rely on finding an intermediate granularity of tasks which does not make optimal use of both CPU and accelerator resources.

The objective of this thesis is to solve this granularity problem by aggregating resources in order to view them not as many small resources but fewer larger ones collaborating to the execution of the same task. One theoretical machine and scheduling model allowing to represent this process exists since several decades: the parallel tasks model. The main contributions of this thesis are to make practical use of this model by implementing a parallel task mechanism inside StarPU and to implement and study parallel task schedulers of the literature. The validation of the model is made by improving the programmation and optimizing the execution of numerical applications on top of modern computing machines.

Keywords Multicore, accelerator, GPU, heterogeneous computing, Intel Xeon-Phi KNL, task DAG, runtime system, dense linear algebra, Cholesky factorization, High Performance Computing

Mots-clés multi-cœur, accélérateur, GPU, machines hétérogènes, Intel Xeon-Phi KNL, graphe de tâches, support d'exécution, algèbre linéaire dense, factorisation de Cholesky, Calcul Haute Performance

Laboratoire d'accueil INRIA Bordeaux, 200 Avenue de la Vieille Tour, 33400 Talence

Remerciements

Je tiens à remercier en premier lieu mes directeurs de thèse, Abdou Guermouche, Raymond Namyst et Pierre-André Wacrenier pour leur aide précieuse au long de ma thèse mais aussi pour leur soutien, éclairages et discussions diverses toujours intéressantes. J'ai sincèrement apprécié réaliser ce travail en leur compagnie ainsi que la formation qu'ils m'ont apportés du métier d'enseignant-chercheur. Je me dois de les remercier aussi pour leurs cours à l'Université de Bordeaux très formateurs ainsi que pour m'avoir proposé cette thèse, chose que je ne m'imaginais pas en quittant ma Normandie natale pour aller étudier le calcul haute performance à Bordeaux.

J'adresse tous mes remerciements aux membres de mon jury, Julien Langou et Jean-François Méhaut pour leur patience et la relecture de mon manuscrit, Thomas Herault et Laurent Colombet pour avoir accepté d'examiner ma soutenance.

J'aimerais remercier aussi tous ceux qui ont contribué à ma thèse ou avec qui j'ai eu l'occasion de collaborer ou de m'en inspirer : Andra Hugo pour m'avoir encadré en master et aidé sur mon rapport, sur StarPU ainsi que pour sa patience à toute épreuve ; Jean-Marie Couteyen qui mérite une mention spéciale pour avoir été le premier à utiliser mes travaux et pour ses contributions ; Suraj Kumar, Lionel Eyraud-Dubois et Olivier Beaumont pour les nombreuses interactions sur l'ordonnancement ; Emmanuel Agullo, Mathieu Faverge et Florent Pruvost pour leur soutien autour de Chameleon et conseils en algèbre linéaire ; Samuel Thibault, Nathalie Furmento, Jérôme Clet-Ortega et Olivier Aumage pour les discussions et leur aide autour de StarPU ; Luka Stanisic pour son aide importante en représentation et analyse des performances ; Cédric Augonnet pour les quelques discussions fort fructueuses autour des supports d'exécution et des applications HPC.

Je souhaite aussi adresser mes sincères remerciements à l'équipe pédagogique de l'IUT, en particulier Pierre Ramet, Isabelle Dutour, Patrick Felix et Olivier Gauwin. Merci aussi à l'équipe de l'Université de Bordeaux avec qui j'ai pu travailler pendant la durée de mon A.T.E.R. : François Pelligrini, Aurélien Esnard, Patxi Laborde Zubieta, toute l'équipe d'initiation à l'informatique et merci encore une fois à Pierre-André, Raymond et Abdou pour leur aide durant cette période.

Un grand merci à Marie-Christine Counilh ma cobureau pour sa sympa-

thie et ses conseils avisés ainsi qu'à Denis Barthou pour son leadership, ses cours d'architecture à l'ENSEIRB et pour sa sympathie. Merci à mes anciens cobureaux de l'open-space STORM, Enguerrand P., Samuel P., Marc S., Christopher H., Adèle V., Nicolas D., Pierre H., Paul-Antoine A. et François T. pour la bonne ambiance et les soirées. Merci encore à toute l'équipe STORM ainsi qu'à HiePACS, TaDAAM et toutes les personnes sus-citées pour leur sympathie et leur accueil formidable à Inria Bordeaux et à l'Université. Mention spéciale à tous les babyfooteux et à ceux que j'ai dû oublier par manque d'exhaustivité !

Et enfin, mes derniers remerciements vont à ma compagne Kristina, à ma belle famille ainsi qu'à mes parents et grands parents pour leurs conseils avisés, soutien et bonne humeur appréciable au cours de cette période.

Table des matières

Chapitre 1

Introduction

Pendant plusieurs décennies, la performance des CPUs a augmenté principalement grâce à l'augmentation de leur vitesse d'horloge, propulsé par l'ajout de plus de parallélisme d'instruction et la mise en place d'une hiérarchie mémoire à plusieurs niveaux, notamment de caches. L'arrivée du multicœur au début des années 2000 a mis fin à l'accroissement automatique des performances des applications à chaque génération de processeur : il faut désormais exploiter de multiples cœurs au sein de chaque nœud pour exploiter pleinement le matériel. Cette tendance à l'augmentation du nombre de cœurs de calculs a largement été anticipée par les concepteurs de GPUs, des machines possédant un très grand nombre de petites unités de calcul. Originellement prévues pour le graphisme, ces unités de calcul peuvent être utilisées pour le calcul haute performance grâce à des technologies dédiées à leur programmation telles que CUDA, OpenCL ou ATI Stream. Pour répondre à cette concurrence des cartes graphiques, les constructeurs de processeurs ont eux aussi créé des cartes accélérateurs puis des processeurs manycore avec une complexité plus grande que les simples multicœurs. Ceci a donné lieu à des machines telles que que l'Intel Xeon Phi Knights Corner (KNC), l'Intel Xeon Phi Knights Landing (KNL) qui possède jusqu'à 72 cœurs et s'utilise comme un processeur ou encore la carte PEZY-SC2[1] qui est un microprocesseur équipé de 2048 cœurs. Pour les programmeurs, cette évolution rend le développement d'applications plus complexe : il doit désormais combiner l'utilisation de nombreuses bibliothèques de programmation parallèles, tel que MPI pour les machines distribuées, OpenMP ou TBB pour les multicœurs et manycore et CUDA ou OpenCL pour les cartes graphiques.

La complexité de développer des applications sur des machines hétérogènes fait qu'il est impossible, pour de nombreuses applications, de planifier a priori un découpage statique du travail. D'une part il est difficile d'estimer à quelle vitesse chaque tâche de calcul peut s'exécuter sur les différentes unités de calcul, mais il est plus difficile encore de prévoir quels seront les coûts induits

1. https://en.wikichip.org/wiki/pezy/pezy-scx/pezy-sc2

par les transferts mémoire explicites. C'est pourquoi des supports d'exécution dynamiques sont utilisés, permettant de reporter ces décisions de placement à l'exécution et capables de corriger ces décisions en fonction du comportement observé. De nombreux travaux récents montrent que les supports d'exécution dynamiques apportent non seulement une portabilité des performances supérieure aux approches statiques, mais permettent également des gains de performance dans de nombreux cas. En plus de l'ordonnancement dynamique, les supports d'exécution permettent de simplifier la programmation d'applications en proposant un modèle de programmation abstrait, souvent à base de graphe de tâches de calcul, et en organisant l'interaction avec les différentes unités de calcul grâce à des drivers.

Cependant, avec l'écart de performance croissant observé entre une carte graphique contemporaine et un cœur élémentaire, la répartition du travail sur les différentes unités de calcul reste très difficile. En particulier, de nombreux algorithmes parallèle s'expriment naturellement en terme de décomposition de domaines en sous-domaines de taille égale, comme les blocs 2D ou tuiles d'une matrice dans des bibliothèques comme PLASMA. Le choix de la taille des sous-domaines, c'est-à-dire la granularité des calculs, est un problème complexe car les unités de calcul telles que les GPUs nécessitent une granularité importante alors que les processeurs ont besoin de nombreuses tâches de faible granularité. Habituellement, la technique utilisée pour régler ce problème est de trouver grâce à une phase manuelle de calibrage de paramètres une granularité intermédiaire ni trop importante ni trop faible. Cette technique dispose de plusieurs limites, d'une part il n'est pas forcément nécessaire de générer systématiquement un si grand nombre de tâches, mais de plus chaque erreur d'ordonnancement de tâches trop grosse sur une ressource lente tel qu'un coœur de calcul sont coûteuses. Plusieurs solutions à ce problème sont identifiables. La solution la plus évidente consiste au découpage dynamique des tâches afin d'adapter le calcul à la machine considérée, cependant ce genre de technique nécessite des noyaux de calculs récursifs et est complexe à mettre en œuvre. Une seconde solution est d'utiliser une granularité de tâche fine et d'exécuter plusieurs tâches simultanément sur les GPUs grâce à l'utilisation de la technologie "multi-stream", avec comme limite principale que les temps d'exécution des calculs deviennent complètement imprévisibles.

Cette thèse financée par l'ANR Solhar [2] apporte une contribution originale au problème de l'adaptation dynamique de granularité, en proposant de reconfigurer virtuellement l'architecture sous-jacente pour adapter la machine aux tâches à exécuter. En regroupant des ressources de calcul, il est possible de rendre la machine plus homogène et de rendre le programmeur d'application plus libre du choix du grain de calcul. Pour cela, on propose la composition de bibliothèques de calcul et de supports d'exécution afin de mettre en œuvre des

2. solveurs pour architecture hétérogènes utilisant des supports d'exécution (ANR-13-MONU-0007) http://solhar.gforge.inria.fr/doku.php

coopérations de technologies et bibliothèques permettant l'utilisation efficace des ressources de calcul.

Le Chapitre 2 propose un état de l'art ainsi qu'une présentation plus détaillée des problèmes de granularité et de composition de code abordés dans cette thèse.

Le Chapitre 3 propose l'utilisation d'un nouveau modèles de tâches, les tâches parallèles rigides pour organiser la reconfiguration des ressources de calcul de la machine. Ce modèle est implanté dans le support d'exécution StarPU, et son efficacité est démontrée dans ce même chapitre sur le cas précis de la factorisation de Cholesky.

Le Chapitre 4 présente deux applications dont la programmation et la portabilité de performances est facilitée par l'utilisation de tâches parallèles rigides, le code de mécanique des fluides FLUSEPA (adaptation du code et expériences réalisées par J.M. Couteyen au sein de AIRBUS Defense & Space) et en algèbre linéaire la décomposition $PA = LU$.

Le Chapitre 5 propose l'utilisation de tâches parallèles moldables grâce à la création d'une plateforme de création d'ordonnanceurs et de gestion de tâches parallèles moldables pour exploiter le dynamisme inhérent au modèle de tâches parallèles.

Chapitre 2

État de l'art et problématiques

Sommaire

Ce chapitre fournit un état de l'art des supports d'exécution, outils permettant de faciliter la portabilité des codes et des performances en se basant sur un modèle de programmation parallèle et facilitant la répartition dynamique du calcul d'une application sur plusieurs unités de calcul grâce à l'utilisation d'ordonnanceurs. Ensuite, un état de l'art de ces bibliothèques applicatives est proposé et notamment d'algèbre linéaire dont l'évolution à travers le temps représente parfaitement l'évolution des architectures des machines de calcul. Un état de l'art des algorithmes d'ordonnancement, utiles aux supports d'exécutions et bibliothèques applicatives implantées au dessus de ceux-ci, est proposé. Ceci est illustré plus en détail en présentant le support d'exécution StarPU, son architecture et ses ordonnanceurs. Enfin, nous discutons des problèmes de gestion de la granularité et de composition d'applications avant d'exposer l'approche conduite dans cette thèse pour solutionner ces problèmes.

2.1 État de l'art

2.1.1 Supports d'exécution

Comme présenté en introduction, une multitude de technologies est nécessaire à l'utilisation des plateformes de calcul haute performance : MPI pour les machines à mémoire distribuée, du parallélisme à mémoire partagé à l'intérieur des machines et CUDA le plus souvent pour les accélérateurs. Les supports d'exécution permettent d'abstraire l'utilisation de ces outils et de concevoir des bibliothèques haute performance portables sans préoccupation bas-niveau pour le programmeur d'applications. Les supports d'exécutions proposent aussi la gestion des données et l'ordonnancement des calculs, ce qui les rend fortement intéressants pour les plateformes équipées d'accélérateurs. On distingue deux familles principales de supports d'exécution, les supports d'exécution génériques et les supports d'exécution spécifiques à un domaine d'application.

Supports d'exécution génériques La plupart des supports d'exécutions représentent les applications grâce à un modèle de graphe orienté acyclique de tâches de calcul où les arêtes représentent les dépendances entre les tâches. D'autres modèles de programmation courants sont les modèles de type "Fork-Join" où une portion de l'application est parallélisée (souvent une boucle for dont les indices sont découpés) puis une synchronisation est assurée par un thread principal "maître". Outre le modèle de programmation, plusieurs autres points différencient ces supports d'exécutions, principalement selon la gestion des mouvements de données entre unités de calcul et l'importance donnée à l'ordonnancement, mais on trouve aussi d'autres différences comme l'existence d'outils d'analyses de performance, la possibilité d'utiliser le support d'exécution en mode distribué et dans ce cas la tolérance aux pannes.

Pour les machines à mémoire partagée et sans accélérateurs, plusieurs supports d'exécutions sont disponibles comme la bibliothèque Intel TBB [100], le langage Cilk [55], les extensions d'OpenMP [19] et la STL de C++11. La "Standard Template Library" de C++11 propose notamment l'utilisation de mots clés async et des objets futures ce qui permet de créer des programmes asynchrones multithreadés. Cilk et TBB proposent un ordonnancement de tâches à base de vol de travail. OpenMP quand à lui propose plusieurs types d'ordonnancement intégrés au langage et a récemment étendu son support de création de tâches et de dépendances ainsi que l'offload de code sur les accélérateurs.

De nombreux supports d'exécutions ont été créés pour l'exécution d'applications sur des plateformes équipées d'accélérateurs. Qilin [87] propose une interface pour soumettre des noyaux qui s'exécutent sur des tableaux automatiquement répartis entre les différentes unités de calcul d'une machine hétérogène. Pour cela, Qilin compile dynamiquement du code pour les CPUs (en reposant sur Intel TBB [100]) et pour les GPUs en utilisant CUDA. Charm++ [74]

est une extension du langage C++ qui propose une exécution distribuée du code et a aussi été étendu pour l'utilisation de GPUs [78]. Le support d'exécution SGPU 2 [92] se concentre sur les plateformes composées de plusieurs machines hétérogènes équipées d'accélérateurs. SGPU 2 ajoute au modèle classique de création de processus légers pour l'exécution multithreadée sur CPU le concept de processus légers accélérateurs dont l'interface est similaire aux processus légers POSIX et qui peuvent réaliser des transferts de données. Le supports d'exécution ACP+ [65] dispose d'une gestion de données basée sur un mécanisme de type DSM (Distributed Shared Memory) : chaque bloc de donnée est associé à un bitmap pour déterminer s'il existe déjà une copie disponible localement de cette donnée sur chaque unité de calcul.

OpenACC [91] est un standard pour le calcul parallèle développé par plusieurs constructeurs (*i.e.* Cray, CAPS, Nvidia et PGI) qui cible les machines de calcul hétérogènes, surtout CPU/GPU. Tout comme OpenMP, ce standard est basé sur une annotation de code à base de pragmas. Le standard OpenACC a été en parti intégré au standard OpenMP, ce qui explique l'ajout de supports d'offload de code dans OpenMP 4.0 notamment. Depuis la norme 4.5 de novembre 2015, le support d'exécution OpenMP [19] traditionnellement un support d'exécution destiné pour les machines multicœurs peut aussi être utilisé pour l'utilisation efficace d'accélérateurs. En effet, la norme 4.5 introduit notamment la possibilité d'envoi asynchrone de travail sur les accélérateurs le tout combiné avec l'utilisation des tâches et dépendances introduites précédemment, ce qui n'était pas possible avant.

Bien que beaucoup de supports d'exécutions sont basés sur des graphes de tâches, ceux-ci ont, pour origine, des conceptions très différentes du modèle de graphe de tâches ainsi que de l'ordonnancement. XAAPI/XKAAPI [67] est un support d'exécution centré sur le vol de tâches et proposait à l'origine une évaluation paresseuse des dépendances, notamment au moment du vol. StarPU [20] à l'origine était conçu pour être un support d'exécution proposant des interfaces génériques pour la création d'ordonnanceurs, notamment basés sur l'utilisation d'un modèle de performance adaptatif pour les tâches. StarPU propose aussi l'utilisation de "sequential task flow" qui signifie qu'il n'y a qu'un seul flot unique de tâches soumises au support d'exécution par l'unique processus léger applicatif ce qui a l'avantage de simplifier la programmation pour l'utilisateur. StarSs [22] quand à lui souhaite à l'origine surtout se concentrer sur le fait de fournir une interface de type OpenMP et proposer une programmation du modèle de tâches à l'aide de pragmas. DAGuE/PaRSEC [27, 28] utilise le modèle de graphes de tâches paramétrés [72], une représentation symbolique qui décrit les tâches par leurs dépendances définies explicitement par le programmeur d'application, ce qui réduit l'empreinte mémoire du graphe de tâche ainsi que plusieurs surcoûts liés à son exécution. Enfin, le support d'exécution Legion [23] (logical regions) décide de se concentrer sur les données. La méthode principale pour instancier le graphe de tâches est de définir une par-

tition des données avec un modèle de description qui permet de partitionner aussi bien des vecteurs, des matrices, que des graphes ou des stencils complexes de façon fine comme grossière. Des tâches peuvent ensuite être soumises sur ces partitions, à différentes échelles en même temps si nécessaire, et Legion assure la cohérence des calculs sur ces partitions. À l'exception de Legion dont l'ordonnancement est basé sur du vol de travail, tous ces supports d'exécution proposent plusieurs stratégies d'ordonnancement, ainsi que des interfaces pour implanter ses stratégies.

Cependant de nos jours, les supports d'exécution semblent fournir certaines fonctionnalités similaires, notamment au niveau des modèles de programmation et d'exécution. En effet, aussi bien PaRSEC, OmpSs, XKAAPI et bien sûr StarPU proposent désormais aux programmeurs d'applications d'utiliser le modèle de tâches "sequential task flow" (STF) au dessus de leurs supports d'exécution. Cependant, PaRSEC continue de proposer le modèle de tâches paramétrée, plus descriptif et plus efficace, en plus de STF. De plus, StarPU, XKAAPI et bien sûr StarSs disposent aujourd'hui d'une interface OpenMP. Cette interface pour les deux premiers supports d'exécution est possible grâce au compilateur KLANG [11]. Enfin, StarSs, XKAAPI et bien sûr StarPU proposent aujourd'hui des ordonnanceurs basés sur une analyse dynamique des performances des noyaux et notamment des ordonnanceurs de type HEFT. Autrement dit, aussi bien du point de vue de l'interface que de l'ordonnancement et l'aspect autotuning, les supports d'exécutions génériques proposent maintenant un sous ensemble de fonctionnalités similaires.

Supports d'exécutions spécifiques Plusieurs supports d'exécution spécifiques à certains domaines applicatifs ont été créés. Le support d'exécution TBLAS [107] est dédié à l'algèbre linéaire. Il automatise les transferts de données et propose une interface simple pour créer des applications d'algèbre linéaire. Dans TBLAS, il est nécessaire de projeter statiquement les données sur les différentes unités de calcul, mais il supporte des tailles de blocs hétérogènes (*i.e.* différents grain de calcul). Le support d'exécution QUARK [79] a été créé spécifiquement pour l'ordonnancement de noyaux d'algèbre linéaire sur des machines multi-cœur. C'était le support d'exécution de la bibliothèque PLASMA jusqu'au récent port de PLASMA au dessus d'OpenMP qui dispose désormais d'un modèle de graphe de tâche plus fourni. QUARK était caractérisé par son ordonnancement à base de vol de travail et sa meilleur scalabilité que les autres supports d'exécution dédiés notamment grâce à sa simplicité. Le support d'exécution SuperMatrix [35] est assez similaire, la matrice est vue de façon hiérarchique : la matrice est vue comme des blocs qui servent comme unité de données sur lesquels des calculs sont exécutés. SuperMatrix enfile les opérations nécessaires de façon transparente en respectant les dépendances de façon interne, et exécute ces opérations en utilisant des techniques de type "out-of-order".

Dans le domaine de l'AMR, un support d'exécution nommé Perilla [90] se base sur les métadonnées des logiciels d'AMR pour permettre plusieurs optimisations aussi bien sur la découverte des dépendances que pour la localité des données et l'utilisation efficace des nœuds NUMA. Perilla propose l'utilisation d'un système de tâches hiérarchiques basé sur la structure des données AMR qui sont découpées à plusieurs échelles, aussi bien du fait du raffinement de maillage adaptif que pour optimiser la réutilisation du cache. Perilla autorise l'exécution sur des machines distribuées grâce à MPI et propose notamment un système d'ordonnancement adapté aux nœuds NUMA.

La bibliothèque STAPL [99] se défini comme une extension de la STL du C++. STAPL a reçu plusieurs évolutions au cours des années et propose notamment une bibliothèque parallèle de graphe SGL [64] qui abstrait la description du parallélisme et la distribution des données pour faciliter le développement d'algorithmes sur de grands graphes. Ce support d'exécution facilite l'expression de calculs à gros grain, automatise l'équilibrage de charge et simplifie certaines optimisations liées à la localité des données.

Composition de code Un problème important subsiste malgré l'utilisation de supports d'exécution qui est central dans cette thèse : la composition de code. En effet, exécuter plusieurs bibliothèques parallèles simultanément est difficile car ces bibliothèques ne sont pas informées de l'utilisation des ressources de chacun, donnant lieu à une surexploitation des ressources. Cette problématique n'est cependant pas uniquement liée aux supports d'exécution et ce problème a déjà été observé sur des codes de type OpenMP + MPI [37], et même OpenMP seul.

Ceci a donné lieu au framework Lithe [93], une interface de gestion du partage de ressources qui défini comment les processus légers sont transférés entre les bibliothèques parallèles à l'intérieur d'une application. Un problème de cette contribution est qu'elle n'autorise pas le changement dynamique le nombre de ressources assignées à une bibliothèque.

Certains supports d'exécution ont reçus des contributions pour permettre la composition de codes. Un premier support d'exécution, SGPU 2, peut partager des ressources telles que les GPUs grâce à l'utilisation du concept de processus léger accélérateur et à l'utilisation d'un ordonnanceur global qui réparti ces processus légers sur les ressources. C'est le cas notamment de StarPU où une contribution dans ce domaine se trouve dans la thèse d'Andra Hugo [69] qui propose des contextes d'ordonnancement pour régler ce problème ainsi que l'utilisation d'un hyperviseur pour répartir dynamiquement les ressources entre les différents flux de tâches applicatifs. Un premier pas pour l'interoperabilité des supports d'exécution est mené par le projet européen INTERTWinE [95] qui facilite la composition de codes. Récemment, une approche basée sur l'utilisation de composants logiciels pour composer les bibliothèques parallèles notamment en tirant parti des approches d'ordonnancement de tâches a été por-

tée dans le cadre de la thèse de Jérôme Richard [101].

Une autre approche est de réaliser cette composition en partie à travers l'utilisation d'un ordonnanceur comme l'ordonnanceur Rinnegan [94] implanté dans StarPU. Un module noyau collecte des statistiques sur l'utilisation des ressources de la machine et ces informations sont utilisées pour proposer aux applications un placement des tâches qui optimise leur exécution en fonction de plusieurs critères : l'obtention d'une faible latence, un débit maximal ou le respect de dates limites en temps réel. L'ordonnanceur Rinnegan respecte les décisions de ce framework pour l'allocation de tâches et accélère ainsi les calculs dans le cadre de l'exécution de plusieurs applications simultanément.

2.1.2 Programmation d'applications et algèbre linéaire

L'évolution des bibliothèques d'algèbre linéaire calque parfaitement l'évolution des architectures des machines et met en avant les problèmes de parallélisme centraux dans cette thèse. Nous verrons ici comment ces bibliothèques permettent la réutilisation des codes et des données grâce à la définition de standards, quelles réponses ont été apportées aux architectures émergentes puis l'émergence des supports d'exécutions facilitant le travail de la communauté applicative.

La première bibliothèque de solveurs d'algèbre linéaire dense LINPACK [47] a proposé dans les années 1970 une implantation fortran basés sur des accès élément par élément facilitant une pseudo-vectorisation des factorisations de Cholesky, QR ou LU, entre autres. Par la suite, LAPACK [18] fut développée dans les années 1980 notamment pour tirer parti des machines de calcul à mémoire partagée avec des caches. L'originalité de LAPACK est de représenter les calculs d'algèbre linéaire comme un assemblage d'opérations élémentaires d'algèbre linéaire sur des sous-matrices, telle que la multiplication de matrices, des routines de calcul définis par le standard BLAS [82]. L'existence de ces standards permettent l'émergence de nombreuses bibliothèques par la suite, notamment les vendeurs de matériels tels qu'Intel fournissent depuis 2003 leur propre implantation des routines BLAS optimisées pour leurs machines, la bibliothèque MKL [40]. Afin d'exploiter efficacement les machines équipées de systèmes de caches émergentes dans les années 80 et de la nécessité de réutiliser les données, les algorithmes d'algèbre linéaire font ensuite usage de systèmes de tuiles. Ce phénomène est ensuite amplifié par l'émergence d'un nouveau standard dans les années 90, le standard MPI [106], ainsi que l'apparition des machines à mémoire distribuée qui motivent la création de la bibliothèque ScaLAPACK [24]. La bibliothèque ScaLAPACK représente les algorithmes d'algèbre linéaire dense comme un assemblage de calculs simples sur des blocs de données et répartie les régions des matrices entre processus, soit par groupe de colonnes soit par blocs rectangulaires.

Une nouvelle évolution dans le début des années 2000 va fortement changer

les algorithmes d'algèbre linéaire et le parallélisme sur les machines de calcul haute performance, les machines multicœurs avec jusqu'à nos jours un nombre de cœurs à l'intérieur des processeurs constamment en augmentation jusqu'à obtenir 72 cœurs de nos jours sur l'Intel Xeon Phi Knights Landing. Ce type de machine va faire évoluer les algorithmes d'algèbre linéaire dense et favoriser l'utilisation de tuiles [61]. Cette nouvelle représentation des données favorise la réduction de la granularité des calculs et ainsi favoriser l'émergence de parallélisme tout en optimisant la réutilisation des données dans les caches et l'utilisation d'unités de calcul vectorielles. De plus, cette évolution a été accompagnée par l'émergence d'un nouveau modèle de représentation des calculs d'algèbre linéaire basés sur un graphe orienté acyclique de tâches de calcul dont les arêtes représentent les dépendances entre les calculs [33]. Ce type de parallélisme réduit les synchronisations au cours du calcul de l'algorithme contrairement aux approches présentées précédemment de LAPACK ou ScaLAPACK. À partir de ce modèle, plusieurs nouveaux algorithmes d'algèbre linéaire dense sont apparus [31, 96] et ont constitué la base de plusieurs logiciels connus comme PLASMA [7] et FLAME [115].

Les machines de calcul haute performance étant composées de nombreuses machines multicœurs, il existe plusieurs types de communications et d'envoi de données soit entre les machines soit à l'intérieur de celles-ci entre les cœurs de calculs. Ceci impose une approche hybride de parallélisation à deux niveaux pour reproduire au sein des calculs de solveurs linéaires cette hiérarchie des machines. Cette approche est notamment utilisée dans les solveurs d'algèbre linéaire [52, 59, 60, 102] en mélangeant la programmation multithreadée avec les passages de message (MPI) pour la gestion des communications et du parallélisme à mémoire distribué.

À cause de leur architecture très hiérarchique, les machines multicœurs ont des accès non uniformes à la mémoire. Ainsi, si les données ne sont pas correctement arrangés dans la mémoire et si les accès à cette mémoire ne sont pas cohérents, les applications multithreadées peuvent souffrir de larges pertes de performances. L'apparition de logiciels comme hwloc [29] permet de fournir une représentation de cette hiérarchie à l'intérieur des machines et de créer une affinité entre les ressources et le calcul. Un exemple intéressant est le solveur d'algèbre linéaire creuse PaStiX [66] qui a notamment vu son efficacité grandement améliorée en modifiant le modèle d'accès aux données par les processus légers déployés sur différents nœuds afin de prendre en compte les accès non uniformes à la mémoire (nœuds NUMA) et d'atteindre un meilleure localité des données [53].

Afin de mieux s'adapter à ces nouvelles machines, de nouveaux solveurs d'algèbre linéaire tels que `qr_mumps` [30], HSL-MA87/HSL-MA86 [68] ou SuperLU-MT [83] ont été développés en se basant sur l'utilisation de parallélisme à grain fin, une parallélisation multithreadée souvent couplée à MPI, et une gestion avancée des données pour gérer les hiérarchies mémoires.

En parallèle, une nouvelle évolution technologique l'apparition des accélérateurs (surtout des GPUs) dans les supercalculateurs et leur puissance de calcul ont soulevés de nouveaux problèmes. Alors qu'un parallélisme a grain fin est nécessaire pour l'utilisation des machines multicœurs, l'utilisation de GPUs demande une utilisation de parallélisme à gros grain et une nouvelle problématique pour la gestion des données : le transfert sur l'accélérateur. De nombreux travaux ont alors été réalisés en algèbre linéaire pour s'adapter à ce nouveau défi. D'abord pour proposer une implantation efficace de noyaux [88, 89, 56, 116] ainsi que la bibliothèque cuBLAS [41] du vendeur NVIDIA puis pour le design d'algorithmes mono-accélérateurs pour des machines hétérogènes basées sur l'offload de calcul [110], puis l'utilisation de CPUs et d'un unique accélérateur simultanément [111] et enfin des algorithmes multi-accélérateurs [97, 85, 4].

Un cas particulièrement intéressant est la bibliothèque d'algèbre linéaire MAGMA [112] qui permet le calcul sur des CPUs et plusieurs GPUs simultanément. Afin d'exploiter ce genre de machine, la bibliothèque MAGMA utilise plusieurs grains de calculs basés sur des colonnes de la matrice. Les CPUs factorisent un panneau très fin de la matrice composé de peu de colonnes et les GPUs réalisent des opérations très efficaces (comme des multiplications de matrices, le noyau BLAS3 GEMM) sur des panneaux très larges de la matrice, autant que de GPUs. On voit donc un retour vers le parallélisme à gros grain de ScaLAPACK par exemple pour les GPUs.

Afin d'aller plus loin que le découpage statique des données proposé par MAGMA, les dernières avancées sont basées sur l'utilisation d'ordonnancement dynamique afin d'obtenir une meilleure flexibilité nécessaire pour ces plateformes [5, 8]. De façon plus spécifique, des supports d'exécutions sont utilisés pour créer des bibliothèques d'algèbre linéaires dense basées sur ceux-ci, telles que `Chameleon` [3], DPLASMA [26], une adaptation GPU de FLAME [71] et récemment en algèbre linéaire creuse de nouvelles versions de `qr_mumps` [6] et de PaStiX [81]. D'autres domaines comme la FMM a aussi reçu plusieurs développement d'applications au dessus de supports d'exécutions [17, 86]. Un logiciel en particulier en algèbre linéaire dense, High Performance Linpack (HPL) qui sert utilise le calcul de la factorisation $PA = LU$ comme benchmark des plus grandes machines de calcul modernes embarque certaines fonctionnalités des supports d'exécution en interne, notamment pour la gestion du parallélisme et l'utilisation d'accélérateurs.

2.1.3 Ordonnancement

La communauté de l'ordonnancement a créé de nombreux algorithmes théoriques permettent d'organiser des calculs, avec différentes connaissances sur le travail à réaliser, au dessus de ressources. Ces travaux théoriques sont centraux dans la communauté des supports d'exécution qui s'en inspirent pour

créer des algorithmes d'ordonnancement efficaces. On distingue deux classes principales d'algorithmes d'ordonnancement. L'ordonnancement statique où tout le graphe de tâches de calcul est connu à l'avance et l'ordonnancement dynamique ou online où les tâches ne sont connues qu'à leur apparition dans le système d'ordonnancement, le plus souvent sans information sur les tâches précédentes ou suivantes.

Ordonnancement statique L'ordonnancement de graphes de tâches statiques ou encore avec précédence suppose la connaissance de la totalité du graphe de tâches d'une application ainsi que les dépendances associées.

Une catégorie importante d'ordonnanceurs sont les ordonnanceurs de liste [75, 57, 117, 80, 105] qui fonctionnent en deux temps. Dans un premier temps, ils ordonnent les tâches en attribuant des priorités à chaque tâche. Dans un second temps, selon ces priorités les tâches sont sélectionnées puis ordonnancées sur un processeur qui minimise une fonction de coût prédéterminée. Un ordonnanceur central de liste dont beaucoup d'ordonnanceurs sont inspirés est l'ordonnanceur HEFT [113] qui est destiné à l'exécution sur des plateformes hétérogènes avec plusieurs classes de ressources. L'algorithme HEFT (Heterogeneous Earliest-Finish-Time) fonctionne en deux phases, une première où le graphe est parcouru en commençant par la fin afin d'assigner des priorités à chaque tâche et identifier le chemin critique, ensuite dans une deuxième phase chaque tâche est soumise à la ressource qui minimise son temps de terminaison en prenant en compte la puissance de calcul de la ressource, sa charge ainsi que le temps de transfert des données.

Assez tôt dans l'histoire de l'ordonnancement de tâches, 1960 étant la date de publication la plus ancienne trouvée [38, 39], le problème d'ordonnancement de tâches sur des ressources pré-déterminées de type cœurs de processeurs a été étendu au cas où une tâche peut être exécutée par plusieurs ressources simultanément, créant ce qui sera appelé plus tard les graphes de tâches parallèles. Une seconde apparition des tâches parallèles sont les problèmes de 2D-packing ou encore strip-packing [51, 84] qui considèrent en fait des tâches parallèles de type "rigides". Une contribution [114] propose notamment une technique pour adapter tout ordonnanceur de tâches parallèles de type rigides à des ordonnanceurs de tâches parallèles de types moldables.

En effet, les graphes de tâches parallèles sont un moyen efficace de modéliser les applications où se mélangent un parallélisme de tâche et un parallélisme de données. Plusieurs types de tâches parallèles sont identifiés [49, 54]. Les tâches *rigides* sont des tâches parallèles dont le nombre de processeurs sur lequel chaque tâche s'exécute est prédéfini et fixe. Les tâches *moldables* sont des tâches parallèles dont le nombre de processeurs attribués à une tâche est décidé avant le début de l'exécution de la tâche, mais reste fixe après cela. Enfin, les tâches *malléables* [50] sont des tâches dont la quantité de ressource peuvent varier au cours de son exécution.

Plusieurs heuristiques telles que CPA [98], BiCPA [44], ou encore CPA13 [70] ont été crées pour l'ordonnancement de graphes de tâches parallèles sur des architectures hétérogènes avec précédence. Ce type d'algorithmes utilisent souvent une approche en deux phases, différentes des algorithmes de listes. La première phase est *l'allocation* où le nombre de ressources attribué à chaque tâche est décidé avant d'exécuter la phase *d'ordonnancement* où l'ordre d'exécution des tâches et sur quelles ressources est décidé. La deuxième phase est souvent une variation d'un algorithme de liste, notamment de HEFT.

Ordonnancement dynamique À la différence des ordonnanceurs statiques, les ordonnanceurs dynamiques (ou encore "online") doivent prendre des décisions sans connaissance complète du graphe de tâches. Il est par exemple possible que les tâches ne sont pas connues par l'ordonnanceur avant qu'elles n'arrivent, le plus souvent après que leurs prédécesseurs ont terminés leurs exécutions, ou que le temps d'exécution des tâches ne sont pas connues en avance. Dans ce contexte, les études théoriques se concentrent sur l'obtention d'une garantie des performances comparé à une stratégie non dynamique. Du fait de ces suppositions, les ratios de performance dans le pire des cas sont souvent très élevés [50, 104]. Dans certains cas, il est cependant possible de réaliser une analyse en cas moyen [62, 83] pour obtenir de meilleures bornes. Malgré cela, une étude empirique [36] a montré un avantage à l'utilisation d'ordonnanceurs dynamiques comparé aux méthodes purement statiques.

En ce qui concerne l'ordonnancement sur une machine hétérogène, très peu d'ordonnanceurs de tâches parallèles (moldables) ont été réalisés. Deux études récentes proposent des ordonnanceurs de tâches moldables dynamiques de tâches indépendantes [25, 34]. L'approche utilisée est comme pour la version statique une approche en deux phases : une première permet de décider du nombre de ressources allouées aux tâches et une seconde phase réalise l'allocation des tâches, transformées en tâches rigides à ce point de l'exécution, aux ressources. Deux ordonnanceurs sont notamment proposés dans [25] tous les deux des algorithmes de ρ-approximation qui prennent une supposition λ en entrée et retournent une solution au pire à $\rho\lambda$ de l'objectif si elle existe. Le premier algorithme est $\frac{3}{2}$-approché basé sur un algorithme d'optimisation linéaire à nombre entier. Le second algorithme est 2-approché avec une complexité polynomiale en temps.

2.2 StarPU

Les travaux de cette thèse seront réalisés à l'aide du support d'exécution StarPU. StarPU propose un cadre intéressant, en effet le support d'exécution a un modèle de tâche simple, expressif et efficace ainsi qu'un modèle d'implantation d'ordonnanceur très fourni. De plus, des travaux conduits dans le

cadre de la thèse d'Andra Hugo [69] permet d'avoir un premier retour et une première approche de l'isolation de flux applicatifs, ce qui est peu commun.

2.2.1 Architecture générale

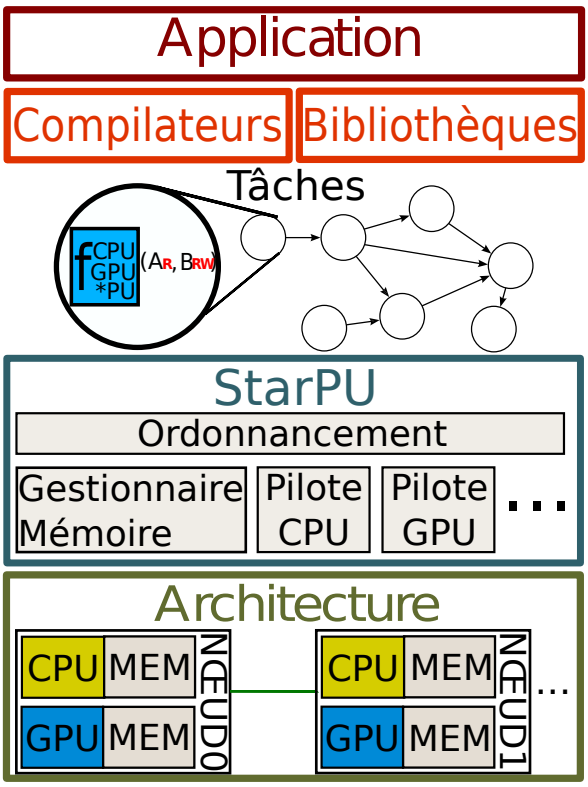


FIGURE 2.1 – Schéma d'architecture logicielle de StarPU.

La Figure 2.1 présente l'architecture de StarPU et son interaction avec les autres couches logicielles et matérielles. Une première partie de StarPU est son modèle d'expression du parallélisme et la facilité d'utilisation proposée aux utilisateurs. En effet, une application peut, soit au travers de bibliothèques ou de compilateurs soit directement, générer des tâches qu'elle soumet à StarPU. Comme montré dans le schéma, chaque tâche peut accéder des données représentées par des pointeurs virtuels appelés handle grâce à des modes d'accès : lecture, écriture. Selon l'ordre de soumission de ces tâches et les données accédées, un graphe de tâche complet est formé tel que dans l'exemple de cette figure.

En utilisant ce graphe de tâche et surtout les "handles" encapsulant les données ainsi que leur modes d'accès, StarPU propose un dispositif de gestion de données intégré notamment dans le cas d'utilisation d'une machine hétérogène. Une surcouche StarPU-MPI permet aussi de gérer les transferts de

données dans le cadre de machines à mémoire distribuée. La gestion des données de StarPU repose sur une implantation interne du protocole de gestion de cache MESI, qui sert à gérer l'état des données et, par exemple à faciliter leurs duplications sur des nœuds distants lors d'accès concurrents en lecture.

Pour l'interaction avec les architectures matérielles, StarPU se repose sur les concepts de "workers" et de drivers. Les workers sont des processus légers initialisés lors du lancement de StarPU sur chaque matériel supporté après une phase de découverte, ce processus est notamment aidé par l'outil hwloc. Les workers interagissent avec les drivers qui fournissent des fonctions, notamment de lancement de calcul et d'envoi de données, pour plusieurs architectures. Les architectures supportées par StarPU sont multiples, on trouve notamment les CPUs classiques, les accélérateurs GPUs Nvidia, les accélérateurs Intel Xeon Phi KNC, etc.

2.2.2 Ordonnancement

Un des avantages principaux de StarPU est la liberté d'implantation d'ordonnanceurs fournie par le support d'exécution. Les ordonnanceurs sont implantés dans StarPU à l'aide de deux fonctions génériques push et pop dont les ordonnanceurs implantent le comportement. D'autres fonctions sont aussi proposées au créateur d'ordonnanceur, notamment pour initialiser et détruire l'ordonnanceur et notamment ses structures de données internes, mais aussi pour exécuter des actions avant ou après l'exécution d'une tâche par exemple.

On distingue deux types d'ordonnanceurs. Ceux-ci utilisent tous les mêmes principes de push et de pop, mais ils diffèrent par leur mode de conception : les ordonnanceurs qui se cantonnent à implanter les fonctions d'interface (ici nommés classiques) et les ordonnanceurs modulaires qui sont composés de modules d'ordonnancement qui interagissent entre eux grâce à de nouvelles fonctions d'interface.

Les ordonnanceurs classiques

Afin de mieux comprendre les ordonnanceurs de StarPU et d'observer certains ordonnanceurs centraux, les ordonnanceurs eager, dmda (*i.e.* HEFT modifié) et HETEROPRIO sont présentés.

L'ordonnanceur eager La Figure 2.2 montre le schéma de l'ordonnanceur eager ainsi que le parcours des tâches au sein de StarPU. De façon générale, lorsque toutes les dépendances d'une tâche sont satisfaites, cette tâche devient prête. Ainsi, à travers une analyse de la progression des tâches exécutées, les prochaines tâches sont libérées, deviennent prêtes et sont poussées une à une grâce à la primitive "push" dans l'ordonnanceur. De l'autre côté de l'ordonnanceur, les ressources lorsqu'elles ont besoin de travail font appel à la méthode

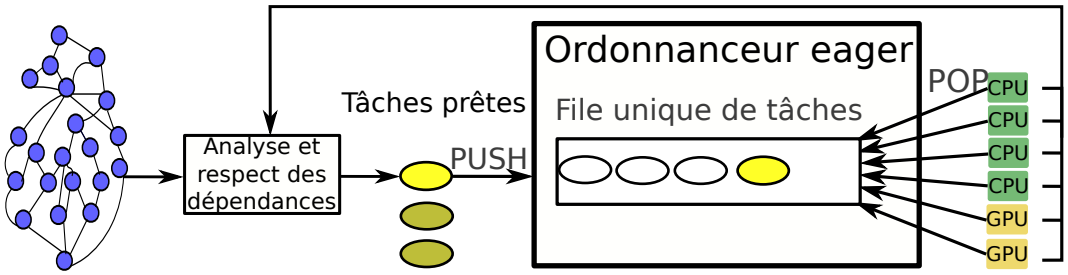


FIGURE 2.2 – Schéma de l'ordonnanceur eager de StarPU.

"pop" de l'ordonnanceur qui leur permet alors de récupérer une tâche. Une fois cette tâche exécutée, toutes les tâches dépendant de celle-ci sont marquées comme libérées et de nouvelles tâches prêtes viennent alimenter l'ordonnanceur grâce à la méthode "push", jusqu'à exécution complète du graphe de tâches.

L'ordonnanceur eager se contente d'ajouter les tâches prêtes dans une file globale, et toutes les ressources lorsqu'elles appellent la méthode pop et enlèvent une tâche de la file si elle n'est pas vide. L'intérêt de l'ordonnanceur eager est qu'un équilibrage de charge naturel peut naître entre les ressources car les ressources rapides prendront simplement plus de tâches. Cependant, plusieurs problèmes apparaissent tels que l'accès concurrent à la file de tâche, la prise de décision tardive qui empêche notamment le préchargement des données dans les mémoires distantes et la non prise en compte de la localité des données.

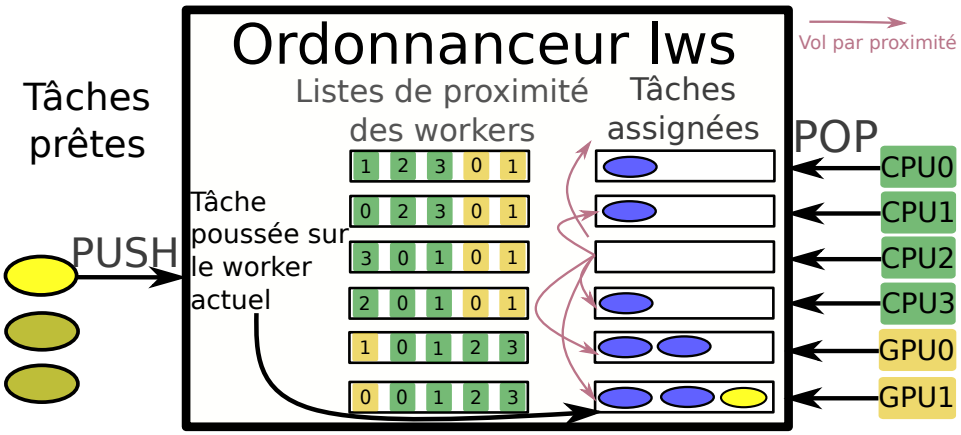


FIGURE 2.3 – Schéma de l'ordonnanceur lws de StarPU.

L'ordonnanceur "locality work-stealing" (lws) La Figure 2.3 présente le schéma de l'ordonnanceur lws implanté au sein de StarPU. Cet ordonnanceur permet un vol de travail rapide en prenant en compte la proximité des

ressources. C'est aussi l'ordonnanceur qui passe le mieux à l'échelle de StarPU. Son fonctionnement est simple : une liste de proximité des ressources est instanciée à la création des ressources. Lorsque des tâches prêtes apparaissent, elles sont poussées sur la ressource actuelle. Ensuite, lorsqu'une ressource souhaite récupérer une tâche, elle essaye d'abord dans sa liste personnelle puis elle utilise cette liste de proximité pour voler une tâche dans la file de tâches d'une des ressources les plus proches. Un mode additionnel est disponible pour forcer une localité des données permettant de pousser les tâches sur les nœuds mémoires contenant le plus de données de cette tâche.

L'ordonnanceur dmda (HEFT modifié) L'ordonnanceur HEFT est un ordonnanceur de liste statique comme présenté en Section 2.1.3. Cet ordonnanceur a été modifié et implanté dans StarPU au nom de dmda sous la forme d'un ordonnanceur dynamique. Cette implantation est présentée dans la Figure 2.4.

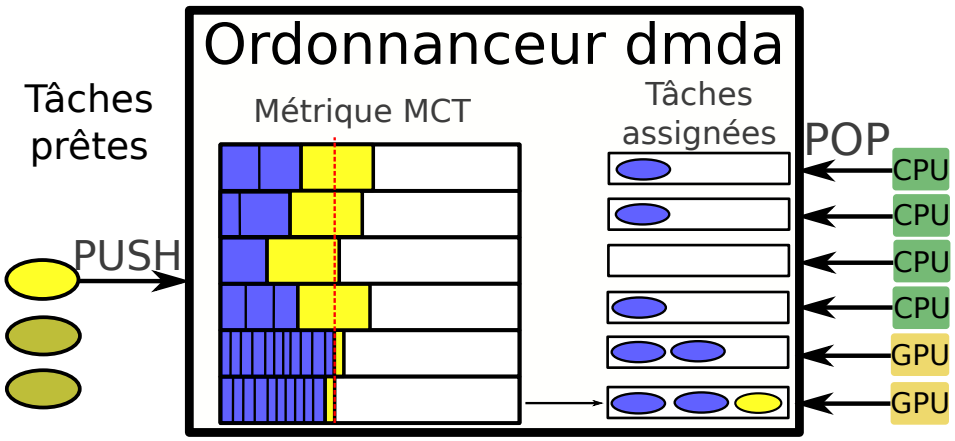


FIGURE 2.4 – Schéma de l'ordonnanceur dmda de StarPU inspiré de HEFT.

Comme présenté précédemment, seules les tâches prêtes, dont leurs dépendances sont satisfaites, peuvent être exécutées par l'ordonnanceur. Dans le cas de dmda lorsqu'une tâche arrive dans l'ordonnanceur, le critère de temps de complétion minimale de la tâche (MCT) est calculé pour chaque ressource. Cela consiste à regarder l'occupation des ressources selon les tâches déjà exécutées et les tâches assignées à chaque ressource, ceci donne une quantité totale présentée en bleu. Ensuite, pour chaque ressource le temps de complétion de la tâche est ajouté à ce temps, ainsi que tout ou partie du temps de transfert des données si celui-ci ne peut pas être complètement recouvert, ceci est représenté en jaune dans le diagramme de Gantt de la figure. Enfin, la ressource qui permet de terminer au plus tôt la terminaison de la tâche est identifié, symbolisé par la barre rouge, et la tâche est alors ajouté à la file de tâche de la ressource en question à l'intérieur de l'ordonnanceur. Au même moment

que cela, le préchargement des données est signalé à StarPU afin d'optimiser l'exécution des tâches. Lorsque les ressources ont besoin de travail et utilisent la méthode pop, elles prennent une tâche dans leur file personnelle s'il y en a.

L'ordonnanceur dmda a beaucoup d'avantages ce qui explique sa popularité. Il limite les problèmes de contention car les ressources ont des files de tâches assignées différentes, il propose aussi de précharger les données et même de favoriser la localité des données grâce à la prise en compte des coûts de transferts. L'ordonnanceur dmda facilite la prise en compte de l'hétérogénéité des ressources grâce à l'estimation du temps de complétion ce qui favorise les ressources adaptées à l'exécution des tâches. Cependant plusieurs problèmes existent, le coût de la soumission d'une tâche est au minimum de l'ordre de $\mathcal{O}(nb\_ressources)$ mais surtout contrairement à HEFT qui dispose d'une vue globale du graphe de tâches, dmda n'observe qu'une tâche à la fois et il a tendance à favoriser excessivement les ressources très rapides. Il peut aussi pour la même raison faire des erreurs d'ordonnancement et choisir une tâche assez mal adaptée pour une ressource alors qu'il est possible que la prochaine tâche soumise soit parfaite pour celle-ci.

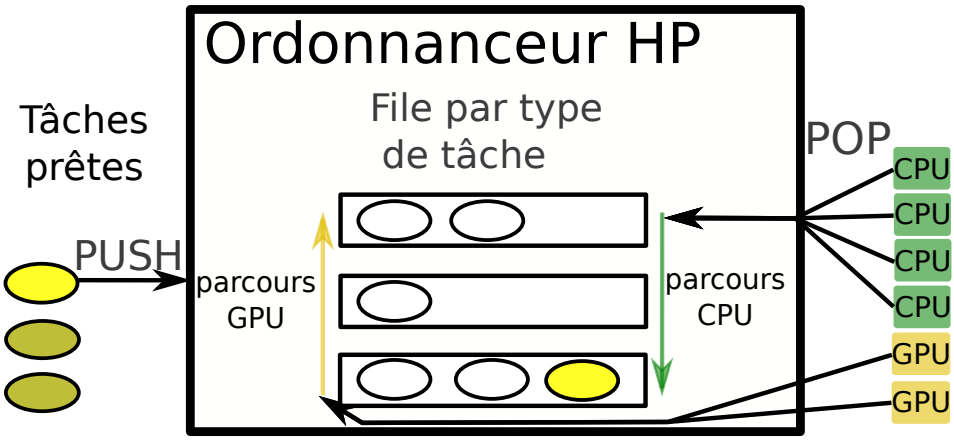


FIGURE 2.5 – Schéma de l'ordonnanceur HETEROPRIO de StarPU.

L'ordonnanceur HETEROPRIO L'ordonnanceur HETEROPRIO [16, 10] repose sur l'utilisation de facteurs d'accélérations sur GPUs par rapport aux CPUs pour chaque type de tâche afin d'établir une affinité entre les ressources et ces différents types de tâches. Afin d'utiliser au mieux les ressources hétérogènes, les GPUs et les CPUs se concentrent sur les tâches pour lesquelles ils sont plus efficaces respectivement. Autrement dit, les GPUs préfèrent les tâches pour lesquelles ils ont un facteur d'accélération élevé, alors que les CPUs exécutent les tâches avec un faible facteur d'accélération sur GPU. Pour ce faire, HETEROPRIO créé plusieurs queues, une par type de tâche, qui sont ordonnées

selon leur facteur d'accélération. Lorsqu'un CPU (*resp.* GPU) devient inactif, il reçoit une tâche provenant de la file non-vide avec la facteur d'accélération le plus faible (*resp.* élevé). Ceci est montré dans la Figure 2.5. Les tâches prêtes, dont toutes leurs dépendances sont satisfaites, sont poussées dans l'ordonnanceur et sont placées dans la file appropriée selon le type de la tâche. Lorsqu'une ressource a besoin de travail, elle parcours ces files dans un ordre croissant ou décroissant selon la ressource afin de trouver les tâches les mieux appropriées pour celle-ci.

Les ordonnanceurs modulaires et l'implantation modulaire de HEFT

Les ordonnanceurs modulaires créent une nouvelle façon de concevoir les ordonnanceurs introduite dans le cadre de la thèse de Marc Sergent [103] afin de faciliter le passage à l'échelle des ordonnanceurs. Les ordonnanceurs modulaires permettent de plus la réutilisation de parties d'autres ordonnanceurs ou encore le contrôle du débit de soumission des tâches prêtes. Ces ordonnanceurs sont structurés en modules (ou composants) d'ordonnancement qui, grâce à plusieurs fonctions d'interface, interagissent entre eux et coopèrent à la réalisation d'un ordonnancement global.

La Figure 2.6 présente la structure de l'ordonnanceur modulaire HEFT, similaire à dmda, réalisé à partir de composants d'ordonnancement. Chaque rectangle sur la figure représente un module d'ordonnancement, chacun branché explicitement aux autres modules selon un modèle spécifique représenté par les flèches. Il existe des modules de plusieurs natures, sur ce schéma on trouve en vert les modules purement algorithmique sans structure de données interne, les composants bleus servent à stocker des tâches et les composants violets peuvent aussi stocker des tâches mais fournissent aussi des primitives particulières car ils représentent les workers.

Lorsqu'une tâche prête est libérée, elle arrive dans un premier composant qui sert de fenêtre d'ordonnancement, implanté de fait par une file de tâches avec priorité, qui consiste à limiter le nombre de tâches poussées au maximum à la taille de la fenêtre. Ensuite, un composant vérifie si la tâche poussée dispose d'un modèle de performance ou non : si oui, elle peut être ordonnancée par le composant MCT de façon similaire à dmda, si non une calibration est forcée grâce à l'utilisation d'un composant de type eager. Lorsque l'algorithme MCT choisie la meilleure ressource, elle pousse la tâche au composant prio lié à la ressource en question. Le composant prio sert à accumuler les tâches de façon paresseuse et notamment à les ordonner correctement au fur et à mesure que de nouvelles tâches arrivent. Lorsque le worker appelle la méthode pop pour trouver une tâche, il va d'abord regarder dans son composant puis remonter récursivement l'arbre jusqu'à trouver une tâche qu'il peut exécuter. Il est important de noter que plusieurs méthodes d'interface entre composants leur permettent de communiquer entre eux et ainsi de propager les requêtes

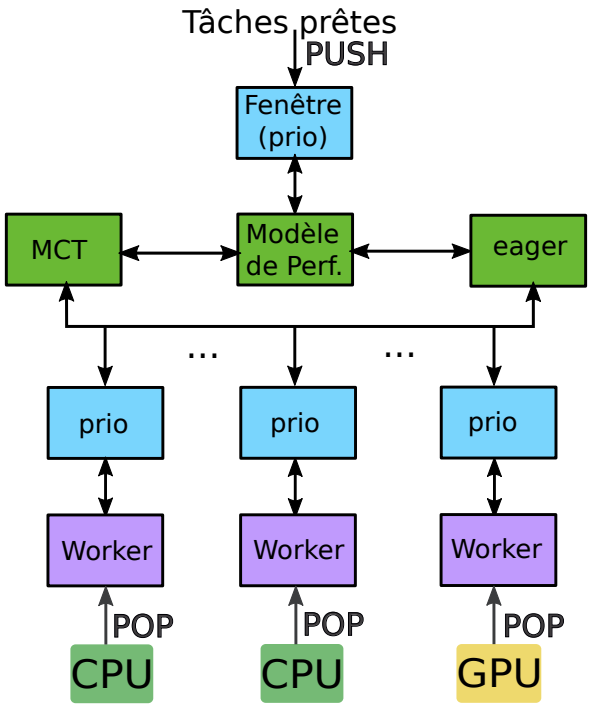


FIGURE 2.6 – Schéma de l'ordonnanceur HEFT implanté grâce aux modules d'ordonnancement de StarPU. Les composants verts sont uniquement algorithmiques, les composants bleus stockent des tâches, les composant violet peuvent aussi stocker des tâches et sont spécifiques aux workers.

de tâches. Les ordonnanceurs modulaires implantent aussi des méthodes de notification aux autres composants, par exemple pour signaler à un composant fils qu'une tâche est disponible dans sa file de tâche locale, que celui-ci peut venir chercher par la suite. Enfin, chaque composant peut retourner sa charge de calcul ainsi que celles de ses composants fils, ce qui facilite une meilleure estimation de la réelle occupation des ressources utiles à des ordonnanceurs comme MCT (HEFT).

2.2.3 Contextes d'ordonnancement

Les contextes d'ordonnancement sont un outil introduit dans le cadre de la thèse d'Andra Hugo [69]. L'objectif des contextes d'ordonnancement est de permettre la composition de bibliothèques parallèles et de proposer à l'utilisateur une façon de contrôler l'allocation de ressources depuis son application. Ces contextes d'ordonnancement donnent notamment la possibilité d'utiliser des ordonnanceurs différents pour différentes ressources, ce qui en font un outil clé pour l'ordonnancement et la composition de flux de tâches.

La Figure 2.7 représente le principe général des contextes. Dans StarPU

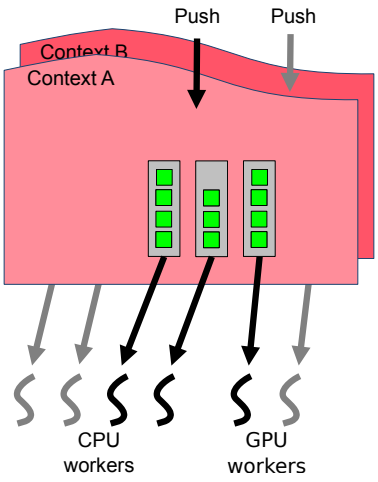


FIGURE 2.7 – Deux contextes d'ordonnancement partageant 4 cœurs de CPUs et 2 GPUs.

des "workers" (processus léger) représentent les ressources. Le principe des contextes est de fournir une API isolant ces ressources à l'intérieur d'un sous ensemble cohérent dans StarPU : les contextes. Chaque contexte possède son propre ordonnanceur et reçoit les tâches séparément des autres. Par exemple si une ressource A se trouve dans les contextes 1 et 2 simultanément, cette ressource peut avoir un lot de tâches qui transite par le contexte 1 et second lot, séparé, qui transite par le contexte 2.

2.3 Problématiques

Malgré les nombreuses avancées des supports d'exécutions afin d'obtenir une expression intuitive du parallélisme et une bonne portabilité des performances, notamment celles apportée par StarPU 2.2, plusieurs problèmes fondamentaux subsistent. Une première limite actuelle concerne les problèmes liés à granularité des tâches et de l'efficacité des algorithmes sur les machines modernes notamment de type "manycore". En second lieu, il est nécessaire de s'intéresser au problème d'expression du parallélisme plus en détail afin de proposer un modèle de composition de bibliothèques parallèles adapté aux modèles de machines modernes, un second objectif est d'identifier les limites du parallélisme de tâches pour l'expression de certains algorithmes.

2.3.1 Granularité et architecture des ordinateurs

Plusieurs types de machines modernes pour le calcul haute performances sortent du lot, d'un côté il y a les machines multicœurs équipées d'accélérateurs

(notamment de GPUs NVIDIA), et plus récemment des machines équipées d'Intel Xeon Phi Knights Landing (KNL) dotées d'un très grand nombre de cœurs de calcul (72 cœurs pour la dernière version) s'introduisent dans le calcul haute performance.

Ces machines créent plusieurs défis pour le calcul haute performance, d'un côté les machines dotées d'accélérateurs sont très hétérogènes et leurs ressources ont deux visions très opposées : d'un côté un nombre non négligeable de cœurs (souvent 20 à 50) composent les CPUs des machines de calcul et chaque cœur est utilisé séparément par les modèles de programmation usuels, de l'autre côté chaque GPU représente une unique unité de calcul avec une très forte puissance de calcul souvent de l'ordre de 30 fois plus rapide qu'un cœur d'un CPU. La Figure 2.8 illustre ce fait en évaluant la performance du noyau BLAS3 DGEMM (multiplication matricielle en double précision) sur un GPU K40m et un cœur de CPU Intel Xeon E3 2680 v3 lorsque ce noyau tourne seul. Comme on peut observer sur cette figure, la performance maximale d'un cœur de CPU de l'ordre de 40 GFlop/s est atteinte dès l'utilisation de matrices de taille 300×300, alors que pour un GPU la performance maximale est de l'ordre de 1125 GFlop/s et est atteinte pour des matrices de taille 2000×2000.

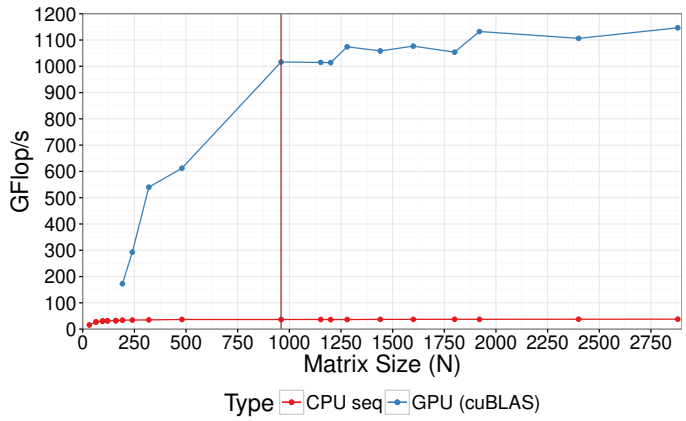


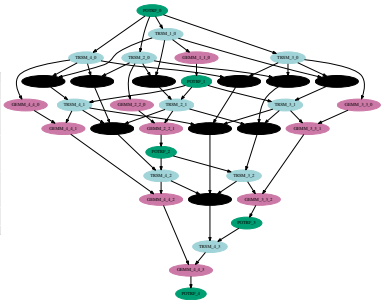
FIGURE 2.8 – Performance du noyau BLAS GEMM sur un GPU Nvidia K40m et un cœur de CPU Intel Xeon E3 2680 v3.

En ce qui concerne les machines de type manycore ou même les machines de calcul récentes équipées de plusieurs processeurs à l'intérieur d'une même machine, un problème important concerne l'utilisation du cache et le respect de la hiérarchie mémoire. En effet, ces machines disposent de caches partagés soit de niveau L3 et même L2 pour l'Intel KNL cependant ces ressources sont utilisées de façon indépendantes. Cette utilisation des cœurs de calcul des CPUs créé de nombreux problèmes de contention sur les ressources et sur la bande passante car les cœurs sont mis en compétition plutôt que d'utiliser un modèle

de calcul coopératif. Une autre solution consiste à s'intéresser à la granularité des calculs afin de limiter ces problèmes de contention.

```
1 for k : 0 → MT-1
    submit(DPOTRF, A_{k,k}:RW)
3   for m : k+1 → MT-1
      submit(DTRSM, A_{k,k}:R, A_{m,k}:RW)
5   for n : k+1 → NT-1
      submit(DSYRK, A_{n,k}:R, A_{n,n}:RW)
7     for m : n+1 → MT-1
        submit(DGEMM, A_{m,k}:R, A_{n,k}:R, A_{m,n}:RW)
```

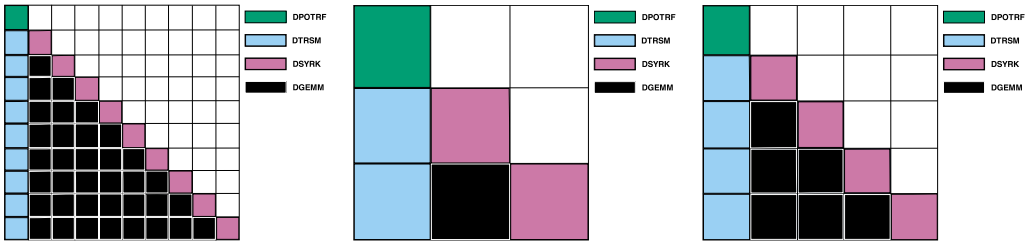
(a) Algorithme de soumission de tâches



(b) Graphe de tâches associé

FIGURE 2.9 – Algorithme et graphe de tâches de la factorisation de cholesky.

La factorisation tuilée de cholesky, un algorithme de référence en algèbre linéaire dense, permet d'illustrer ces problèmes de granularité. La Figure 2.9a montre l'algorithme simplifié à base de tâches de la factorisation de cholesky tel qu'implanté au dessus de StarPU dans la bibliothèque d'algèbre linéaire `Chameleon` [15]. Comme présenté précédemment des tâches sont soumises et prennent des données en paramètres avec un mode d'accès. De plus une structure (nommée codelet dans starpu) est passée en paramètre de la fonction de soumission, cette structure représente l'opération à exécuter et encapsule notamment les différentes implantation du noyau sur les architectures ciblées. La première opération DPOTRF est la factorisation de Cholesky en question et s'applique uniquement à la tuile diagonale $A_{k,k}$. L'opération TRSM résout le système d'équations ainsi créé pour factoriser la première colonne de la matrice, et les autres opérations DSYRK et DGEMM réalisent la mise à jour de la matrice grâce à des multiplications matricielles (de tuiles). Le résultat de la soumission de cet algorithme est représenté par le graphe de tâches 2.9b.



(a) Pour CPU. (b) Pour GPU. (c) Découpage intermédiaire.

FIGURE 2.10 – Exemples de granularité des blocs de factorisation de Cholesky d'une matrice.

La Figure 2.10a (*resp*. Figure 2.10b), montre un découpage d'une matrice de

taille identique dans les deux cas pour le calcul d'une factorisation de Cholesky proposant un nombre de tâches et une granularité adapté à plusieurs cœurs d'un CPU (*resp.* GPU). De fait, l'algorithme utilisé par `MAGMA` un code de référence pour l'algèbre linéaire dense sur plusieurs GPUs utilise une variante avec une partition 1D de la matrice et non pas 2D, créant des tâches de plus gros grain que montré sur la Figure 2.10b. Cette forte différence de granularité des calculs entre les deux problèmes illustrent la difficulté du choix de granularité de calcul pour les plateformes hétérogènes. De fait, pour résoudre ce problème une granularité intermédiaire est souvent choisie comme dans la Figure 2.10c. L'utilisation d'une granularité intermédiaire créé plusieurs problèmes. Souvent cette granularité est assez large (*e.g.* de l'ordre de 1000×1000) car la performance des GPUs est plus importante que celle des CPUs. Ceci tend à favoriser les GPUs et à focaliser la plupart des performance sur ceux-ci car il est possible que les cœurs des CPUs n'aient pas assez de tâches pour être remplis, et les CPUs deviennent ainsi sous-utilisés. De plus, bien que ces granularités sont assez large cela n'est souvent pas suffisant pour utiliser les GPUs à pleine puissance comme montré dans la Figure 2.8.

2.3.2 Programmation et modèle

Régler le problème du choix de la granularité des tâches et des différences de calcul entre CPU et GPUs permettrait de régler en partie le problème de portabilité des performance. Il est cependant nécessaire de considérer les problèmes de portabilité des codes lors de l'utilisation de supports d'exécutions.

Composition de codes et bibliothèques Comme montré en partie 2.1, les bibliothèques de calcul haute performance utilisent désormais des supports d'exécution pour l'utilisation des machines modernes. De plus, pour favoriser la réutilisation de codes et la combinaison des compétences de différents programmeurs, les programmes sont créés à base d'un empilement de bibliothèques spécialisées. Ceci créé un challenge pour le futur. Lorsqu'une bibliothèque utilise un support d'exécution spécifique pour son implantation, l'empilement de bibliothèques devient complexe. Un premier problème concerne la compatibilité ou non des modèles de parallélisme des différents supports d'exécutions. De plus, chaque support d'exécution dispose de sa propre vision de la machine sans avoir connaissance des autres supports d'exécutions, ainsi il y a de forts risque de surexploitation des ressources de calcul. Enfin, chaque support d'exécution peut avoir une vision différente d'un calcul, l'un peut par exemple avoir une vision du calcul global et un autre seulement un noyau, ceci créé donc un challenge pour l'ordonnancement et une nécessité d'organiser le calcul de l'algorithme et son parallélisme. L'approche proposée par [69] fourni des outils à l'utilisateur pour organiser la composition de supports d'exécution sur différentes ressources de la machine. Cependant, il reste à proposer une méthode

permettant d'orchestrer, de façon automatique si possible, différentes bibliothèques parallèles et supports d'exécutions afin de composer les parallélismes.

Programmation de noyaux Le principe des supports d'exécution est de proposer des modèles adaptés aux calculs. Cependant il est possible qu'un modèle atteigne ses limites lors de l'implantation de certains algorithmes, et ainsi ne permette pas l'obtention de performances. Dans le cas de la programmation à base de tâches deux limites importantes peuvent être identifiées : 1) le modèle de tâche à lui seul peut forcer la création de tâches trop petites pour être efficaces dans certains algorithmes, tel que la factorisation $PA = LU$ et 2) pour certains algorithmes il est possible que le graphe de tâche manque de parallélisme et ne permette pas l'obtention de performances satisfaisantes.

L'algorithme $PA = LU$ est présenté plus en détail dans la Section 4.2. Cet algorithme réalise la décomposition d'une matrice A en deux matrices triangulaires L et U. Cet algorithme pour être appliqué aux matrices générales nécessite une recherche de maximum dans chaque colonne de la matrice et la permutation de la ligne contenant cet élément avec la ligne où se trouve l'élément diagonal de la matrice afin d'obtenir une meilleure stabilité numérique. Pour ce faire, l'algorithme requiert donc une synchronisation entre chaque colonne de la matrice pour ces recherches de maximum, puis permutation, puis factorisation ce qui créé un grand nombre de tâches de très petites tailles lorsque l'on utilise le modèle de tâches pures.

Le deuxième problème survient dans le cas inverse pour les applications qui à certains moments de l'exécution génèrent de longues chaînes de tâches, et n'exhibent donc aucun parallélisme. C'est le cas notamment de l'application FLUSEPA qui est exposé plus en détail dans la Section 4.1. Cette application est un code de mécanique des fluides développé chez Airbus DS. Selon les maillages qui ne sont pas uniformes, dans certains cas il est possible que le travail soit concentré sur certaines cellules et ainsi des chaînes de tâches sont générées.

2.3.3 Discussion

Plusieurs solutions ont été proposées dans le passé pour résoudre les problèmes de granularité qui apparaissent entre les CPUs et les accélérateurs dans le contexte spécifique de l'algèbre linéaire dense. La plupart de ces méthode dépendent de l'utilisation de tailles de tuiles hétérogènes [108] ce qui peut générer des copies supplémentaire en mémoire quand les données ont besoin d'être fusionnées [76]. Cependant, la décision de découper une tâche est principalement faite statiquement lors de la soumission du travail. Plus récemment, une approche plus dynamique a été proposée dans [118] où une tâche à gros grain est découpée hiérarchiquement lors de l'exécution de celle-ci sur des CPUs.

Bien que cette solution résout le problème, celle-ci est spécifique à l'algèbre linéaire.

L'utilisation de tâches divisibles fournit un contrôle total sur la granularité des tâches. Cependant, cela nécessite une division récursive et des modèles de performances complexes afin de trouver la bonne taille de donnée pour un certain problème. Cette recherche de bonne granularité possède de plus un surcoût significatif. Les tâches hiérarchiques comme proposées par [118] simplifient cela en partant du principe que dans le cadre d'une machine hétérogène, il n'y a besoin que de deux granularités : celle pour les CPUs et celle pour les GPUs. Le modèle de tâches hiérarchiques peut aussi être considéré comme une nouvelle fonction implanté pour chaque noyau, où plutôt que de réaliser l'opération souhaitée d'autres tâches sont soumises sur les données d'entrée mais à un grain plus fin, les tâches hiérarchiques permettent donc de substituer une tâche d'un graphe de tâche par un sous-graphe de tâche. Cela favorise ainsi le contrôle de la granularité du calcul tout en simplifiant la recherche d'une bonne granularité. Cependant, il est tout de même difficile d'évaluer les performances d'un groupe de tâches dans le cadre d'une exécution dynamique sur toute une plateforme ce qui complexifie l'ordonnancement des tâches hiérarchiques. Le support d'exécution se doit aussi d'analyser les dépendances et de soumettre les transferts de données nécessaires entre les différents niveaux de données, soit à gros grain soit à petit grain ce qui peut être complexe. De plus, du point de vue de l'utilisateur ce mode d'utilisation l'oblige à implanter une version hiérarchique de son algorithme et ainsi implanter une nouvelle soumission de tâches pour chaque noyau. Enfin, pour ces deux solutions il n'y a pas d'améliorations pour la composabilité des supports d'exécutions, seuls les problèmes de granularités sont résolus.

Une solution possible pour résoudre le problème de granularité centrée sur les GPUs serait de ne plus considérer le GPU comme un accélérateur prenant un unique calcul en entrée, mais de la même façon qu'un CPU, comme un ensemble de cœurs sur lesquels il est possible de lancer plusieurs calculs avec un grain plus fin simultanément. Ceci semble d'autant plus que l'évolution de la puissance de calcul montre que l'écart de performance entre CPU et GPU ne fait qu'empirer. En effet, le GPU Pascal P100 possède 1.6× plus de GFlop/s que le GPU Kepler K80, la génération précédente. Une technologie récente et son ajout dans les supports d'exécutions permettent d'obtenir un résultat similaire : la technologie multistream de CUDA. Grâce à cette technologie il est possible de soumettre des calculs de façon simultanée aux GPUs Nvidia qui, en utilisant son ordonnanceur interne, les réparti automatiquement sur les unités de calcul matérielles du GPU. Le problème de cette technique est que l'ordonnanceur interne de la carte graphique peut assigner le noyau de calcul à plusieurs SMs (unités de calcul similaire à un cœur de processeur) ou bien à seulement une fraction de ce SMs (sur un warp ou deux). Autrement dit, les noyaux de calcul poussés sur GPU avec l'utilisation de multistream obtiennent

une forte variabilité ce qui rend toute prédiction de temps de calcul impossible et complexifie grandement l'ordonnancement. Une technique intéressante pour contrôler cette variabilité serait de réaliser une partition de GPUs en plusieurs unités de calcul de même taille (ou non de façon volontaire si cela a un intérêt pour les performances), chacune contenant un SM ou plus.

Dans le contexte de cette thèse, on propose de s'intéresser au problème de granularité des calculs et de composition de code grâce à une nouvelle approche, basée sur un modèle théorique bien étudié : plutôt que de couper des tâches à gros grains comme dans les modes de tâches divisibles ou hiérarchiques, on agrège les unités de calcul afin qu'elles coopèrent à l'exécution d'une tâche en parallèle. Pour cela, on se base sur le concept de *tâches parallèles* introduit de façon théorique dans le monde de l'ordonnancement. Ceci fait aussi écho à la partition de GPUs présentée plus tôt : en partitionnant le GPUs en unités de calcul de certaines tailles, on peut aussi concevoir les tâches exécutées sur chaque sous-unité des tâches parallèles, dont le nombre de ressources attribuées à chaque tâche peut être contrôlé comme l'impose le modèle théorique. L'intérêt de l'usage de tâches parallèles est que cela permet de faire concurrencer la puissance agrégée des cœurs de CPUs avec celle d'un GPU entier, dont l'ordre de grandeur est comparable contrairement à la comparaison d'un cœur de calcul vis à vis d'un GPU entier.

L'utilisation des tâches parallèles a plusieurs inconvénients mais aussi de nombreux avantages. En effet, le problème de l'ordonnancement de tâches est complexifié du fait de la décision du nombre de ressources à attribuer aux tâches et comme observé en Section 2.1.3, peu de travaux ont été réalisés pour la réalisation d'ordonnanceurs de tâches moldables dynamiques pour une machine hétérogène, alors que ces ordonnanceurs sont nécessaires dans le cadre d'un support d'exécution dynamique. De plus, puisque le modèle est pour l'instant purement théorique il reste à juger si celui-ci est adapté à la pratique et simple à mettre en œuvre et à utiliser. Enfin, ce modèle requiert une bonne scalabilité des bibliothèques parallèles de calcul pour permettre leur utilisation à l'intérieur d'une tâche. Ce point ne semble pas consister en une grande exigence puisque le contexte d'application de cette technique est le calcul haute performance.

Cependant, un premier avantage de l'utilisation de tâches parallèles est que cela n'implique aucun changement pour l'utilisateur : le graphe de tâches soumis au support d'exécution reste le même, seul la vision des ressources change, ce qui peut être réalisé en interne par le support d'exécution. Ce modèle peut de plus proposer à l'utilisateur plus de contrôle sur l'exécution de son algorithme. Il est par exemple possible de donner des indices sur l'importance d'une tâche pour favoriser son exécution rapide grâce à l'utilisation d'un nombre important de ressources en parallèle pour l'exécution de cette tâche. L'utilisation de tâches parallèles permet de combiner le parallélisme de tâches avec le parallélisme de données, ce qui propose une bonne réutilisa-

tion des données à l'intérieur des tâches et ainsi une meilleure utilisation des ressources de la machine ainsi qu'un respect de la hiérarchie mémoire. Cette combinaison des parallélismes favorisent des entorses au modèles de tâches. Des applications comme la factorisation $PA = LU$ qui ont besoin de beaucoup de synchronisation fine pour chaque colonne de la matrice peuvent réaliser ces synchronisations non pas entre des tâches mais à l'intérieur de celle-ci pour un surcoût et un nombre de tâches généré bien moindre. Un autre avantage important de cette technique est que le fait de rendre les ressources plus homogènes rend le choix de la granularité des tâches de calcul libre. En effet, plutôt que de garder la granularité de calcul de compromis entre les cœurs des CPUs et les GPUs présentée dans la Figure 2.10c, il est judicieux de jouer sur cette granularité et par exemple l'augment à celle requise par les GPUs comme dans la Figure 2.10b. Enfin, l'utilisation de tâches parallèles propose une composition naturelle des bibliothèques de calcul haute performance en appelant des bibliothèques parallèles efficaces à l'intérieur des tâches.

2.4 Contributions

Lors de sa thèse, Cédric Augonnet [21] a mis en place le système de "combined workers" une première approche qui a mis en avant l'intérêt de créer des tâches parallèles dans le cadre d'applications de type stencil. Cette première implantation explique que Cédric Augonnet conclue sa thèse en proposant pour le futur du calcul haute performance que les programmeurs devraient avoir la possibilité d'écrire des algorithmes de tâches pouvant invoquer des bibliothèques parallèles existantes fortement optimisées afin de créer un partage efficace des ressources de la machine. Andra Hugo a apporté une première contribution dans ce sens dans le cadre de sa thèse [69] qui a introduit les contextes d'ordonnancement présentés en Section 2.2.3 afin de permettre aux utilisateurs de structurer le parallélisme d'applications complexes.

Le premier objectif de cette thèse est de permettre une composition de codes parallèles, notamment du modèle de tâches avec des bibliothèques existantes proposant des noyaux parallèles efficaces, grâce à l'utilisation du modèle de tâches parallèles. Le second objectif est de répondre au problème de la granularité en proposant l'agrégation de ressources (pour utiliser des tâches parallèles) comme solution plutôt que le découpage de tâches. Enfin, le dernier objectif majeur de cette thèse est de prouver avec des études expérimentales fournies que ce modèle de tâches parallèles couplé à l'utilisation de bibliothèques parallèles existantes permet d'obtenir de meilleures performances que les solutions de l'état de l'art.

Le Chapitre 3 propose une étude détaillée du modèle des tâches parallèles. Une implantation de tâches parallèles rigides est proposée et il est montré dans le cadre d'une étude expérimentale détaillée sur la factorisation de Cholesky

que l'utilisation de tâches parallèles rigides permet des gains de performances significatifs sur les architectures modernes de type hétérogène ou manycore.

Le Chapitre 4 montre l'utilisation de tâches parallèles rigides pour permettre à l'application de CFD Flusepa et la factorisation $PA = LU$ d'être implantées efficacement sur un support d'exécution à base de tâche.

Le Chapitre 5 propose une évolution du modèle pour l'utilisation de tâches parallèles moldables. En premier lieu, l'ordonnanceur HeteroPrio est adapté à l'utilisation de tâches parallèles grâce a la généralisation du concept de facteur d'accélération. Une étude expérimentale grâce à une évaluation exhaustive montre l'intérêt des tâches parallèles moldables pour l'obtention de performances. En second lieu, une plateforme d'exécution et de création d'ordonnanceurs de tâches parallèles moldables est proposée afin de permettre le changement dynamique de la configuration des ressources de la machine sous le contrôle de l'ordonnanceur. Une évaluation de cette implantation est conduite en implantant deux ordonnanceurs de tâches parallèles moldables pour machines hétérogènes, un ordonnanceur simple et l'ordonnanceur de [25].

Chapitre 3

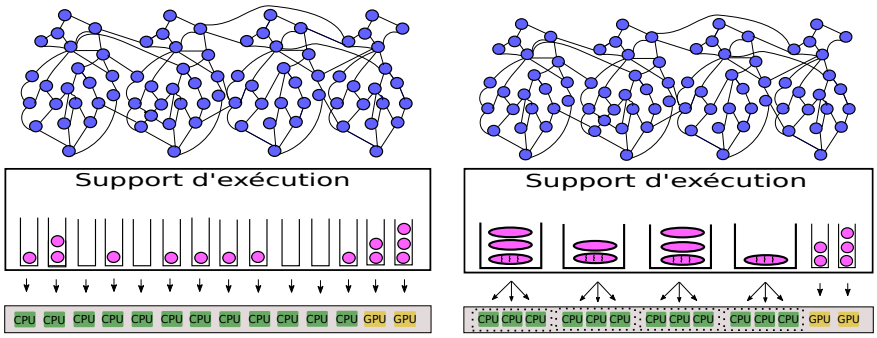
L'agrégation de ressources pour la mise en œuvre de tâches parallèles rigides

Sommaire

L'objectif de ce chapitre est de présenter une approche pratique de l'exploitation du parallélisme interne aux tâches grâce à l'utilisation de tâches parallèles. Les questions abordées dans ce chapitre sont les suivantes. Quels sont les besoins du modèle de tâches parallèles ? Comment mettre en œuvre l'utilisation de tâches parallèles ? Comment combiner l'utilisation plusieurs supports d'exécution à travers le modèle des tâches parallèles ? Comment modéliser les performances des tâches parallèles ? Quelles sont les limites de ce modèle et des choix d'implantations réalisés ici ? L'étude de ces questions amènent à la conception d'outils et de principe d'exploitation concrète du parallélisme interne au tâches à travers le support d'exécution StarPU. Afin d'analyser avec

détail les avantages et limites de cette méthode, une étude expérimentale détaillée est menée dans ce chapitre en se positionnant dans le cas où l'on fixe le nombre de ressources pour toute une exécution, afin d'obtenir une partition homogène de la machine. Cette méthode permet d'obtenir de meilleures performances que les autres bibliothèques étudiées sur une factorisation de Cholesky sur l'Intel Xeon Phi Knights Landing (`Chameleon` [15], `PLASMA` [7], Intel MKL [40]) ainsi que sur une machine hétérogène moderne (`Chameleon` [15], `DPLASMA` [26], `MAGMA` [7]).

3.1 Le modèle de tâches parallèles



(a) Tâches séquentielles uniquement. (b) Utilisation de tâches parallèles.

FIGURE 3.1 – Schéma général de gestion de tâches parallèles dans un support d'exécution.

Pour utiliser les tâches parallèles il est nécessaire d'introduire tout un ensemble de mécanismes permettant le contrôle du parallélisme interne aux tâches dans le support d'exécution. La Figure 3.1a présente l'architecture standard d'un support d'exécution. Un graphe de tâches est soumis au support d'exécution et les tâches prêtes (en violet) sont assignées dynamiquement à des files associées aux ressources sous-jacentes. Par contraste, la Figure 3.1b représente une adaptation du premier schéma avec l'utilisation de tâches parallèles. En premier lieu, l'utilisation de tâches parallèles ne remet pas en cause le graphe de tâches soumis au support d'exécution. Ce graphe de tâches ne change pas et l'objectif ici est d'impacter un minimum l'écriture d'application. Le changement principal ici est que les ressources sur CPU qui sont différenciées dans Figure 3.1a sont regroupées afin d'avoir une vision de *ressources virtuelles* que l'on appellera par la suite *groupes de cœurs*. C'est sur ces ressources virtuelles que les tâches prêtes, désormais parallèles, s'exécutent. Dans cet exemple, chaque groupe de cœurs contient 3 cœurs de CPU, cependant le

modèle ne doit pas imposer une partition parfaite de la machine en groupes de cœurs.

Plusieurs points critiques sont étudiés dans la suite de cette section. Le premier concerne la gestion des deux niveaux de parallélisme qui sont le parallélisme du graphe et le parallélisme interne aux tâches ; le deuxième concerne l'extension de politiques d'ordonnancement afin de rendre l'utilisation de tâches parallèles transparentes.

3.1.1 Imbrication des parallélismes

Comme observé sur la Figure 3.1b, l'utilisation de tâches parallèles implique deux niveaux de parallélisme qu'il faut faire cohabiter. Afin de poursuivre les efforts des programmeurs pour la combinaison de briques logicielles spécialisées, il est nécessaire de supposer dans notre modèle que plusieurs outils différents doivent cohabiter pour gérer ces deux niveaux de parallélisme. Dans ce modèle, il y a donc un premier support d'exécution que l'on appelle *support d'exécution externe* qui s'occupe du parallélisme de tâches. Ce support d'exécution aura un fonctionnement très similaire à la Figure 3.1a, il reçoit les tâches soumises et les ordonnancent sur des ressources. La seule différence est qu'ici les ressources sont des groupes de cœurs de cœurs de CPU. Enfin, il existe un second support d'exécution que l'on appelle le *support d'exécution interne* dont le but est de s'occuper du parallélisme interne aux tâches. Ce mécanisme est plus général et permet une imbrication de nombreux outils et parallélisme, mais il est suffisant de considérer le cas de deux supports d'exécution.

Plusieurs suppositions sont nécessaires afin que le modèle présenté soit réalisable :

1. Le support d'exécution interne doit accepter de s'exécuter sur un ensemble restreint de ressources
2. Le support d'exécution interne doit être réentrant, plus précisément pouvoir être appelé plusieurs fois en parallèle au même moment et sans interférences. Cela implique donc une privatisation ou non utilisation de variables globales et statiques.

Le support d'exécution externe quand à lui doit contraindre le support d'exécution interne à exécuter les tâches parallèles sur les groupes de cœurs qu'il créé. Cependant, il est nécessaire que cette interaction soit générique. De plus, elle doit fonctionner même sans aucune vue du parallélisme interne à la tâche. Pour ce faire, il est nécessaire de définir des points d'interaction entre les supports d'exécution externe et interne. Dans le modèle présenté ici, on propose deux points d'interactions présentés dans la Figure 3.2. Le premier permet de mobiliser les ressources du point de vue du support d'exécution interne, c'est à dire signifier au support d'exécution interne les ressources qu'il doit utiliser pour exécuter une tâche. Le second point d'interaction permet de libérer les

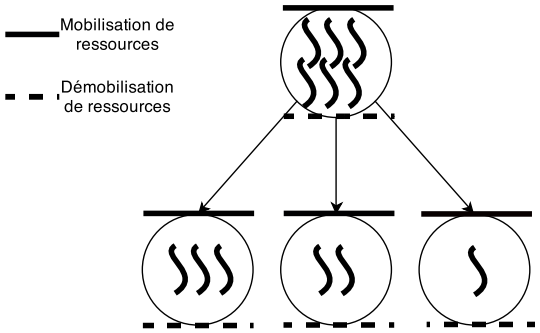


FIGURE 3.2 – Points d'interactions entre les supports d'exécution externe et interne.

ressources du point de vue du support d'exécution interne lorsque le travail est terminé. Ce point peut être facultatif selon le support d'exécution interne considéré ou intégré avant de mobiliser les ressources des tâches suivantes.

3.1.2 Modélisation des performances des tâches parallèles

Un problème avec le modèle considéré pour le support d'exécution externe est qu'il n'a aucune connaissance sur le support d'exécution interne ni sur le parallélisme interne. Cette utilisation en boîte noire de la bibliothèque et du support d'exécution interne aux tâches rend les décisions d'ordonnancement plus complexes. Afin de régler ce problème, le modèle proposé suppose que le support d'exécution externe possède un mécanisme de suivi de la performance des tâches grâce à un historique par exemple. Auquel cas, il est nécessaire d'ajouter un nouveau paramètre au modèle de performance des tâches qui est le nombre de ressources sur lequel la tâche s'est exécutée. Il pourrait même être nécessaire de connaître les ressources précises sur lesquelles une tâche parallèle s'est exécutée (*e.g.* si elles sont éloignées ou non). L'approche proposée ici est de déléguer ce problème à l'ordonnancement. En effet, on suppose que les ressources attribuées à une tâche parallèle sont systématiquement dans la meilleure configuration possible, auquel cas le nombre de ressources est un paramètre suffisant pour un modèle de performance de tâches parallèles.

Comme montré dans la Figure 3.1b, les ressources doivent être regroupés dans des ressources virtuelles (groupes de cœurs) pour l'utilisation de tâches parallèles. Pour ce faire, il est donc nécessaire de proposer aux ordonnanceurs des outils permettant de considérer ces ressources virtuelles pour l'allocation de tâches. Le deuxième point à prendre en compte est de mettre à jour de façon transparente les statistiques de toutes les ressources d'origine du support d'exécution après l'utilisation d'un groupe de cœurs de cœurs de CPUs. De

cette façon, les ordonnanceurs n'ont pas besoin de connaître la structure des groupes de cœurs pour réaliser l'ordonnancement de tâches séquentielles ou parallèles.

3.2 Éléments d'implantation

Dans cette section, des éléments d'implantation sont proposés. Le support d'exécution externe considéré est StarPU présenté précédemment. La première partie aborde la composition des supports d'exécution à travers un outil de StarPU, les contextes d'ordonnancement [69]. Le second point abordé concerne les modèles d'interaction avec les supports d'exécution externes. Ensuite, on regardera en détail l'adaptation d'ordonnanceurs de StarPU. Enfin, la dernière section aborde l'utilisation de hwloc pour prendre en compte la topologie de la machine.

3.2.1 Composition de supports d'exécution à travers les contextes d'ordonnancement

Les contextes d'ordonnancement ont été présentés en section 2.2.3. Les tâches parallèles partagent plusieurs propriétés avec les contextes, ce qui en fait un bon support pour une première implantation. En effet, il est nécessaire de composer des codes et bibliothèques parallèles (et même des supports d'exécution), tout en isolant des appels concurrents à une même bibliothèque parallèle.

L'utilisation de contextes pour représenter des ressources parallèles implique un changement crucial. Plutôt qu'isoler des codes distincts s'exécutant sur plusieurs groupes de ressources, il faut qu'un contexte devienne une ressource et la représente, c'est à dire une ressource virtuelle représentant plusieurs cœurs de CPUs utilisés en parallèle, un groupe de cœurs. Plutôt qu'isoler différents flux de calculs avec différents ordonnanceurs, il faut cette-fois ci ordonnancer au dessus de plusieurs contextes (*i.e.* dans ce cas des ressources parallèles) à la fois. Enfin, bien que ces ressources parallèles ainsi définies sont des ressources virtuelles, il faut permettre d'identifier ces ressources de façon transparente. La Figure 3.3 montre l'infrastructure générale mise en œuvre pour exécuter des tâches parallèles à travers les contextes.

Un contexte représentant une ressource parallèle a une propriété essentielle permettant de le distinguer d'un véritable contexte d'exécution : il n'a pas d'ordonnanceur interne StarPU puisque la gestion de l'ordonnancement est délégué soit à l'application, soit au support d'exécution interne. Du point de vue de StarPU, une ressource parallèle n'est qu'une simple ressource. De plus, un maître est distingué parmi les ressources et sert d'interface pour les ordonnanceurs : c'est à travers ce worker que l'on gère l'exécution et la soumission

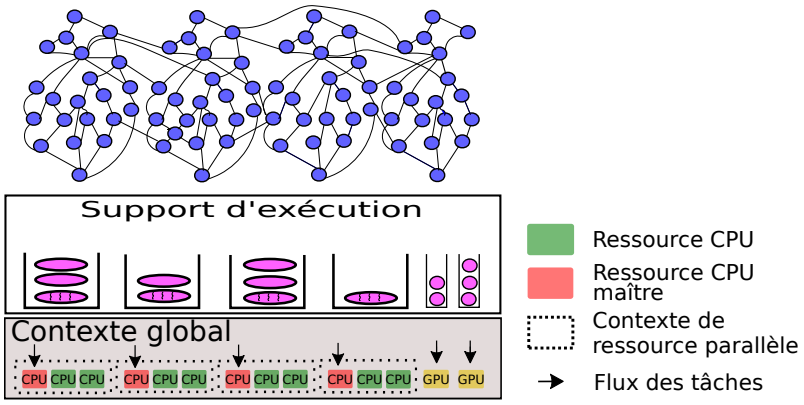
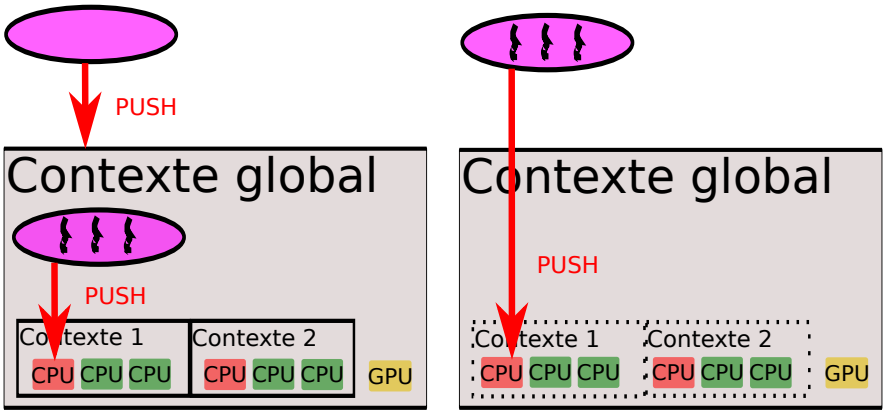


FIGURE 3.3 – Représentation de l'implantation de tâches parallèles à travers l'utilisation des contextes de StarPU.

de tâches parallèles. Il sert à identifier le groupe de ressources au travers d'une ressource pour StarPU. Il est important de noter que l'on suppose ici qu'une ressource ordinaire ne fait parti que d'une seule ressource parallèle à la fois. Un autre changement est l'ajout d'un type au contexte pour représenter différents modes d'utilisations des ressources qui sont principalement Single Program Multiple Data (SPMD) et maître-esclaves (fork-join). Plus de détails sur les types de ressources et leur gestion sont apportés dans la section 3.2.2. Enfin, un modèle de performance est associé au contexte, qualifié par le nombre de ressources correspondant pour représenter la performance d'une tâche parallèle. Dans le cas où une tâche parallèle est placée sur un groupe de cœurs, c'est le modèle de performance placé dans la structure des contextes qui est utilisé et mis à jour plutôt que celui associé à la ressource (cœur de CPU).

La sémantique des contextes soulève une question importante pour l'utilisation de ceux-ci comme une ressource. Faut-il respecter les contextes et soumettre la tâche à tout prix à travers le contexte représentant la ressource virtuelle ? Ceci a l'avantage de ne pas remettre en question le mode d'utilisation des contextes. Cependant, plusieurs point peuvent être source de problèmes. Chaque tâche doit être poussée deux fois, la première normalement comme dans StarPU et la seconde pour atteindre la ressource virtuelle parallèle. Un deuxième point est que puisque le contexte représentant une ressource parallèle n'a pas d'ordonnanceur, comment est-ce que l'on pousse cette tâche dans le contexte interne, dans quelle file et à quel moment, au push ou au pop ? Deux solutions sont possibles et implantées et présentées dans la Figure 3.4. Une première version présentée en Figure 3.4a nécessite pour cela l'utilisation explicite d'un code à l'intérieur des ordonnanceurs réalisant la soumission d'un contexte général à un contexte parallèle si besoin. Le positionnement de celui-ci est laissé à l'expertise du créateur de l'ordonnanceur. Afin de contourner ces points, une seconde méthode représentée en Figure 3.4b est aussi proposée.



(a) Respect de la sémantique des contextes.

(b) Non respect de la sémantique des contextes.

FIGURE 3.4 – Schémas de respect ou non de la sémantique des contextes pour la création de groupes de cœurs.

Il s'agit de considérer les contextes uniquement comme une source d'information sur l'existence d'une ressource parallèle ou non, avec quels workers, et quel modèle de performance. Autrement dit, le contexte représentant une ressource parallèle dans ce cas n'est réellement qu'une structure pour stocker des informations et gérer une certaine cohérence des ressources. La principale limitation de ce mode est son incompatibilité avec une utilisation classique des contextes. En pratique, cette deuxième version est utilisée dans l'évaluation expérimentale présentée en section 3.3.

3.2.2 Modèles d'interactions avec les supports d'exécution internes

Comme présenté en section 3.1.1, on intègre dans une fonction *callback* en prologue des tâches le code spécifique nécessaire pour contraindre le support d'exécution interne à s'exécuter sur un groupe de ressource parallèle. Ce prologue est appelé de façon transparente avant l'exécution de quelconque tâche parallèle. Cette approche peut être utilisée pour la plupart des supports d'exécution internes tant que le programmeur peut implanter ce code de mobilisation de ressources sur le groupe de cœurs correspondant. Ainsi, du point de vue de l'utilisateur en considérant qu'il a une implantation parallèle de ses noyaux, l'utilisation de groupes de cœurs dans son application est simple : il a besoin d'implanter le callback situé en prologue des tâches, de créer des groupes de cœurs et s'assurer qu'il utilise un ordonnanceur capable d'utiliser les tâches parallèles. En pratique, plusieurs optimisations peuvent être placées dans ce callback notamment pour ne réaliser l'interaction avec le support d'exécution

interne que lorsque cela est nécessaire, *i.e.* quand le nombre de ressources attribuées à ce groupe a été modifié.

Afin de faciliter l'écriture de ce prologue, plusieurs fonctions d'interface sont fournies. Des fonctions permettent d'identifier le maître du groupe de ressources courant et de récupérer les informations associées, tel que le nombre de ressources, les numéros virtuels de ces ressources et le type du groupe de cœurs courant.

Il est nécessaire d'identifier plusieurs types d'interactions entre le support d'exécution interne et le support d'exécution externe pour l'exécution d'une tâche parallèle. Trois modèles sont présentés selon l'utilisation de nouveaux processus légers ou de processus légers existants pour l'exécution de la tâche parallèle :

— Un modèle maître-esclave, où une création de processus légers équipiers depuis le support d'exécution interne est nécessaire

— Un modèle de réutilisation des processus légers du support d'exécution interne par le support d'exécution interne, typiquement à base de pthreads.

— Un modèle où les processus légers du support d'exécution interne de type pthreads seulement sont exécutés.

Modèle maître-esclave (OpenMP)

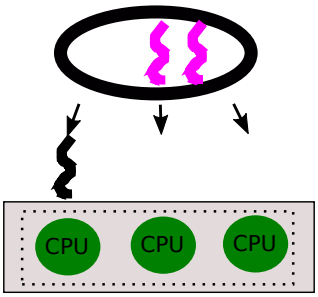


FIGURE 3.5 – Gestion d'une équipe de processus légers à l'intérieur d'un groupe de cœurs en mode maître-esclave.

L'utilisation d'une ressource maître/esclave quand-à elle oblige à endormir à la création de la ressource tous les workers StarPU correspondant aux esclaves du worker parallèle, à les réveiller à la destruction du contexte (de la ressource) et à proposer des données pour le suivi de l'état des processus légers créés par/pour le support d'exécution interne.

Si le support d'exécution possède son propre ensemble de processus légers (*e.g.* OpenMP), les workers StarPU correspondant au groupe de cœurs doivent être endormis jusqu'à la fin de la tâche parallèle. Ceci est fait pour éviter de "surexploiter" les ressources correspondantes et ralentir l'exécution. La

Figure 3.5 montre comment l'interaction est mise en œuvre. Seulement un worker StarPU a le droit de s'exécuter, c'est le worker distingué "maître" présenté dans la Figure 3.3. Ce worker maître se charge de récupérer les tâches attribuées au groupe de cœurs par l'ordonnanceur. Quand une tâche est exécuté, ce processus léger maître est considéré comme un processus léger d'application habituel pour le support d'exécution interne. Dans la Figure 3.5, le processus léger noir représente le worker StarPU et les processus légers roses créés par le support d'exécution interne.

```
#include <omp.h>
#include <mkl.h>

void cl_prologue()
{
  /* Get the current cluster */
  int cluster = starpu_get_current_cluster();

  /* get the CPUs of the cluster */
  int cpus[MAX_WORKERS];
  int ncpus = starpu_get_cpus(cluster, cpus);

 /* bind openmp threads to CPUs */
#pragma omp parallel num_threads(ncpus)
  bind_to_cpu(cpus[omp_get_thread_num()]);
}

void codelet_cpu_func()
{
  /* call the mkl parallel kernel */
  DGEMM(...);
}
```

FIGURE 3.6 – Exécution de code `Intel MKL` parallèle au dessus d'un groupe de cœurs.

Pour la bonne exécution de la tâche parallèle il est nécessaire de contrôler la création des processus légers roses de la Figure 3.5. La Figure 3.6 représente un exemple d'implantation du prologue nécessaire dans ce cas. L'objectif est d'isoler l'exécution des noyaux écrits avec OpenMP et de fixer l'exécution des processus légers créés par OpenMP sur les ressources prévues à cet effet. L'implantation du prologue "callback" présenté est un exemple avec des noms de fonction raccourcis qui consiste à :

1. Récupérer l'identifiant du groupe de cœurs actuel (ligne 7),
2. Récupérer un tableau d'identifiants des ressources attribuées au groupe de cœurs (lignes 10-11),
3. Créer une équipe de processus légers OpenMP requis par le noyau (ligne 14),
4. Fixer chaque processus léger OpenMP sure une ressource différente (ligne 15).

Par la suite pendant l'exécution de la bibliothèque parallèle utilisant OpenMP, par exemple la `Intel MKL` parallèle, le support d'exécution interne peut réuti-

liser les processus légers créés précédemment, tous fixés sur les bonnes ressources.

En pratique cette approche sera utilisée dans la section expérimentale pour forcer l'exécution de la bibliothèque `Intel MKL` parallèle, qui se base sur OpenMP, sur l'ensemble des ressources correspondant aux groupes de cœurs définis par StarPU.

Pthreads utilisant les processus légers de StarPU

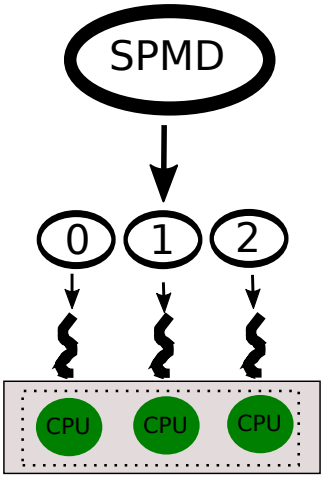


FIGURE 3.7 – Gestion d'une équipe de processus légers à l'intérieur d'un groupe de cœurs en mode SPMD avec des pthreads.

Un autre mode d'exécution différent du premier consiste à utiliser des "pthreads" pour l'écriture d'un noyau parallèle. Une première version pour réaliser cela consiste en la réutilisation des processus légers de StarPU, qui sont des pthreads. Pour ce faire, l'implantation fournie des groupes de cœurs duplique la tâche pour l'utilisateur et soumet la tâche en question sur chacun des workers de StarPU. Pour que cela fonctionne, un calcul atomique de rang ainsi qu'une paire de barrière avant, et après l'exécution de la tâche parallèle est mis en place. La Figure 3.7 représente l'utilisation du mode SPMD de cette façon. L'utilisateur a alors accès au travers du rang associé à chaque appel de la tâche parallèle à un mécanisme permettant l'implantation d'un noyau parallèle (ou la coordination avec un support d'exécution avec ce fonctionnement). Enfin, comme précédemment, de façon interne uniquement, un worker de StarPU est distingué comme "maître" du groupe de cœurs et sert d'interface pour celui-ci.

Pthreads utilisant les processus légers de l'application

Un second mode d'utilisation des pthreads consiste à utiliser les processus légers de l'application. Pour ce faire, une fonction est fournie qui endort tous les processus légers de StarPU que l'application peut appeler avant d'appeler une fonction de l'utilisateur. On se retrouve alors dans le cas de la Figure 3.5 et l'utilisateur peut créer ses processus légers comme il souhaite, mais doit s'assurer de les lier sur les ressources réservées par StarPU. Une fois le code utilisateur exécuté, les processus légers sont automatiquement mis dans leur état d'origine.

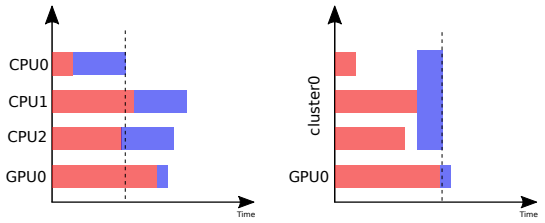
3.2.3 Adaptation d'ordonnanceurs

Puisque l'on se place dans un contexte d'exécution dynamique de tâches, l'ordonnancement des tâches est un point critique du support d'exécution. Pour faire usage de tâches parallèles dans ce cadre, plusieurs adaptations sont nécessaires. Les adaptations proposées dans cette section concernent aussi bien les ordonnanceurs eux-même que les modèles de performances sur lesquels ils peuvent se baser.

Afin de permettre un parcours des ressources transparent pour les ordonnanceurs, un itérateur de ressource est fourni. Cet itérateur a deux modes de fonctionnement : en mode tâche parallèle, uniquement les ressources maîtres de contextes sont considérés. Dans le mode habituel, toutes les ressources sauf celles endormies par StarPU sont considérées. Afin d'améliorer la réactivité des ressources vis à vis des contextes, le système de sélection du contexte dans lequel une tâche est récupérée a été amélioré. Par défaut, les workers StarPU alternent entre tous les contextes dont ils font parti pour chercher une tâche. Afin d'améliorer ce système, des compteurs sont proposés pour garder des statistiques sur l'occupation des contextes. Ces statistiques sont à ajouter à l'ordonnancement selon le comportement de ceux-ci. En utilisant ces statistiques, StarPU favorise les contextes avec le plus de tâches.

Avec ces outils, la généralisation des ordonnanceurs gloutons pour l'utilisation des tâches parallèles est simple tant que l'ordonnanceur n'a pas besoin d'utiliser des modèles de performance. Ainsi, les stratégies d'ordonnancement comme le vol de travail peuvent utiliser naturellement les tâches parallèles. Au contraire, la stratégie Minimum Completion Time (MCT), qui dépend de la prédiction du temps d'exécution des tâches, doit être adaptée. Dans StarPU, l'estimation du temps de calcul et de transfert d'une tâche est calculé par les modèles de prédictions de performance. Ces modèles sont basés sur des tables d'historique de performance créées dynamiquement pendant l'exécution de l'application. De plus, l'ordonnanceur MCT garde une trace des temps de début et fin d'exécution d'une tâche, ainsi que toutes les tâches prévues pour une ressource donnée. Ainsi, sans l'intervention du programmeur, le sup-

port d'exécution peut fournir une estimation de performance en fonction des données de la tâche ce qui permet à l'ordonnanceur de prendre des décisions appropriées lors de l'ordonnancement de tâches sur des ressources.



(a) MCT.

(b) MCT avec groupe de cœurs.

FIGURE 3.8 – Adaptation de la stratégie d'ordonnancement MCT.

La Figure 3.8a illustre le comportement de la stratégie MCT. La tâche en bleu représente celle que l'ordonnanceur est en train de considérer. Cette tâche a différents temps d'exécution sur CPU et GPU. Dans cette situation, l'ordonnanceur prend le choix de placer cette tâche sur le CPU0 qui termine le calcul de cette tâche au plus tôt. Cette stratégie MCT est adaptée ainsi que les modèles de performances associés pour prendre en compte l'existence de plusieurs ressources exécutant une tâche simultanément dans le cas des tâches parallèles. Dans le cas d'un groupe de cœurs, le modèle de performance est aussi paramétré par le nombre de ressources exécutant la tâche. Ainsi, une tâche parallèle peut être attribuée à un groupe de cœurs soit explicitement par l'utilisateur soit en choisissant le groupe de cœurs capable de terminer l'exécution de la tâche le plus tôt. Ceci est illustré par la Figure 3.8b, où les trois CPUs de la plateforme fictive sont regroupés dans un groupe de cœurs. Pour l'attribution de la tâche, le choix est fait entre la ressource fictive représentant les trois CPUs utilisés simultanément ou le GPU. Dans ce cas, la tâche est attribuée aux trois CPUs car ils peuvent terminer l'exécution de cette tâche avant le GPU. Afin de rendre l'utilisation de tâches parallèles transparente par rapport à l'utilisation des ressources avec des tâches séquentielles, le maître (surtout dans le cas d'un modèle "maître-esclave"), doit mettre à jour non seulement ces statistiques mais aussi celles des workers qu'il utilise. Ceci est-fait à travers deux fonctions programmables de l'interface des ordonnanceurs, les *pre-exec* et *post-exec* "hooks". Ces fonctions sont exécutées avant et après l'exécution d'une tâche et permettent de fournir à l'ordonnanceur les temps précis de début et fin d'exécution.

3.2.4 Respect de la topologie de la machine via hwloc

Pour faciliter l'utilisation de tâches parallèles, il est nécessaire de fournir des interfaces claires aux utilisateurs. L'objectif est de permettre la création

de ressources dans le support d'exécution externe, mais aussi de préparer les processus légers du support d'exécution interne si nécessaire.

Pour ce faire, une interface de création de groupes de cœurs est fournie aux utilisateurs. Cette interface permet de définir une partition de la machine en des groupes de cœurs compactes du point de vue de la hiérarchie mémoire. En pratique, cette interface est réalisée à l'aide de `hwloc` [29], qui fourni la topologie de la machine. L'interface permet dans son utilisation la plus simple de regrouper tous les cœurs situés sous un niveau précis de la topologie (par exemple aux niveaux Socket, NUMA, cache L2, ...). Plusieurs paramètres optionnels peuvent être passés pour définir un découpage plus précis de la machine. Cette interface s'occupe aussi d'initier l'équipe de processus légers associée à la partition décidée par l'utilisateur dans le cas de groupes de cœurs "maître-esclave", soit selon des types prédéfinis (*e.g.* OpenMP, avec ou sans MKL, ...), soit avec une fonction définie par l'utilisateur pour définir la mobilisation des ressources du support d'exécution interne. Cette interface permet aussi automatiquement d'identifier les accélérateurs de la machine et les cœurs utilisés pour les gérer par StarPU, afin de ne créer aucune surexploitation de processus légers sur les ressources.

```
starpu_clusters *clusters;
/* ... */
clusters = starpu_cluster_machine(
            HWLOC_OBJ_NUMANODE,
            STARPU_CLUSTER_TYPE, GNU_OPENMP_MKL,
            0);
/* Parallel tasks submission and computation */
START_TIMING();
MORSE_zpotrf_Tile(uplo, descA);
STOP_TIMING();
/* Come back to usual mode */
starpu_uncluster_machine(clusters);
```

FIGURE 3.9 – Exemple d'utilisation de l'interface de création de groupes de cœurs sur une machine avec 24 CPUs et 4 GPUs.

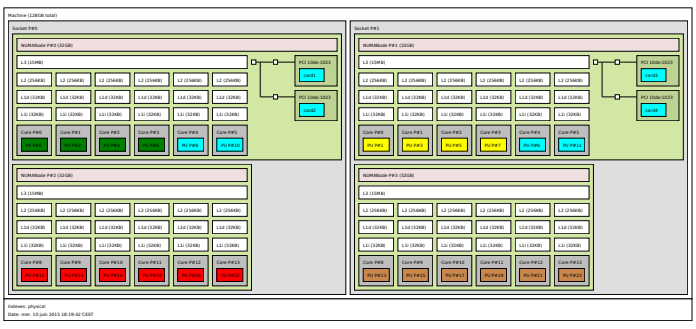


FIGURE 3.10 – Résultat de la création de groupes de cœurs en prenant en compte la topologie de la machine.

La Figure 3.10 montre le résultat pratique de la création de groupes de cœurs avec le code 3.9. L'exemple de code montré est l'adaptation réalisée pour la bibliothèque d'algèbre linéaire dense `Chameleon` utilisée dans la section expérimentale. La machine utilisée en exemple possède 2 CPUs de 12 cœurs chacun et est composée de 4 nœuds NUMA de 6 cœurs. Puisque cette machine a aussi 2 GPUs par socket, StarPU réserve un cœur par GPU (en cyan dans la figure) ce qui implique que seul 20 CPUs sont disponible pour créer des groupes de cœurs. En se basant sur hwloc, il est possible d'analyser la topologie de la machine et regrouper tous les cœurs sous chaque nœud NUMA dans un groupe de cœurs. Les groupes de cœurs rouge, vert, jaune, et marrons sont ainsi créés et possèdent 4 ou 6 cœurs selon le positionnement des workers GPU de StarPU. Puisqu'un type déjà fourni de groupes de cœurs OpenMP est utilisé, les processus légers du support d'exécution OpenMP sont initialisés correctement pendant l'appel à l'interface.

3.3 Évaluation expérimentale

L'agrégation de ressources est évaluée ici à l'aide de la factorisation de Cholesky, une méthode très répandue en algèbre linéaire. Cet algorithme est présent dans plusieurs bibliothèques d'algèbre linéaire qui représentent les calculs comme un graphe orienté acyclique de tâches [7, 27]. Dans la Figure 3.11 est présenté le graphe des tâches d'une factorisation de Cholesky sur une matrice carrée contenant 5×5 tuiles. Le parallélisme disponible avec la factorisation de Cholesky dépend de la largeur du graphe. Ainsi, la factorisation de Cholesky a des propriétés intéressantes car ce degré de parallélisme disponible évolue au cours de l'exécution avec des parties très critiques au début et à la fin du graphe. De plus, le chemin critique de la factorisation de Cholesky est identifié comme la chaîne principale contenant toutes les tâches POTRF, cependant celui-ci peut évoluer au cours des décisions dynamiques d'ordonnancement et passer par certains GEMM si ceux-ci sont exécutés trop tard. Enfin, la factorisation de Cholesky est un benchmark de référence dans la communauté des supports d'exécution et obtenir de meilleures performances sur cet algorithme devient ainsi un réel défi.

Pour notre évaluation, nous utilisons la factorisation de Cholesky de `Chameleon` [3] [1], une bibliothèque d'algèbre linéaire dense pour plate-formes hétérogènes proposant l'utilisation de plusieurs supports d'exécution de façon transparent, tels que StarPU, PaRSEC et QUARK.

Comme toute bibliothèque d'algèbre linéaire à base de tâches, `Chameleon` dépend de la disponibilité d'une bibliothèque fournissant une implantation optimisée des noyaux BLAS.

Notre adaptation de `Chameleon` ne change pas le parallélisme à base de

1. https://project.inria.fr/chameleon/

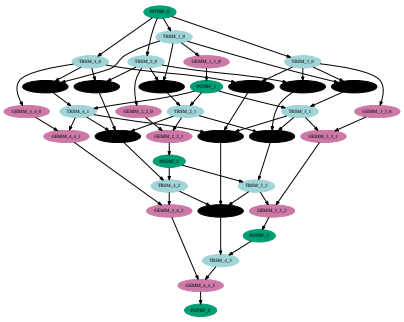


FIGURE 3.11 – Graphe de tâches d'une factorisation de Cholesky d'une matrice carrée ayant cinq tuiles dans chaque dimension.

tâche à haut niveau et le graphe de tâches sous-jacent. Nous implantons seulement comme expliqué en Section 3.2.2 le prologue de chaque tâche de `Chameleon` pour rendre possible l'utilisation de `Intel MKL` basée sur OpenMP (la `Intel MKL` parallèle) à l'intérieur des groupes de cœurs dont on gère la création. Dans toute cette partie, cette version adaptée de `Chameleon` supportant l'utilisation de tâches parallèles sera appelée `pt-Chameleon`.

L'objectif de cette étude expérimentale est de démontrer que l'agrégation de ressources permet non seulement de s'attaquer au problème de granularité exposé précédemment (notamment sur les machines hétérogènes), mais aussi propose plus de flexibilité pour exploiter efficacement des machines complexes. Ceci tout en facilitant la création d'une pile de logiciels spécialisés, dans ce cas en combinant deux supports d'exécutions et deux bibliothèques d'algèbre linéaire dense.

Dans le cadre de cette section, il sera considéré uniquement le cas où nous utilisons des groupes de cœurs de cœurs de CPU de même taille fixés pour toute la durée de l'exécution. La notation $n \times m$ sera utilisée pour désigner les configurations de groupes de cœurs utilisées, ce qui signifie que n groupes de cœurs de m cœurs chacun sont créés.

Toutes les expériences sont conduites sur des machines de la plateforme d'expérimentations scientifiques PlaFRIM [2]. Deux machines sont utilisées en particulier :

1. un Intel Xeon Phi Knights Landing (KNL),
2. une machine hétérogène moderne équipée de 24 cœurs et 4 GPUs.

Pour chaque machine, le plan d'expérience est le suivant.

(a) Étudier les différents noyaux (*i.e.* opérations basiques) qui seront utilisés et notamment leur efficacité en les utilisant en parallèle.

2. https://plafrim.bordeaux.inria.fr/

(b) Comparer de façon exhaustive les différentes versions de `Chameleon` et de `pt-Chameleon`.

(c) Détailler à travers de nouvelles expériences les phénomènes observables dans chaque version afin de distinguer leurs propriétés.

(d) Synthétiser les résultats précédents à travers un graphique comparant chaque version en utilisant les meilleurs paramètres pour chaque point.

3.3.1 Évaluation expérimentale sur la machine `Intel KNL`

Les expériences de cette partie sont réalisées sur une machine avec un processeur Intel Xeon-Phi de nom de code KNL (Xeon Phi 7210). C'est une machine de type homogène avec 64 cœurs à 1.3 GHz, chacun possédant 4 Hyper-Threads. La machine consiste en 32 tuiles de 2 cœurs, chaque tuile partage 1 MB de cache L2 et chaque cœur possède en plus 32 KB de cache L1. Ce Xeon Phi est utilisé comme un processeur et dispose de 16 GB de mémoire embarquée, ainsi que 192 GB de mémoire RAM externe. Le matériel de cette machine est configurable de deux façons : 1) pour créer de la localité et des régions NUMA, 2) pour utiliser la mémoire embarquée comme un cache ou comme une mémoire addressable externe. Pour ces expériences, le Xeon-Phi est configuré en SNC-4 pour créer 4 groupes de 16 cœurs et 4 nœuds NUMA. Le mode cache est aussi utilisé pour la mémoire embarquée qui agit ainsi comme un cache L3. Après plusieurs expériences en discussions avec Intel, ces paramètres semblent les plus efficaces pour les expériences envisagées. La technologie HyperThreading ne proposant pas d'améliorations pour les noyaux BLAS 3 étudiés, celle-ci est désactivée pour les expériences. Pour le support d'exécution StarPU, l'ordonnanceur utilisé est l'ordonnanceur "ws" qui est une variante de l'ordonnanceur à vol de travail avec localité introduit dans le papier [14]. Cet ordonnanceur est réputé comme efficace pour les machines homogènes.

Dans un premier temps une étude des profiles de performance des différents noyaux intervenants dans la factorisation de Cholesky est proposée afin de sélectionner des configurations intéressantes de groupes de cœurs pour `pt-Chameleon`. Dans un second temps, une étude pratique puis plus détaillée des versions présentées est réalisée. Une dernière étude est proposée pour comparer `Chameleon` et `pt-Chameleon` avec deux bibliothèques de référence, *i.e.* `PLASMA` et la versions parallèle de la `Intel MKL` en utilisant tous les cœurs de la machine.

Performance des noyaux composant la factorisation tuilée de Cholesky

Dans la Table 3.1 les efficacités des quatre noyaux composant la factorisation dense de Cholesky à base de tâche est représenté avec une variation du nombre de processus légers utilisés sur le `Intel KNL`. Les résultats sont montrés

	DPOTRF					DTRSM				
	320	480	960	1440	2880	320	480	960	1440	2880
1 cœurs (GFlop/s)	5.372	8.818	14.695	18.646	23.515	20.544	22.837	27.554	28.614	30.226
2 cœurs / 1 cœurs / 2	0.79	0.72	0.84	0.78	0.90	0.76	0.74	0.83	0.87	0.92
4 cœurs / 1 cœurs / 4	0.53	0.51	0.71	0.70	0.75	0.55	0.63	0.77	0.83	0.88
8 cœurs / 1 cœurs / 8	0.32	0.30	0.52	0.53	0.72	0.43	0.43	0.58	0.70	0.84
16 cœurs / 1 cœurs / 16	0.10	0.19	0.31	0.39	0.52	0.26	0.30	0.44	0.55	0.69
32 cœurs / 1 cœurs / 32	0.02	0.07	0.16	0.27	0.37	0.14	0.17	0.29	0.43	0.60

(a) DPOTRF et DTRSM.

	DSYRK					DGEMM				
	320	480	960	1440	2880	320	480	960	1440	2880
1 cœur (GFlop/s)	18.548	21.384	25.725	27.207	29.400	24.426	26.703	29.339	29.872	30.856
2 cœurs / 1 cœur / 2	0.65	0.82	0.88	0.91	0.95	0.82	0.88	0.92	0.86	0.87
4 cœurs / 1 cœur / 4	0.48	0.67	0.78	0.83	0.90	0.73	0.74	0.86	0.89	0.92
8 cœurs / 1 cœur / 8	0.40	0.54	0.68	0.76	0.85	0.47	0.70	0.76	0.89	0.92
16 cœurs / 1 cœur / 16	0.26	0.39	0.53	0.62	0.74	0.34	0.46	0.71	0.82	0.90
32 cœurs / 1 cœur / 32	0.14	0.21	0.28	0.37	0.58	0.34	0.33	0.52	0.69	0.86

(b) DSYRK et DGEMM.

TABLE 3.1 – Efficacité des quatre noyaux composant la factorisation de Cholesky en les exécutant seuls sur la machine `Intel KNL` avec les tailles de tuile 320, 480, 960, 1440 et 2880.

avec plusieurs tailles (320, 480, 960, 1440, 2880) représentants les tailles de bloc considérées de l'algorithme à base de tâche. Ces mesures sont réalisées en utilisant la `Intel MKL` s'exécutant seule et avec un minimum de perturbations sur la machine. On remarque en premier que plus la taille des données en entrée des noyaux augmente, plus ceux-ci deviennent efficaces. Ceci est inhabituel : la performance maximale des noyaux de la `Intel MKL` est usuellement atteinte avec des tailles de données raisonnables (*e.g.* 320 ou 480) sur un cœur de machine courante pour le calcul haute performance comme les Xeons. Deuxièmement, l'efficacité des noyaux parallèles est limitée en utilisant un large nombre de cœurs avec une taille de données de l'ordre de 320 ou 480. Cependant, pour un nombre intermédiaire de cœurs (*i.e.* 8 et 16), l'efficacité parallèle obtenue est satisfaisante pour des tailles de données assez large de 1440 ou plus (le noyau DGEMM a une efficacité parallèle proche de 0.9). Elle est aussi acceptable pour des tailles de données de 960 avec ces mêmes nombres Enfin, on remarque aussi que le DGEMM et le DSYRK ont une meilleur efficacité parallèle ainsi qu'une meilleure scalabilité que le DTRSM et le DPOTRF. Ces résultats laissent penser que les combinaisons de paramètres pour `pt-Chameleon` seront de combiner de grosses tailles de tuile avec un nombre intermédiaire de cœurs par tâche (*e.g.* 8 ou 16) pour obtenir une bonne performance sur le `Intel KNL`. De plus, pour les problèmes plus petits, l'utilisation de plus petites tailles de tuile et d'un petit nombre de cœurs (4 ou moins) par tâche peut être plus intéressant.

Afin de compléter cette étude, une série d'expériences a été réalisée afin d'observer la performance du noyau DGEMM en présence d'un bruit artificiel

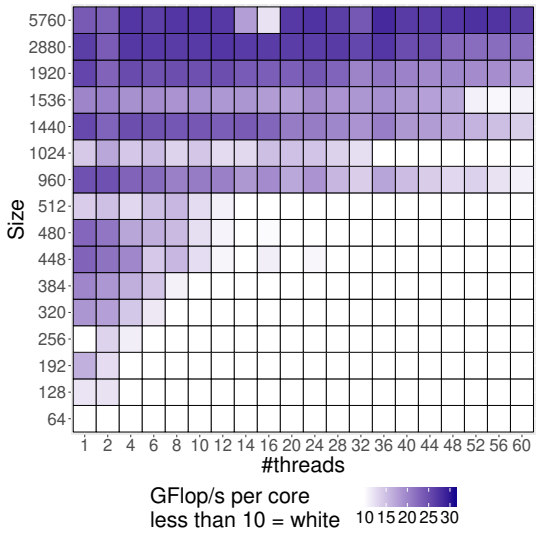


FIGURE 3.12 – Performance par cœur observée avec le noyau DGEMM en créant un bruit artificiel sur la machine `Intel KNL`. Les tailles de tuiles et la quantité de processus légers pour exécuter les noyaux sont explorés. Pour améliorer la lisibilité, toutes les tuiles avec moins de 10 GFlop/s sont affichées en blanc.

sur la machine. Pour cela, le même appel au noyau DGEMM est dupliqué sur des groupes de cœurs de la même taille que celui mesuré, ce jusqu'à ce que la machine soit pleine. Considérons par exemple la performance du noyau DGEMM sur un cœur. Pour cette expérience, chacun des 64 cœurs de la machine exécute alors différents appels au noyau DGEMM avec les mêmes paramètres. Si l'on considère l'exécution du noyau DGEMM avec 6 cœurs, 10 groupes de 6 cœurs et le dernier groupe de 4 cœurs exécutent des appels similaires à DGEMM afin de remplir complètement la machine. Les résultats correspondants sont montrés sur la Figure 3.12 où est représenté la performance du noyau avec une variation de la taille des données ainsi que du nombre de processus légers par noyau. Cette figure confirme tout-d'abord les observations réalisées à partir de la Table 3.1. En effet, on observe que la performance atteinte sur un cœur est de l'ordre de 25 GFlop/s à partir d'une taille de données de 480. Afin d'obtenir une meilleure performance par cœur et se rapprocher de 30 GFlop/s, il est nécessaire d'augmenter le rang de la matrice jusqu'à 2880. Ce graphique montre que à ces tailles de données, on ne perd pas ou très peu de performance en augmentant le nombre de processus légers utilisés pour exécuter le noyau DGEMM. Enfin, il est important de noter la sensitivité des ressources à la contention induite par l'ajout de bruit sur notre plateforme lorsque l'on utilise un cœur par noyau, ce qui représente une exécution possible d'un support à base de tâches ne faisant pas utilisation de tâches parallèles. Par exemple, la performance du noyau DGEMM pour une matrice de taille 320 passe de 24.426

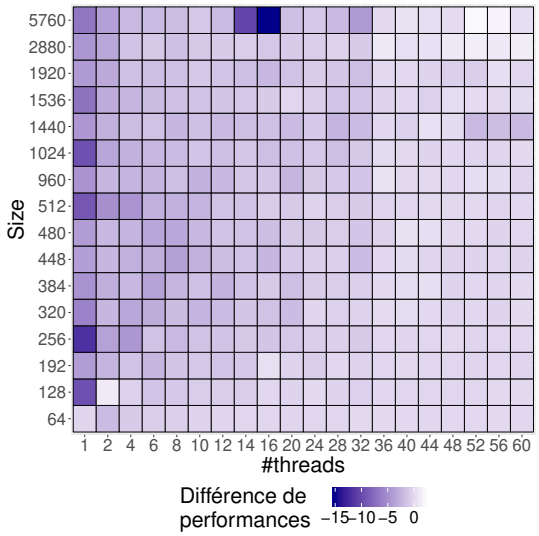


FIGURE 3.13 – Différence de performance par cœur observée avec le noyau DGEMM *sans_bruit* − *avec_bruit* artificiel sur la machine `Intel KNL`. Les tailles de tuiles et la quantité de processus légers pour exécuter les noyaux sont explorés.

GFlop/s dans la Table 3.1 à 18.591 GFlop/s dans le contexte bruité de cette expérience. Une observation similaire peut être faite pour toutes les tailles de tuiles lorsque l'on utilise un cœur en contexte bruité. Pour confirmer cela, la Figure 3.13 montre la différence entre la performance obtenue en contexte bruité et la performance obtenue lorsque le noyau DGEMM tourne seul. Il est alors apparent que la perte est plus significative pour la colonne représentant l'utilisation d'un cœur par noyau.

Performances de `Chameleon` et `pt-Chameleon`

Dans la Figure 3.14 se trouvent les performances en GFlop/s de la factorisation de Cholesky de la bibliothèque `Chameleon` avec plusieurs tailles de tuile. On observe ici que de petites tailles de tuiles donnent de bonnes performances avec de petites tailles de matrices, mais rapidement un plateau est atteint. Par exemple, la performance de `Chameleon` avec une taille de tuile 320 est meilleure que celle avec une taille de tuile 480 jusqu'à atteindre une matrice de rang 11K. où l'on observe un plateau qui s'établit vers 1100 GFlop/s. Ceci est dû à la relativement mauvaise performance des noyaux comme observé avec la Table 3.1 et la Figure 3.12. Par opposition, pour des matrices de petites tailles il n'y a pas assez de parallélisme pour complètement remplir la machine avec une taille de tuile de 480. Pour ces tailles de matrices, les plus petites tailles de tuiles donnent de meilleurs performances (*i.e.* 240 et 320). Un gain jusqu'à 200

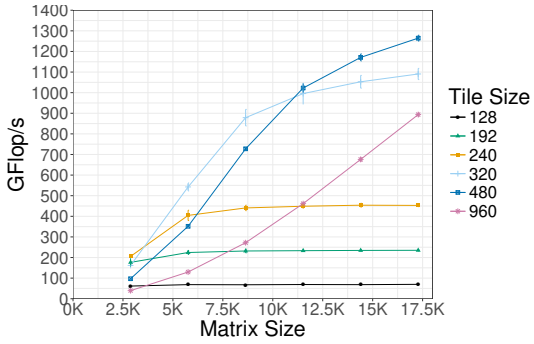


FIGURE 3.14 – Performances de la factorisation de Cholesky avec `Chameleon` et plusieurs tailles de tuile.

GFlop/s peut être observé pour les matrices de rang inférieur à 8.5K comparé à la performance observée avec une taille de tuile de 480.

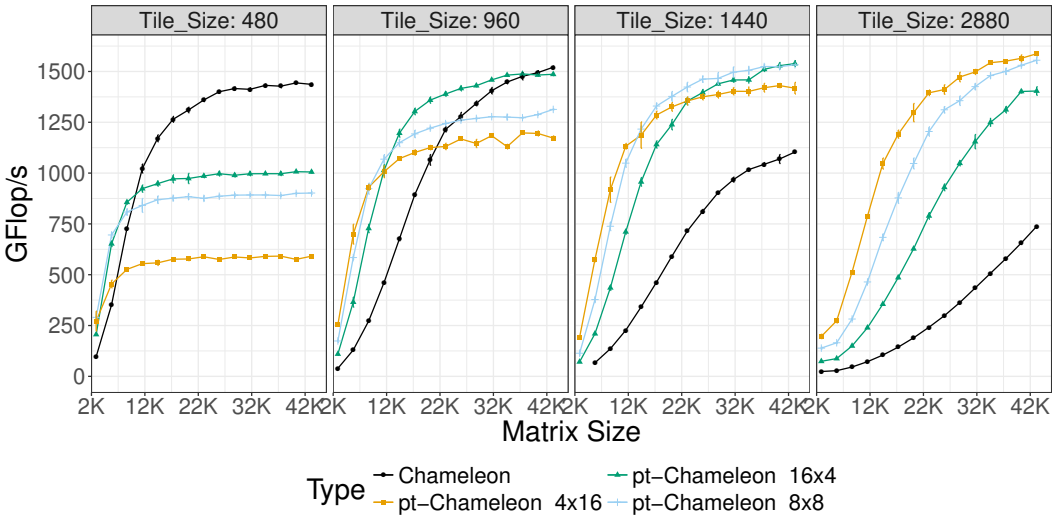


FIGURE 3.15 – Performances de la factorisation de Cholesky de `pt-Chameleon` pour plusieurs configurations de groupes de cœurs et tailles de tuiles.

La Figure 3.15 présente l'évaluation expérimentale de `Chameleon` et `pt-Chameleon` avec la factorisation de cholesky pour plusieurs configurations de groupes de cœurs, tailles de tuiles et tailles de matrices.

Cette figure confirme que pour une taille de tuile de 480, utiliser le parallélisme interne n'est pas très efficace (comme remarqué dans la Table 3.1). Dans ce cas, `Chameleon` et `pt-Chameleon` en configuration 16×4 sont les versions les plus efficaces. Cependant, pour atteindre une meilleure efficacité par noyau il est nécessaire d'augmenter la taille de tuile (*cf.* Table 3.1 et Figure 3.12).

En augmentant la taille de tuile, les configurations basées sur peu de groupes de cœurs de grosses tailles, *i.e.* 8 (*resp.* 4) groupes de cœurs de 8 (*resp.* 16) cœurs chacun, deviennent de plus en plus compétitifs alors que la performance de `Chameleon` se détériore. Cette baisse de performance peut s'expliquer par le fait qu'en augmentant la taille de tuile, la largeur du graphe et donc le parallélisme de celui-ci décroît, ce qui induit plus d'inactivité pour les ressources de la machine (surtout au début et à la fin du calcul de la factorisation de Cholesky, *cf.* le graphe de tâches en Figure 3.11). De plus, on observe que pour `pt-Chameleon` avec une taille de tuile de 2880, la meilleure performance absolue est atteinte pour la configuration 4×16 avec 1587.20 GFlop/s obtenus. Ceci illustre l'intérêt d'utiliser des tâches parallèles et des groupes de cœurs afin de trouver un meilleur compromis entre l'efficacité des noyaux et le degré de parallélisme du graphe de tâches. Globalement, pour un rang de matrice de taille intermédiaire à faible, *i.e.* inférieur à 12K, la version utilisant une large granularité et une configuration de groupes de cœurs 4×16 est moins efficace que les versions utilisant une taille de tuile plus petite et de plus petites configurations de groupes de cœurs. Ceci est principalement dû au manque de parallélisme induit par l'utilisation de grosses tailles de tuile. Cependant pour ces tailles de matrices on observe que les configurations de groupes de cœurs 8×8 ou 16×4 avec des tailles de tuiles 480 ou 960 sont capables de dépasser les meilleurs configurations de `Chameleon`.

Analyse détaillée des performances

Dans la Figure 3.16 se trouve une comparaison plus détaillée de `Chameleon` et `pt-Chameleon` à travers la mesure des performances de chaque noyau DGEMM pour une exécution donnée de `Chameleon` et `pt-Chameleon`. Il est important de noter que DGEMM est le noyau dominant dans la factorisation de Cholesky pour de larges tailles de matrices. Pour plus de clarté, les résultats montrés ici concernent uniquement une matrice de rang 34560 et les meilleurs configurations de `Chameleon` (taille de tuile 960) et de `pt-Chameleon` (taille de tuile 2880 avec 4 groupes de cœurs de 16 cœurs chacun).

Les résultats de la Figure 3.16 sont présentés à travers un histogramme représentant le pourcentage de tâches DGEMM s'exécutant à une certaine performance (GFlop/s). Pour faciliter la comparaison, cette performance est rapportée au nombre de cœurs pour `pt-Chameleon`. De plus, deux barres verticales sont présentées montrant la performance du noyau DGEMM d'après Table 3.1 lorsqu'il s'exécute seul sur la machine avec la même taille de tuile en utilisant un ou seize cœurs pour le cas `pt-Chameleon`. Dans cette figure est aussi représenté en gris clair la performance moyenne de chaque exécution. On observe que la moyenne des performances des noyaux DGEMM sur `pt-Chameleon` est plus grande de 4.8% que la moyenne des performances des DGEMM sur `Chameleon`. De plus, en regardant la distribution de chaque histogramme on

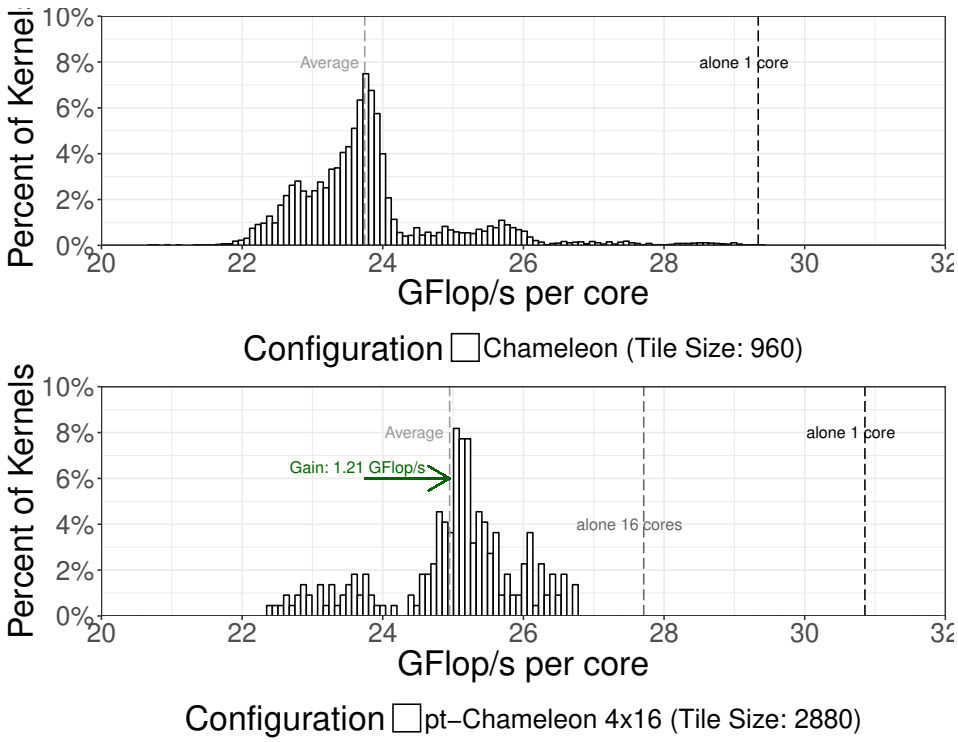


FIGURE 3.16 – Comparaison des performance du noyau DGEMM avec les meilleurs configurations de `Chameleon` et `pt-Chameleon` avec une matrice de rang 34560.

constate que la performance de `pt-Chameleon` est bien plus stable que celle de `Chameleon` ainsi que plus proche de sa référence avec 16 cœurs. Ces observations donnent de premiers arguments pour comprendre comment `pt-Chameleon` surpasse `Chameleon`.

Puisque `pt-Chameleon` utilise moins de ressources (plus grosses) et du fait du partage de travail, on s'attend à ce que `pt-Chameleon` soit plus conservateur en terme d'utilisation des ressources. En effet, l'utilisation de la `Intel MKL` parallèle et d'OpenMP dans notre cas améliore l'utilisation de la hiérarchie mémoire et la localité des données des noyaux de la factorisation de cholesky tuilée. La Figure 3.17 représente le nombre total de défaut de cache L2 pour `Chameleon` et `pt-Chameleon` (en configuration 4×16) avec une matrice de rang 34560 et différentes tailles de tuile. À cause d'un accès limité sur le `Intel KNL` à plusieurs compteurs de performance, il n'est pas possible de présenter des résultats pour la MCDRAM ni de montrer des taux d'accès au cache. On observe ici grâce à cette figure que le nombre absolu de défaut de cache pour `Chameleon` est au moins deux foins plus grand que `pt-Chameleon` quelque soit la taille de tuile utilisée. Cette meilleure utilisation des ressources par `pt-Chameleon` est un autre argument permettant d'expliquer les meilleurs

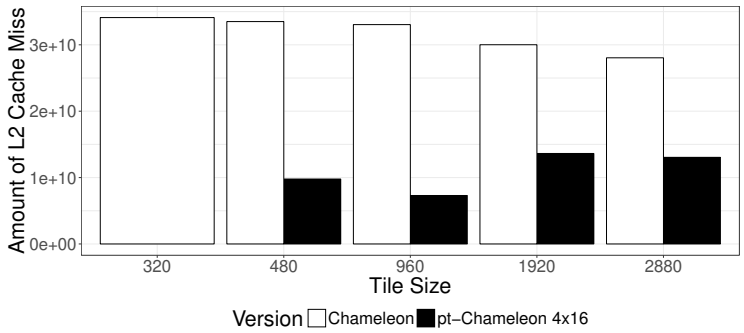


FIGURE 3.17 – Défauts de cache L2 absolus observés avec la factorisation de Cholesky sur le `Intel KNL` avec `pt-Chameleon` et `Chameleon` pour une matrice de rang 34560.

performances observées.

Comparaison avec `Intel MKL` et `PLASMA`

Pour finir cette étude sur le `Intel KNL`, la Figure 3.18 compare les meilleurs configurations obtenues précédemment de `Chameleon` et `pt-Chameleon` à `Intel MKL` et `PLASMA`. Il n'est pas possible de se comparer avec la bibliothèque `MAGMA` pour le `Intel KNL` (*cf.* [63] pour plus de détails) car la version `Intel KNL` de `MAGMA` n'est pas disponible.

Pour plus de clarté, pour `Chameleon`, `pt-Chameleon` et `PLASMA` seuls les meilleurs performances pour chaque taille de matrice sont présentées parmi les tailles de tuiles et groupes de cœurs présentés dans les Figures 3.15 et 3.14. La courbe `Intel MKL` quand-à consiste en un simple appel a la routine DPOTRF avec la matrice entière en entrée sur toute la machine.

On observe que la performance de `Chameleon` est supérieure à `PLASMA` pour toutes les tailles de matrice avec une amélioration de moins de 200 GFlop/s pour de larges tailles de matrice. Pour des matrices de rang d'environ 10K à 20K, `PLASMA` a une performance plus proche de `Chameleon` avec une différence de moins de 100 GFlop/s.

En ce qui concerne la courbe de référence représentée par `Intel MKL`, on observe qu'elle obtient de meilleurs performances que toutes les autres versions pour des problèmes de taille inférieure à 12K. Cependant à partir de matrices de rang 12K, `pt-Chameleon` a une performance similaire ou supérieure à celle de `Intel MKL` et plus stable (nous n'avons pas de raisons concrète pour expliquer les nombreux défauts de la courbe `Intel MKL`). `Chameleon` est aussi égal ou légèrement supérieur à la `Intel MKL` à partir de matrices de rang environ 25K. `pt-Chameleon` obtient une performance absolue de 1587.20 GFlop/s pour une matrice de taille 43K ce qui représente une amélioration de 5.4% par rapport à la `Intel MKL` et 70 GFlop/s de plus que `Chameleon`. Pour des tailles de

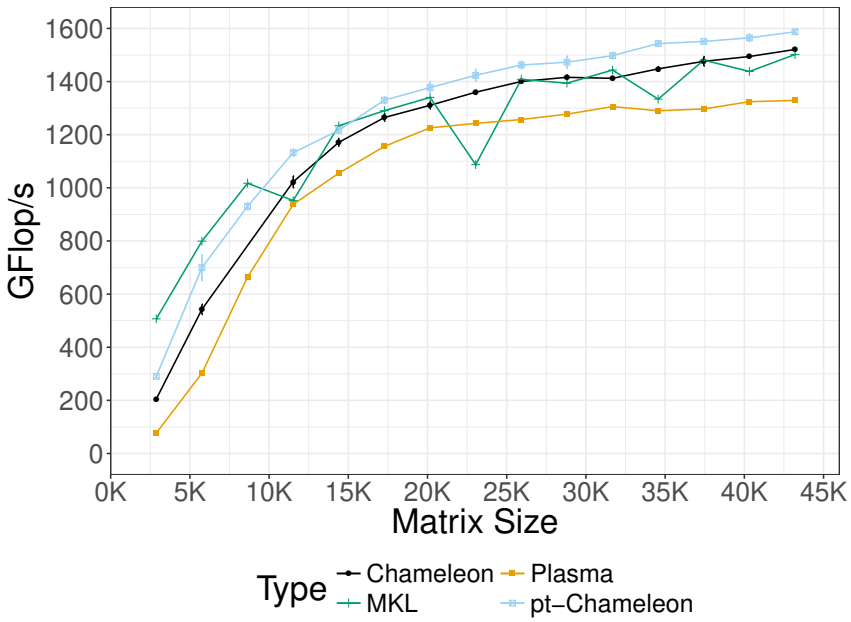


FIGURE 3.18 – Comparaison de la performance de la factorisation de Cholesky sur le `Intel KNL` avec `pt-Chameleon`, `Chameleon`, `PLASMA` et `Intel MKL`.

matrices de rang faible à intermédiaire, `pt-Chameleon` obtient la performance la plus proche de la `Intel MKL` et supérieure à `Chameleon` et `PLASMA`.

Ces résultats illustrent l'importance d'étudier les trade-offs entre le parallélisme interne aux tâches et le parallélisme externe sur des machines basées sur des processeurs "many-core". Cette approche permet d'utiliser des tâches à gros grain et ainsi une meilleure performance de base tout en ne perdant pas trop de performance à cause du manque de parallélisme. De plus, ces résultats montrent comment StarPU est capable d'utiliser la `Intel MKL` parallel et OpenMP à travers l'utilisation des groupes de cœurs et d'utiliser une spécialisation des supports d'exécution afin d'obtenir de meilleurs performances pour la factorisation de Cholesky.

3.3.2 Étude expérimentale sur une machine hétérogène

Pour les expériences de cette partie, la machine utilisée est hétérogène et composée de deux processeurs 12-cœurs Intel Xeon E5-2680 v3 (@2.5 GHz équipés de 30 MB de cache chacun) ainsi que quatre GPUs NVidia K40m. Dans StarPU, un cœur est dédié à chaque GPU, par conséquent pour `Chameleon` et `pt-Chameleon` les résultats seront présentés avec 20 cœurs pour les CPUs. La configuration utilisée pour `pt-Chameleon` pour toutes ces expériences est 2×10, ainsi chaque groupe de cœurs est composé de 10 cœurs. En ce qui concerne le support d'exécution StarPU, l'ordonnanceur utilisé est la version

adaptée de l'ordonnanceur MCT introduit en Section 3.2.3.

Enfin, pour toutes les figures la performance montrée est la moyenne observée et la déviation standard de ces performances sur 20 à 5 itérations (selon la taille du problème) avec des matrices carrées.

Performance des noyaux composant la factorisation tuilée de Cholesky

	DPOTRF		DTRSM		DSYRK		DGEMM	
	960	1920	960	1920	960	1920	960	1920
1 cœur (GFlop/s)	27.78	31.11	34.42	34.96	31.52	32.93	36.46	37.27
GPU / 1 cœur	1.72	5.95	8.72	18.59	26.96	31.73	28.80	30.86
10 cœurs / 1 cœur	5.55	7.48	6.75	8.48	6.90	8.63	7.77	8.56

TABLE 3.2 – Facteurs d'accélération des noyaux de la factorisation de Cholesky sur un GPU et sur 10 cœurs par rapport à un cœur avec les tailles de tuile 960 et 1920.

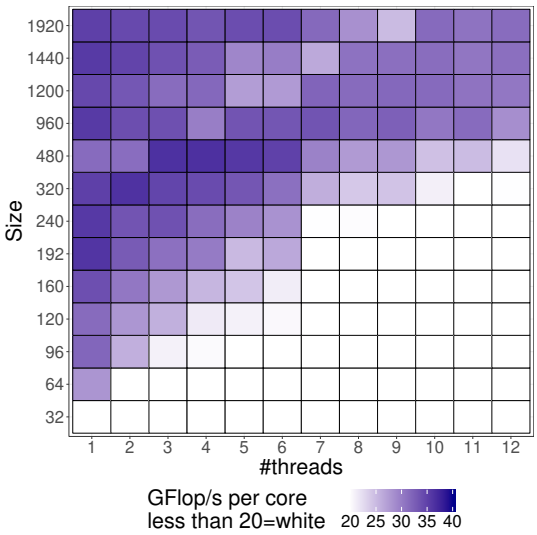


FIGURE 3.19 – Performance observée par cœur pour le noyau DGEMM en créant un bruit artificiel sur les CPUs. Les tailles de tuiles et la quantité de processus légers pour exécuter les noyaux sont explorés. Pour améliorer la lisibilité, toutes les tuiles avec moins de 20 GFlop/s sont affichées en blanc.

La Table 3.2 contient les facteurs d'accélération en utilisant 10 cœurs ou un GPU comparés à la performance d'un seul cœur pour chaque noyau de la factorisation de Cholesky. Cette évaluation a été réalisée avec la `Intel MKL`

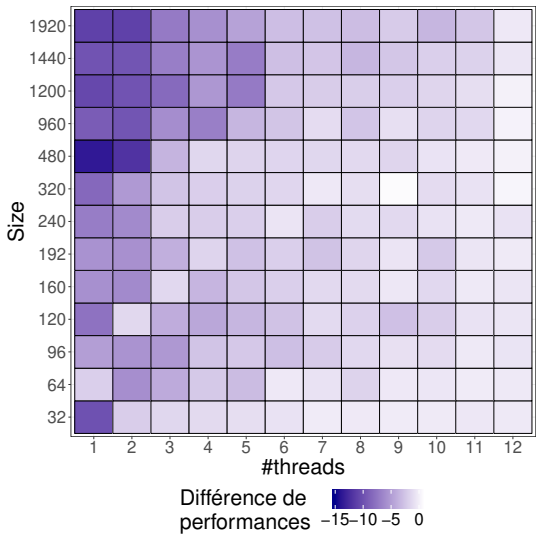


FIGURE 3.20 – Différence de performance par cœur observée avec le noyau DGEMM *sans_bruit* − *avec_bruit* artificiel sur les CPUs de la machine hétérogène. Les tailles de tuiles et la quantité de processus légers pour exécuter les noyaux sont explorés.

pour les CPUs et CuBLAS (*resp.*`MAGMA`) pour les GPUs. Cette table montre que la scalabilité de l'utilisation de 10 cœurs est sous-linéaire par rapport à l'utilisation d'un cœur. Le meilleur noyau DGEMM a son exécution accélérée par un facteur de 7.77 en utilisant 10 cœur et celui-ci augmente à 8.56 en utilisant une taille de tuile de 1920. Cependant, on observe que utiliser des noyaux séquentiels dégrade l'écart de performance entre CPUs et GPUs alors qu'utiliser des groupes de cœurs rend l'ensemble des ressources plus homogènes. Il est possible d'obtenir un facteur d'accélération du GPU sur la version CPUs utilisés en parallèle en divisant la seconde ligne par la troisième. Ainsi, l'écart de performance du noyau DGEMM avec une taille de tuile de 960 est 28.8 en utilisant 1 cœur comparé à un GPU alors que celui-ci est de $28.80/7.77 \simeq 3.7$ en utilisant 10 cœurs comparé à un GPU. Par conséquent, si 28 DGEMM indépendants de taille 960 sont soumis à une machine de 10 cœurs et un GPU avec ces facteurs d'accélération, l'ordonnanceur de `Chameleon` assigne toutes les tâches sur GPU alors que `pt-Chameleon` assigne 6 tâches au groupe de cœurs de 10 cœurs et 22 tâches sur les GPUs. Un autre aspect important que l'on peut observer sur cette figure est la capacité de `pt-Chameleon` à accélérer le chemin critique. En effet, un groupe de cœurs de 10 cœurs peut exécuter le noyau DPOTRF sur de taille 960 trois fois plus vite que sur un GPU. La performance est aussi assez proche entre l'utilisation de 10 cœurs et un GPU pour le DTRSM.

La Figure 3.19 précise les résultats de la table 3.2. Elle montre la perfor-

mance mesurée du noyau DGEMM sur les CPUs en créant un bruit artificiel sur la machine. La technique utilisée est la même que celle pour la Figure 3.12. L'appel au noyau est dupliqué pour chaque groupe de cœur de la machine jusqu'à ce que la machine soit pleine. Cette figure présente la performance en GFlop/s/core en fonction du nombre de processus légers utilisé pour exécuter le noyau. On remarque que pour un noyau de taille 960 dans ce contexte bruité, la performance du DGEMM est de 35.6 GFlop/s/cœur pour un cœur alors qu'elle est de 30.5 GFlop/s/cœur en utilisant 10 cœurs. De façon similaire, pour un noyau de taille 1920, utiliser un cœur donne 35.1 GFlop/s/core alors qu'utiliser 10 cœurs donne 31.6 GFlop/s/cœur. Ainsi, pour des noyaux de taille 960 et 1920, la perte de performance ne dépasse pas 15% en utilisant 10 processus légers. La Figure 3.20 montre de façon similaire à la Figure 3.13 que la perte de performance entre la version bruitée et la version non bruitée est plus prononcée en utilisant seulement un ou deux cœurs par noyaux.

Bornes de performance pour `Chameleon` et `pt-Chameleon`

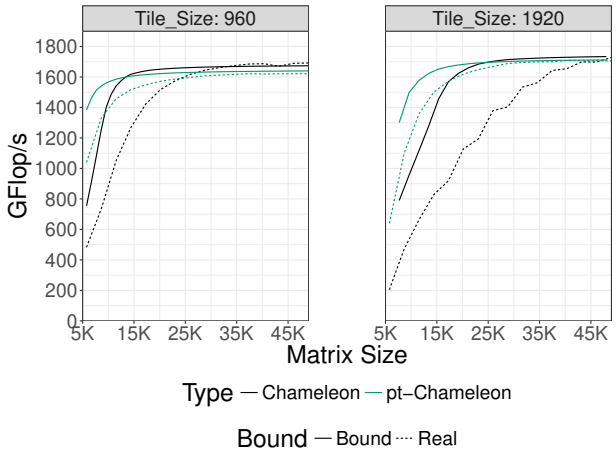


FIGURE 3.21 – Comparaison de la factorisation de Cholesky avec `pt-Chameleon` et `Chameleon` avec leur borne théorique. 20 CPUs et 1 GPU sont utilisés.

La Figure 3.21 montre la performance réelle de la factorisation de Cholesky pour `Chameleon` et `pt-Chameleon` avec des tailles de tuile de 960 et 1920 avec leurs bornes inférieures sur le makespan théorique. Ces bornes sont calculées avec la méthode introduite dans [10] qui calcule une borne en ajoutant itérativement les nouveaux chemins critiques découverts jusqu'à ce qu'ils soient tous pris en compte. Comme ces bornes ne prennent pas en compte les communications avec les GPUs, elles sont clairement inatteignable en pratique. Ces bornes montrent que `pt-Chameleon` peut théoriquement obtenir de meilleures

performance que `Chameleon` sur des matrices de rang faible à intermédiaire. En effet, les CPUs sont sous utilisés dans le cas de `Chameleon` à cause du manque de parallélisme alors que l'utilisation de groupes de cœurs réduite la quantité de tâche nécessaire pour remplir les cœurs des CPUs. De plus comme montré dans [10] l'ordonnanceur MCT peut favoriser d'avantage les GPUs dans le cas `Chameleon` à cause de leur avantage en performance important. Le point avec la matrice de rang 5K montre une différence entre les performance réelles de `Chameleon` et de `pt-Chameleon` de l'ordre de 500 GFlop/s en utilisant une taille de tuile de 1920 et de l'ordre de 600 GFlop/s en utilisant une taille de tuile de 960. Ceci est proche de la performance disponible sur ces CPUs (comme montré par la Table 3.2), ce qui suggère que `pt-Chameleon` utilise tous les CPUs et GPUs alors que `Chameleon` utilise principalement les GPUs. Pour les deux tailles de tuile avec de grandes matrices (*e.g.* 40K), la borne de `Chameleon` est supérieure à celle de `pt-Chameleon`. Ceci est dû à la meilleure efficacité des noyaux séquentiels puisque les noyaux parallèles ne possèdent pas une scalabilité parfaite. On observe qu'avec des grains de 1920, la performance maximale atteignable est plus grande, principalement grâce à la meilleure efficacité des noyaux sur GPUs avec ces tailles de tuile. Pour le noyau DGEMM l'utilisation de cette taille plutôt que 960 permet de gagner près de 100 GFlop/s sur un GPU (ou 10%). On remarque aussi que l'écart entre les bornes de `Chameleon` et `pt-Chameleon` décroît légèrement en augmentant la taille de tuile jusqu'à 1920 grâce à la meilleure efficacité par cœur des noyaux sur les CPUs avec cette taille de tuile. On remarque que les exécutions réelles sont toujours sous leur borne théorique, ce qui est normal puisque les transferts GPU-RAM ne sont pas pris en compte dans le calcul des bornes. Enfin, cette figure montre une performance supérieure de `pt-Chameleon` sur `Chameleon`, notamment avec des matrices de taille 11K et une taille de tuile de 960, `pt-Chameleon` obtient un gain de performance de 65% et même jusqu'à 100% de performances pour les matrices de taille inférieure à 10K. Avec des matrices de cette taille, l'exécution réelle de `pt-Chameleon` est au dessus de la borne de `Chameleon` ce qui montre la supériorité de notre approche sur `Chameleon`.

Performances de `Chameleon` et `pt-Chameleon`

On montre dans la Figure 3.22 la performance de `pt-Chameleon` comparé à la version existante `Chameleon` avec plusieurs tailles de tuile et nombres de GPUs. Pour les deux versions on ajoute une version "contrainte" (`Chameleon`-c and `pt-Chameleon`-c) où l'on optimise l'ordonnancement en forçant l'exécution des noyaux DPOTRF et DTRSM sur les CPUs. On observe que `pt-Chameleon` obtient des performances supérieures a `Chameleon` pour tous les cas testés lorsque l'on utilise de petites tailles de matrices. Plusieurs améliorations apportées par `pt-Chameleon` par rapport à `Chameleon` expliquent ce comportement. Dans le cas de `pt-Chameleon`, l'utilisation de tâches parallèles

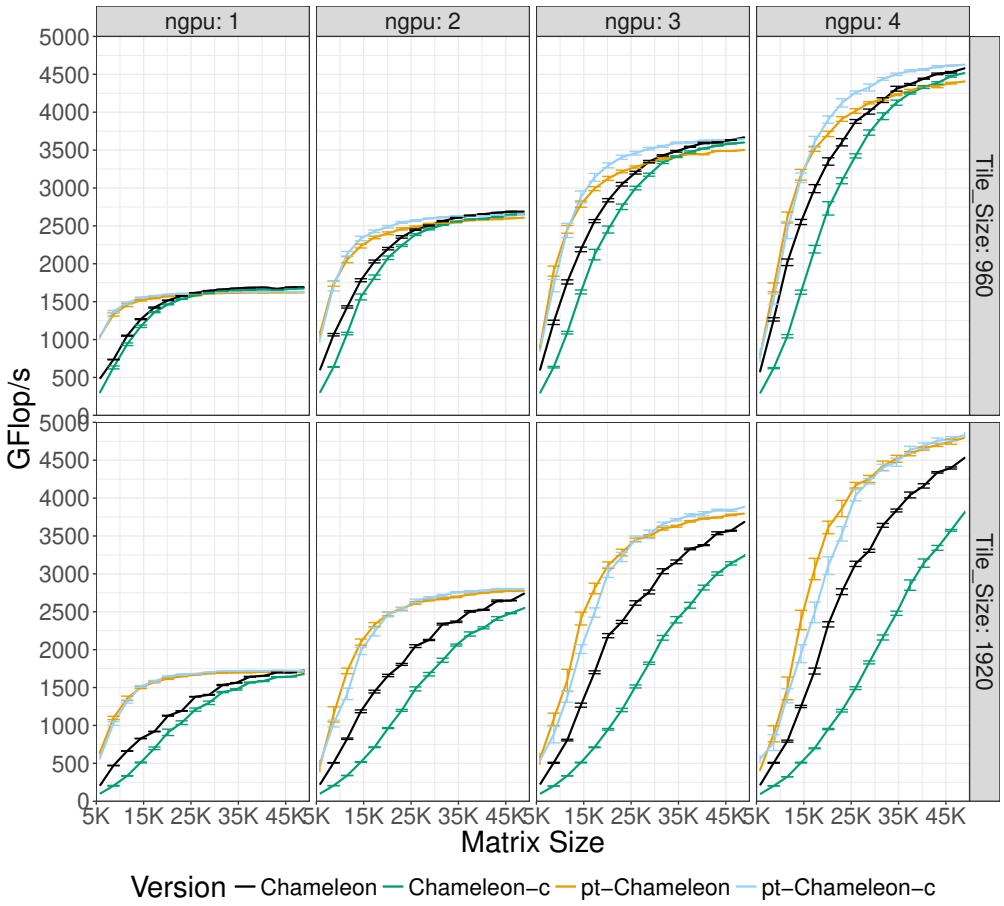


FIGURE 3.22 – Performances de la factorisation de Cholesky de `pt-Chameleon` et `Chameleon` en variant le nombre de GPUs et la granularité des tâches. Pour les deux versions, une version "contrainte" est ajoutée où l'on force l'exécution des noyaux DPOTRF et DTRSM sur les CPUs afin d'optimiser l'ordonnancement.

peut accélérer le chemin critique grâce à une meilleure efficacité des noyaux DPOTRF comme montré dans la Table 3.2. De plus, puisque `pt-Chameleon` a besoin de moins de parallélisme pour remplir ses deux workers CPUs (de 10 cœurs chacun), `pt-Chameleon` a une occupation des CPU supérieure à celle de `Chameleon`. Par exemple, avec une taille de tuile de 960 et 2 GPUs pour une matrice de rang 14400 `Chameleon` (*resp.*`pt-Chameleon`) a un taux d'inactivité des CPUs de 60.59% (*resp.* 5.01%). De façon globale, on remarque que grâce aux versions "contraintes", `Chameleon` comme `pt-Chameleon` sont capables d'augmenter leur performance avec de grandes tailles de matrices. On peut voir que `pt-Chameleon` est moins pénalisé par l'utilisation de la version contrainte surtout si l'on considère les performances avec les tailles de tuile de 1920. Ceci est encore une fois dû au fait que `pt-Chameleon` a besoin de moins

de tâches pour remplir ses CPUs ainsi qu'à la plus faible hétérogénéité entre les CPUs et GPUs. Aussi bien que `Chameleon` que `pt-Chameleon` ont une bonne scalabilité de leur performance lorsque l'on augmente le nombre de GPUs. Par exemple, la meilleure performance avec 1 GPU et une taille de tuile de 960 est de 1.7 TFlop/s et avec 2 GPUs elle est de 2.7 TFlop/s. Cette amélioration est attendue puisque 1 TFlop/s est la performance d'un GPU sur cette plateforme avec un noyau DGEMM de cette taille (comme montré par la Table 3.2). `Chameleon` a une scalabilité légèrement inférieure à `pt-Chameleon` avec une taille de tuile de 1920. Cet écart entre les deux versions augment lorsque l'on augmente le nombre de GPUs. Une première raison pour expliquer ce comportement sont les différences de décisions d'ordonnancement prises. Comme précisé précédemment, à cause du grand facteur d'hétérogénéité, l'ordonnanceur MCT peut favoriser exagérément les GPUs dans le cas de `Chameleon` créant une sous-utilisation des CPUs.

Analyse détaillée des performances

Dans cette partie, on compare plus précisément les comportements de `Chameleon` et `pt-Chameleon` avec une taille de matrice de 48960 et une taille de tuiles de 960 (*resp.* 1920) pour `Chameleon` (*resp.*`pt-Chameleon`) afin de mieux comprendre les gains de performances observés précédemment. Pour rappel, ces paramètres correspondent aux meilleures performances obtenues pour chaque version d'après 3.22.

Premièrement, pour mettre en avant le fait que `pt-Chameleon` permet de trouver de meilleurs compromis entre l'efficacité des noyaux et la quantité de parallélisme dans ce contexte hétérogène, on trouve dans la Figure 3.23 la performance de tous les noyaux DGEMM de la factorisation de Cholesky pour `Chameleon` et `pt-Chameleon`. Pour cette expérience, la même technique que pour la Figure 3.16 est utilisée. Cela consiste en tracer la performance de tous les noyaux DGEMM et montrer sur cette figure le pourcentage de noyaux (en ordonnée) avec une performance donnée (en abscisse). À gauche dans la figure se trouvent les noyaux exécutés sur CPUs et à droite les noyaux exécutés sur GPUs. Pour chaque version, la somme de l'histogramme CPU représente 100% et de façon similaire la somme de l'histogramme GPU représente aussi 100. En plus de l'histogramme, des barres verticales sont fournies pour représenter la performance moyenne de chaque exécution ainsi que la performance de référence du noyau en DGEMM sur CPU en contexte bruité (*i.e.* avec 1 cœur ou 10 cœur).

Ce qu'il est important d'observer dans cette expérience est la différence entre `Chameleon` et `pt-Chameleon` des performances moyennes sur CPU et GPU. En effet, `pt-Chameleon` perd 2.83 GFlop/s/cœur en moyenne pour les noyaux DGEMM exécutés sur CPU, mais pour `pt-Chameleon` l'utilisation d'une taille de tuile plus grosse permet d'obtenir 36.71 GFlop/s/GPU. De plus,

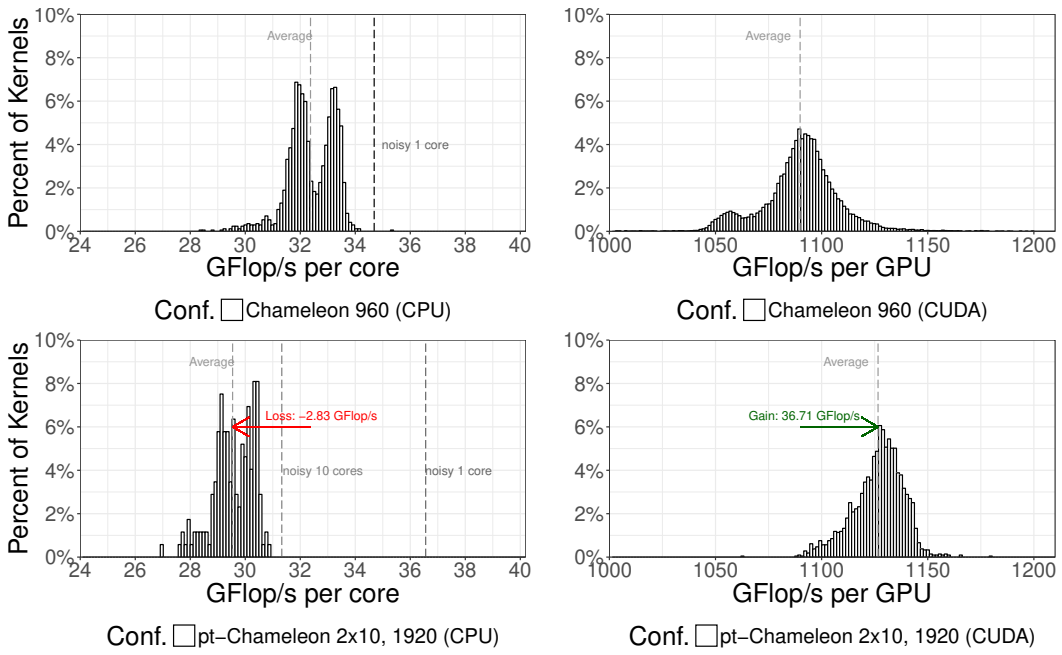


FIGURE 3.23 – Comparaison du comportement du noyau DGEMM avec les meilleurs configurations pour `Chameleon` et `pt-Chameleon` avec une matrice de rang 48960 et 4 GPUs. À gauche se trouve un histogramme des noyaux exécutés sur CPUs, à droite un histogramme des noyaux exécutés sur les GPUs.

92.43% (resp. 91.77%) des noyaux DGEMM de la factorisation de Cholesky sont exécutés sur GPU pour `pt-Chameleon` (resp. `Chameleon`). Ceci illustre que en utilisant une taille de tuile plus importante sans perdre trop de performance sur les CPUs, `pt-Chameleon` est capable de trouver un meilleur compromis entre le grain des calculs, le parallélisme et l'efficacité des noyaux.

La Figure 3.24 montre la quantité de transferts mesurés sur la plateforme (*i.e.* GPU-GPU ou GPU-RAM) lors du calcul de la factorisation de Cholesky pour une matrice de taille 48960. On peut voir que `Chameleon` génère un plus gros volume de communication que `pt-Chameleon`. Par exemple avec 4 GPUs, `pt-Chameleon` génère environ 50 Go de transferts avec une taille de tuile de 1920 alors que `Chameleon` en génère 60 Go avec une taille de tuile de 960. On rappelle aussi que sur cette machine l'outil *nvidia-smi topo -m* montre que tous les transferts, y compris entre GPUs, transitent par le processeur (technologie Intel QPI). Ainsi, la consommation de la bande passante mémoire peut être un autre facteur limitant la performance de la version `Chameleon`.

De plus, une autre explication de la différence de performances observée entre `Chameleon` et `pt-Chameleon` est l'efficacité de chaque version à utiliser la hiérarchie mémoire. Pour élucider ce problème, la Figure 3.25 montre le

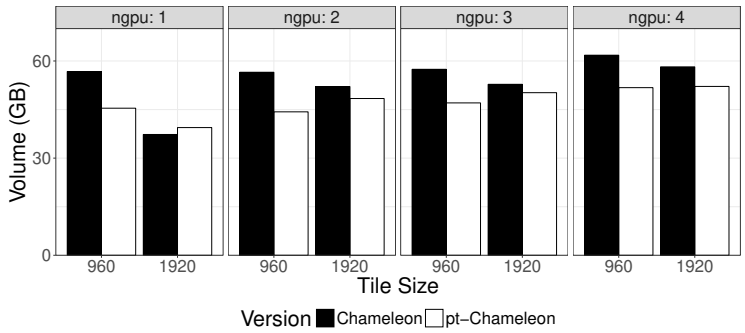


FIGURE 3.24 – Quantité de transferts pour la factorisation de Cholesky avec `pt-Chameleon` et `Chameleon` pour une matrice de rang 48960. Le nombre de GPUs et la granularité des tâches sont explorés.

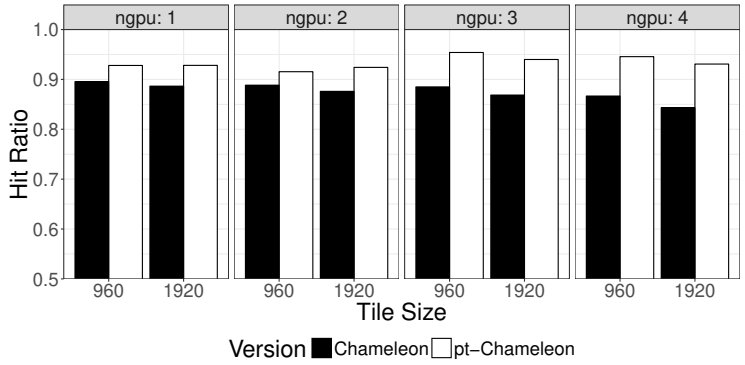


FIGURE 3.25 – Taux d'accès au cache pour la factorisation de Cholesky avec `pt-Chameleon` et `Chameleon` et une matrice de rang 48960. Le nombre de GPUs et la granularité des tâches sont explorés.

taux d'accès au le cache L3 pour la factorisation de Cholesky avec une matrice de rang 48960 pour `Chameleon` et `pt-Chameleon`. On observe que pour les deux tailles de tuile 960 et 1920, le taux d'accès au cache L3 de `pt-Chameleon` est comprise entre 90 et 95% alors que pour `Chameleon` celui-ci est compris entre 85 et 90%. Cette différence pouvant aller jusqu'à 10% est expliquée par l'architecture du processeur. En effet, chaque processeur possède 30 Mo de cache et en utilisant un groupe de cœurs par processeur plutôt que 10 unités de calcul indépendantes on divise par 10 le volume de travail. Puisqu'une tuile de taille 960 pèse 7 Mo alors qu'une tuile de taille 1920 pèse 28 Mo, la version `pt-Chameleon` est même capable de placer entièrement une tuile de taille 1920 dans le cache L3. De fait, `Chameleon` utilisant une matrice de taille 48960 a un volume de communication plus important et un taux d'accès au cache inférieur à `pt-Chameleon` ce qui, mis bout-à-bout peut créer une contention mémoire.

Pour observer ce phénomène une expérience a été menée en allouant la matrice complète sur un des nœuds NUMA de la machine grâce à l'outil numactl. On a pu observé grâce à Intel VTune que pour `Chameleon` 59% des noyaux DGEMM deviennent bornés par la mémoire, alors que pour `pt-Chameleon` seulement 13% deviennent bornés par la mémoire.

Utilisation de la technologie "multi-stream"

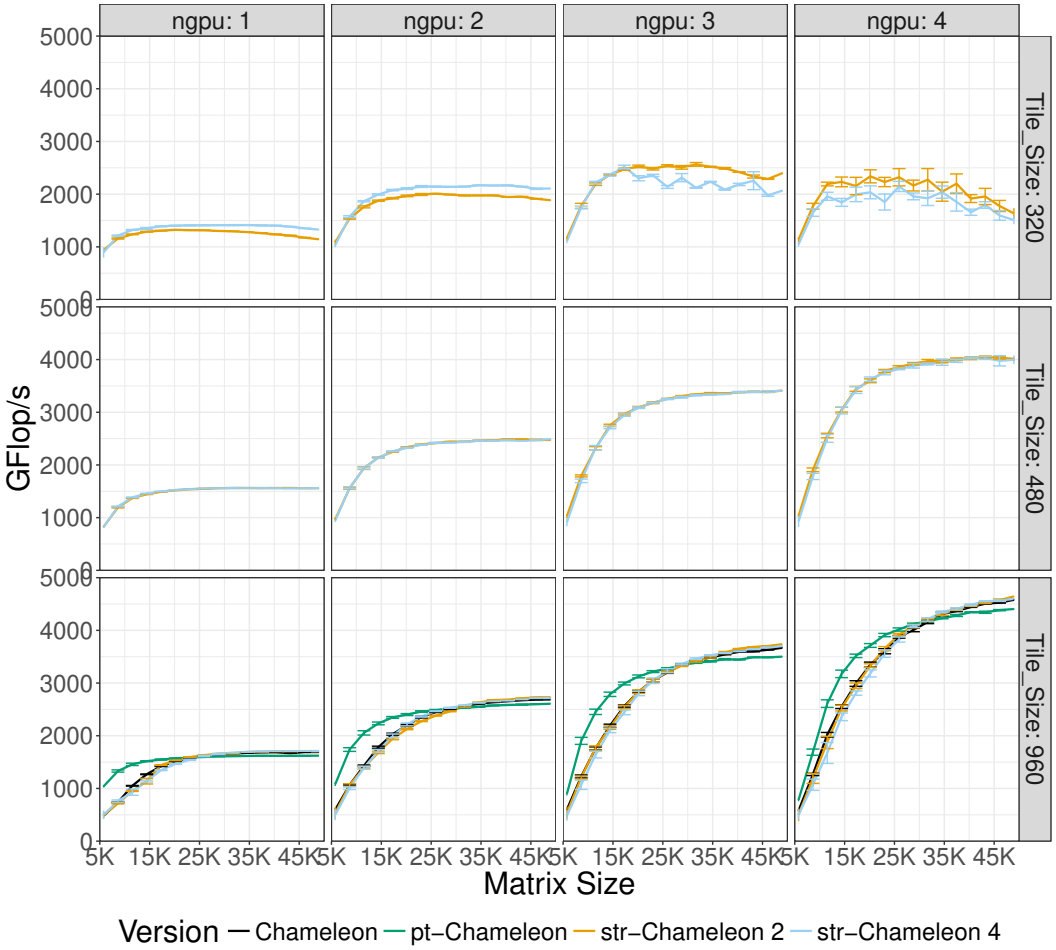


FIGURE 3.26 – Performance de la factorisation de Cholesky avec `pt-Chameleon` et une version de `Chameleon` utilisant la technologie multi-stream sur le GPU. Le nombre de GPUs et la granularité des tâches sont les paramètres explorés.

Dans la Figure 3.26, la performance montré est celle d'une nouvelle version de `Chameleon`, `str-Chameleon` qui utilise plusieurs streams pour chaque GPU NVidia. `str-Chameleon` utilisant 2 et 4 streams est comparé à `pt-Chameleon` et `Chameleon`. L'intérêt d'utiliser le multi-stream est que cela permet

d'exécuter plusieurs noyaux à la fois sur le GPU et permet ainsi d'augmenter l'occupation du GPU en utilisant des tuiles de petites tailles. Les résultats sont donc montrés avec des tailles de tuile de 320, 480 et 960. On observe qu'avec une taille de tuile de 960 il y a très peu de différence entre `Chameleon` et `str-Chameleon`. Pour ces tailles de tuiles, `pt-Chameleon` est capable de dépasser `str-Chameleon` avec de petites tailles de matrices mais aussi bien `Chameleon` que `str-Chameleon` obtiennent de meilleures performances pour de grosses matrices. Avec une taille de tuile de 480, on observe que l'utilisation de streams permet d'obtenir une plutôt bonne performance d'environ 4 TFlop/s avec 4 GPUs pour de grosses matrices. On observe aussi que `str-Chameleon` avec une taille de tuile de 480 et des matrices de tailles intermédiaires obtient une meilleure performance que `Chameleon` et `pt-Chameleon` avec une taille de tuile de 960. Par exemple, avec une matrice de rang $15K$ et 4 GPUs, `str-Chameleon` atteint 3.1 TFlop/s, alors que pour une taille de tuile de 960 `Chameleon` obtient 2.6 TFlop/s et `pt-Chameleon` obtient 3.2 TFlop/s. De façon générale, on observe qu'utiliser le multi-stream et réduire la taille de tuile est une solution intéressante pour les matrices de rang faible à intermédiaire. Cependant, il est important d'ajouter que malgré ces avantages, un problème avec l'utilisation du multi-stream est qu'il de modéliser la performance des noyaux sur GPU : ceci rend le modèle de performance de StarPU moins précis et peut impacter négativement les décisions d'ordonnancement.

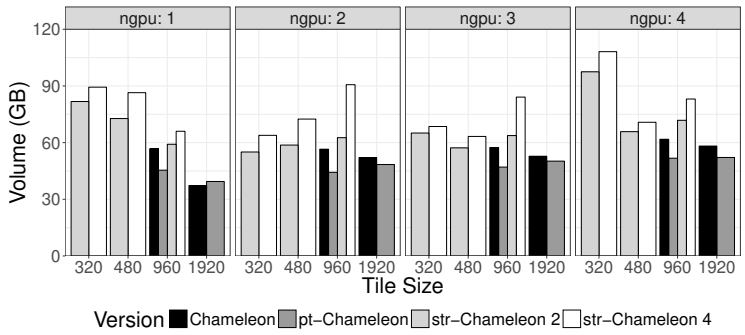


FIGURE 3.27 – Quantité de transferts pour la factorisation de Cholesky avec `str-Chameleon`, `pt-Chameleon` et `Chameleon` pour une matrice de rang 48960. Le nombre de GPUs et la granularité des tâches sont explorés.

La Figure 3.27 montre le volume de transferts de façon similaire à la Figure 3.24. La différence ici est que l'on ajoute les versions de `str-Chameleon` afin d'en observer le comportement. On remarque de façon générale `str-Chameleon` génère un plus gros volume de communications que `Chameleon` et `pt-Chameleon`. Par exemple pour une taille de tuile de 960 et 4 GPUs, `str-Chameleon` avec 2 streams génère 75 Go de transferts et génère 85 Go de transferts avec 4 streams. De plus, avec ce même nombre de GPUs et pour une taille de tuile de

320, `str-Chameleon` avec 4 streams génère 110 Go de transferts. Cette forte consommation mémoire peut être un facteur de limitation de l'utilisation de multi-stream.

Comparaison avec `h-DPLASMA` et `MAGMA`

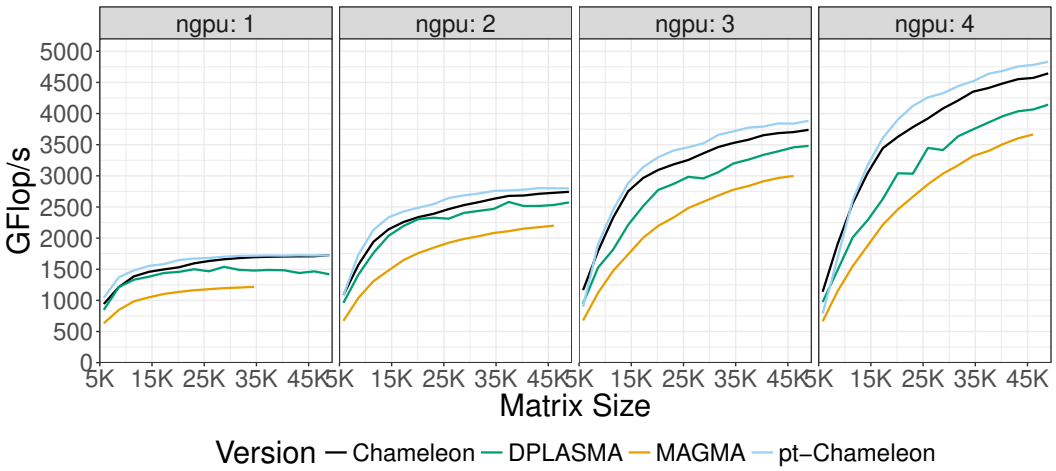


FIGURE 3.28 – Comparaison de `pt-Chameleon` contre `Chameleon`, `MAGMA` (paramètres par défaut et `Intel MKL` parallèle) et `h-DPLASMA`.

Enfin, la Figure 3.28 compare `pt-Chameleon` à plusieurs librairies d'algèbre linéaire dense de référence : `MAGMA`, `Chameleon` et `h-DPLASMA`. `h-DPLASMA` est `DPLASMA` avec le schéma de granularité hiérarchique présenté dans [118]. Cette version de `DPLASMA` permet d'obtenir de meilleurs performances via l'utilisation de deux granularités simultanément : (1) sur GPU une granularité (taille de tuile NB) grosse est utilisée alors que (2) sur CPU une granularité plus faible que NB (IB) est utilisée pour maximiser le parallélisme sur CPU. Ceci est réalisé grâce à une synchronisation en début de tâche et une seconde en fin de tâche lorsque la tâche s'exécute sur CPU, permettant d'utiliser un sous-découpage et une vision plus fine des données le temps de l'exécution de la tâche. La version `MAGMA` utilise les 24 cœurs du CPU et `h-DPLASMA` utilise 20 cœurs des CPUs, de façon similaire à StarPU un cœur est dédié à la gestion de chaque GPU. Pour créer cette figure, pour chaque point de `Chameleon` et `pt-Chameleon` uniquement la meilleure performance obtenue d'après les Figure 3.22 et Figure 3.26 est présentée. Pour `Chameleon`, les données de `str-Chameleon` sont aussi considérées dans cette recherche du maximum. Pour `MAGMA` la performance présentée est celle avec sa configuration par défaut et pour `h-DPLASMA` la performance montrée est la meilleure performance pour chaque point trouvée dans les données fournies par l'équipe de `DPLASMA`. `h-DPLASMA` utilise des

tailles de tuiles équivalentes à `pt-Chameleon` et `Chameleon` (*i.e.* 480, 960, 1440, 1920) avec des tailles de bloc sur CPU (IB) de (160, 192, 240 et 320).

De façon générale, `pt-Chameleon` obtient dans la plupart des cas la meilleure performance et se trouve notamment au dessus de `Chameleon`, `h-DPLASMA` et `MAGMA` pour de larges tailles de matrices si l'on regarde les résultats avec un nombre de GPUs compris entre deux et quatre. Cependant, grâce à la fusion des données de `str-Chameleon` et `Chameleon`, on remarque que `Chameleon` devient très compétitif avec les autres versions y compris `pt-Chameleon` pour des matrices de rang faible à intermédiaire (jusqu'à $15K$ pour 4 GPUs, *i.e.* avant d'atteindre le plateau de performance). Enfin, on remarque que `pt-Chameleon` obtient la meilleure performance absolue, avec un maximum de 4.8 TFlop/s sur cette machine équipée de 4 GPUs avec une matrice de rang $48K$.

Ces résultats mettent en avant la portabilité aussi bien en terme de logiciel qu'en terme de performance de l'utilisation de tâches parallèles puisque cette technique permet d'exploiter des systèmes hétérogènes modernes avec efficacité de façon transparente. Cette approche permet aussi de s'attaquer au problème de la granularité des tâches qui survient avec ce genre de machine hétérogène en réduisant le nombre de ressources et l'écart de performance sur CPU par rapport au GPU. Ceci permet à notre approche d'obtenir de meilleures performances en changeant la granularité des tâches pour de grosses tailles de matrices et permet aussi d'obtenir de meilleures performances pour de plus petites tailles de matrices grâce à un plus faible besoin en parallélisme pour remplir la machine.

3.4 Discussion

Sur la plupart des processeurs (avec l'exception notable des cœurs de l'Intel KNL), l'implantation séquentielle des noyaux BLAS obtiennent de meilleures performances que leur version avec plusieurs processus légers si l'on considère la métrique GFlop/s/cœur, d'efficacité des noyaux. Ceci explique pourquoi les bibliothèques de l'état de l'art en algèbre linéaire dense créent des tâches invoquant des noyaux BLAS sur des cœurs individuels. À l'asymptote sur des machines homogènes cette approche est la plus efficace, si l'on considère que la granularité des tâches est gardée suffisamment petite.

Cependant, sur des machines hétérogènes, la taille de bloc doit être significativement plus large pour exploiter efficacement les GPUs. Ceci créé plusieurs problèmes pour les CPUs. En premier lieu, l'augmentation de la durée des tâches sur CPU augmente en général aussi la durée du chemin critique. En second lieu, exécuter plusieurs tâches sur de larges données créé une mauvaise utilisation du cache. Enfin, à cause de la plus faible quantité de tâche prête à tout instant lorsque l'on augmente la granularité, les CPUs peuvent avoir des périodes de famines.

Afin de contourner ce problème, l'application peut essayer d'utiliser des tâches de grain fin et utiliser les fonctionnalités "multistream" sur GPU afin d'améliorer l'occupation globale des GPUs avec de petites tâches. En effet, les accélérateurs GPUs modernes permettent au programmeur de lancer plusieurs noyaux de façon concurrente avec ces technologies. Cependant, cette approche est très "boîte noire", dans le sens où aucun contrôle n'est donné sur quand et comment sont exécutés les noyaux sur GPU, rendant l'estimation de la durée des noyaux plus complexe. De plus, en augmentant le nombre de noyaux tournant sur GPU, le volume de calcul entre l'hôte et l'accélérateur augmente significativement.

Une autre façon de régler ce problème serait de dynamiquement découper des tâches à gros grain lorsqu'elles sont attribuées sur les cœurs des CPUs. Dans cette approche, les tâches doivent être remplacées par une sous-graphe de tâches de granularité plus fine, permettant une gestion plus fine des dépendances et un pipeline plus efficace des noyaux. En effet, l'utilisation de tâches parallèles quand-à elle créé des barrières à la fin des tâches. Cependant, cela demande une gestion plus coûteuse des données au niveau du support d'exécution puisque les blocs de données doivent être distribués et réunis dynamiquement. De plus, cela soulève la question difficile mais intéressante de l'ordonnancement : quel est le bon intervalle de temps pour exécuter une tâche qui génère un sous-graphe ? et, alors, quel sous-graphe choisir parmi plusieurs options ? Je pense que les graphes de tâches hiérarchiques et les tâches parallèles sont deux approches complémentaires et prometteuses pour s'attaquer à l'exécution distribuée de graphes de tâches (le mode distribué est décrit et analysé dans [9]). En effet, les tâches parallèles aussi bien que les graphes de tâches hiérarchiques permettent de grossir la granularité des communications inter-nœuds tout en préservant un bon équilibre de charge intra-nœud.

L'approche présentée ici ne souffre pas des problèmes décrits précédemment. Son problème principal est que lorsque l'on utilise un modèle d'interaction maître-esclave, les surcoûts du support d'exécution externe sont démultipliés sur tout le groupe de cœurs de cœur puisque seule une ressource est réveillée et utilisée par le support d'exécution externe. Ce phénomène peut cependant être contrebalancé par l'utilisation d'une granularité des tâches plus importante. Par rapport aux techniques présentées précédemment, l'utilisation de tâches parallèles possède plusieurs avantages. Les tâches sur le chemin critique peuvent être accélérées grâce à l'utilisation de larges groupes de cœurs de cœurs. L'utilisation d'une granularité de tâche plus importante est aussi pratique quand il n'y a pas assez de tâches prêtes pour remplir la machine. De plus, former des cœurs de CPU permet une utilisation plus efficace de la hiérarchie des caches. En effet, les implantations haute performance des noyaux BLAS sont très optimisés en terme de réutilisation et localité des données. Ainsi, exécuter ces noyaux sur plusieurs cœurs partageant directement un cache L2 ou L3 augmentera l'utilisation du cache et permet de baisser la pression sur

le bus de la machine. Du point de vue de l'ordonnanceur, cette approche a deux avantages majeurs. D'abord, regrouper les ressources dans des groupes de cœurs permet de réduire fortement la quantité de workers qui interagissent entre l'ordonnanceur et le support d'exécution. Ensuite, regrouper les cœurs de CPUs dans des groupes de cœurs, permet à la plateforme de devenir moins hétérogène en terme de puissance de calcul. Cela réduit significativement les pénalités introduites par de mauvaises décisions d'ordonnancement.

Dans la suite de cette thèse, plusieurs problèmes restent à traités. En premier lieu, il est nécessaire d'étudier si, et comment, les tâches parallèles permettent d'implanter des applications plus complexes. En second lieu, il est nécessaire de s'intéresse à l'ordonnancement dynamique de tâches parallèles et à la dimension des groupes de cœurs. En effet, dans ce chapitre les configurations utilisées sont homogènes et statiques et il est nécessaire d'étudier l'intérêt de configurations non homogènes, même si elles restent statiques, et d'étudier s'il est possible d'obtenir de meilleures performances dans ce cas. Ensuite, il est nécessaire de s'attaquer au problème de trouver la meilleure configuration de groupes de cœurs pour une partie de l'exécution donnée et changer ces configurations dynamiquement grâce à un ordonnanceur.

Chapitre 4

Application de l'agrégation de ressources

L'objectif de ce chapitre est de montrer des exemples d'utilisation de tâches parallèles dans des applications complexes. La première application étudiée est `FLUSEPA` en section 4.1. FLUSEPA est un code de mécanique des fluide développé chez Airbus DS et porté au dessus du support d'exécution StarPU par Jean-Marie Couteyen-Carpaye au cours de sa thèse [42]. La forme du graphe de tâches de cette application rend l'utilisation de tâches parallèles critique pour obtenir de bonnes performances. La seconde application étudiée est l'algorithme d'algèbre linéaire dense de la factorisation LU ($PA = LU$) en section 4.2 dont l'implantation efficace au dessus d'un support d'exécution n'est pas naturel sans l'utilisation de tâches parallèles à cause de la nature de l'algorithme.

Sommaire

4.1 FLUSEPA, un code de mécanique des fluides

Les tâches parallèles ont été utilisés en premier dans l'application FLUSEPA. Cela a été l'occasion de plusieurs interactions fructueuses permettant d'apporter de nouvelles fonctionnalités aux groupes de cœurs.

4.1.1 Contexte

FLUSEPA est un code de mécanique des fluides (CFD) développé depuis environ 30 ans chez Airbus DS. Ce code sert à simuler les écoulement autour des lanceurs. Il est utilisé pour la rentrée atmosphérique, le calcul des ondes de souffle au décollage, l'étude des phénomènes de turbulence pour les lanceurs,... Ce code permet aussi de modéliser la séparation des étages d'accélération à poudre d'Ariane 5, tel que montré dans la Figure 4.1.

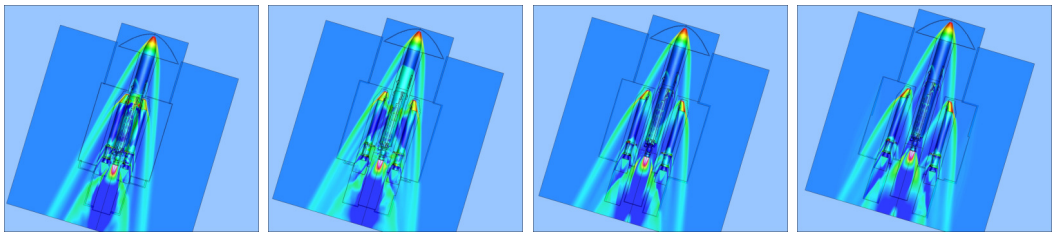


FIGURE 4.1 – Séparation des étages d'accélération à poudre d'Ariane 5

À chaque itération, le code FLUSEPA possède trois phases importantes : le solveur aérodynamique, le calcul de la cinématique et la réactualisation du maillage lorsque nécessaire. Lors de sa thèse de Jean-Marie Couteyen-Carpaye [42] a finalisé et optimisé une version OpenMP+MPI du code. Ensuite, il s'est concentré sur l'utilisation du modèle de programmation à base de tâches pour exécuter le solveur aérodynamique au dessus du support d'exécution StarPU. Lors de cette étude, la nécessité de réaliser des compromis sur la granularité des tâches et le parallélisme sont apparus, amenant un intérêt pour les tâches parallèles. On présente donc ici le travail et les résultats de Jean-Marie Couteyen-Carpaye sur le solveur aérodynamique mettant en œuvre les tâches parallèles en contextes mono-nœud et distribué.

4.1.2 Apparition de chaînes de tâches dans la version taskifiée

La source essentielle de parallélisme à l'intérieur du solveur est la décomposition spatiale en domaines. Cependant, la quantité de calcul à l'intérieur d'une cellule est trop faible pour créer des tâches avec cette granularité. Ainsi, ces cellules sont regroupées dans des *entités de calcul* (EC). Une distinction entre les cellules de bord et les cellules intérieures aux EC est aussi introduite ce qui permet une exécution plus efficace du graphe de tâches généré avec cette version.

De plus, l'algorithme utilisé pour la version taskifiée utilise un pas de temps local à travers une méthode d'intégration temporelle adaptative. Cette méthode est utilisée afin de découper une itération en sous-itérations régies par θ, tel qu'il y a 2^θ sous itérations. Puisque la méthode utilise des maillages dont la résolution n'est pas uniforme, cette technique d'intégration temporelle adaptative permet de ne pas utiliser le pas de temps le plus faible pour les grosses cellules qui serait pénalisant.

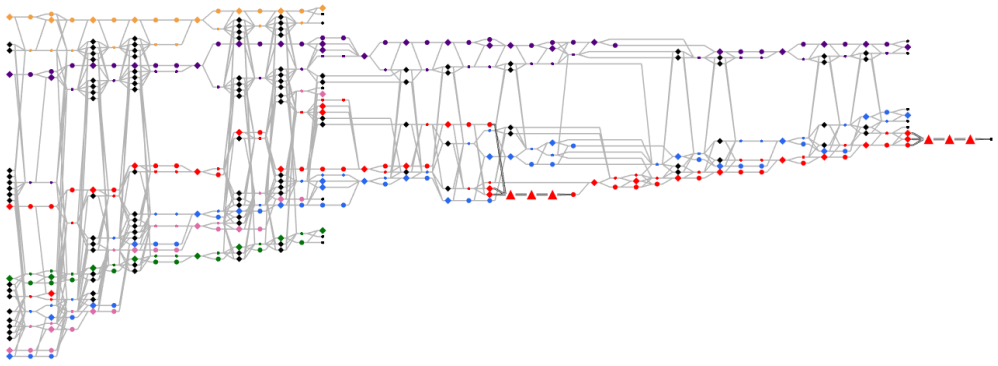


FIGURE 4.2 – Graphe de tâches généré par FLUSEPA pour un calcul avec 6 EC et $\theta = 2$

Un inconvénient des techniques utilisées pour la création du graphe de tâches, est que pour certains cas des chaînes de tâches sont générées avec relativement peu de parallélisme de tâche disponible. La Figure 4.2 illustre ce phénomène. Dans cet exemple, 6 ECs sont créées et il y a 3 niveaux temporels. On remarque dans ce graphe qu'à la fin l'EC rouge créé trois tâches en triangles. Ces tâches sont en fait des "packs" de tâches qui sont des chaînes de tâches compressées et sont donc très coûteuses. Le calcul se termine donc sur un enchaînement de tâches coûteuses qui ne peuvent être exécutées qu'en séquence par une seule unité de calcul par le support d'exécution.

4.1.3 Les tâches parallèles, une solution au problème

L'utilisation de tâches parallèles est une solution adaptée au problème pour plusieurs raisons. Premièrement, FLUSEPA possède une version OpenMP+MPI, ainsi des versions OpenMP des noyaux sont disponibles et peuvent être réutilisés dans la version taskifiée. Deuxièmement, exécuter ces grosses tâches sur plusieurs unités de calcul en parallèle permet d'obtenir une meilleure efficacité.

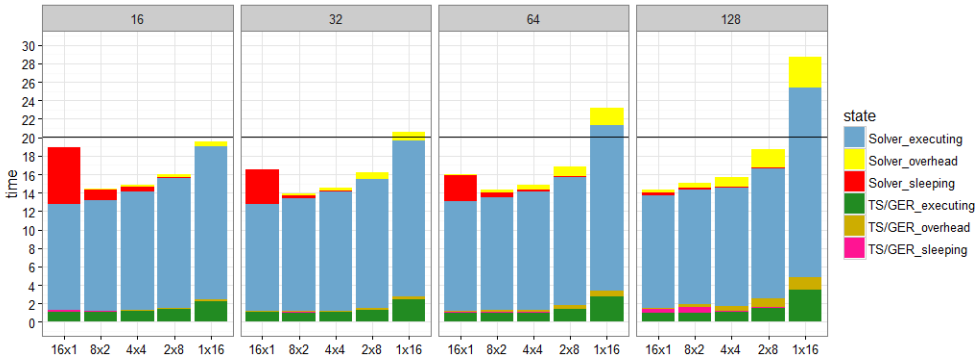


FIGURE 4.3 – Performance de FLUSEPA en temps avec différentes configurations de groupes de cœurs

La Figure 4.3 montre les résultats de performance de FLUSEPA au dessus de StarPU avec l'utilisation de tâches parallèles. Les différentes sous-figures montrent les performances avec 16, 32, 64 et 128 EC. En abscisse est représenté plusieurs façons de regrouper les ressources. La configuration 16×1 créé 16 groupes de cœurs de 1 unité de calcul chacun, alors que la configuration 1×16 créé un groupe de cœurs de 16 unités de calcul. On remarque que la configuration 1×16 est systématiquement moins efficace que les autres, cela s'explique par la scalabilité limité des noyaux en version OpenMP. Pour cette version, il n'y a plus aucun temps d'attente (en rouge). Cependant, utiliser des groupes de cœurs permet d'améliorer les performances dans la plupart des cas, notamment en utilisant les configurations 8×2 et 4×4. Cela se fait en réduisant le temps d'inactivité sur les unités de calcul grâce à l'exécution parallèle des tâches. Ce temps d'exécution est légèrement plus élevé lorsque la taille des groupes de cœurs augmente à cause de la scalabilité des noyaux. Un autre soucis est l'augmentation des surcoûts en fonction du nombre de ressources à l'intérieur du groupe de cœurs. Ce problème est simplement dû au fait que le même surcoût que précédemment (*e.g.* calcul des dépendances) est appliqué à plusieurs ressources et non plus une seule, mais cela reste minime pour de relativement petits groupes de cœurs. Au final, la version qui obtient la meilleure performance absolue est celle qui utilise 32 EC et la configuration 8×2. Dans une seconde expérience de [42] dans le cadre d'une version dis-

tribuée (28 nœuds de 20 cœurs), on observe que le gain de performance de la version StarPU en utilisant une configuration de groupes de cœurs de 4×5 est entre 25% et 40% par rapport à la version OpenMP + MPI.

4.1.4 Synthèse

D'après les résultats d'expérience, l'utilisation de tâches parallèles permet à FLUSEPA de jouer sur deux paramètres simultanés : la granularité des tâches à travers la taille des EC (et les niveaux temporels) et la granularité des ressources. À travers l'utilisation de ces deux paramètres, il est possible de trouver un meilleur compromis le parallélisme du graphe et le nombre de ressources permettant de maximiser l'occupation des ressources tout en gardant une efficacité d'utilisation des ressources acceptable. La Figure 4.3 montre que cette amélioration se fait malgré une perte d'efficacité des noyaux et une augmentation du surcoût du support d'exécution. On observe que l'utilisation de tâches parallèles dans certaines configurations (8×2 et 4×4) permet dans ce cas d'obtenir des performances plus stables lorsque l'on varie la granularité des tâches.

4.2 Factorisation LU avec pivotage partiel

Dans cette section, une implantation de l'algorithme $PA = LU$ est proposée. Il est montré que l'utilisation de tâches parallèles est nécessaire afin de simplifier le graphe de tâches et permettre d'obtenir de meilleurs performances. Plusieurs autres problèmes de l'implantation de l'algorithme sur un support d'exécution sont exposés et des solutions sont proposées. Enfin, une étude est réalisée sur plusieurs machines modernes et types de matrices.

4.2.1 Présentation de l'algorithme $A = LU$

L'algorithme $A = LU$ calcule la factorisation d'une matrice carrée A en deux matrices L et U, où L est une matrice triangulaire inférieure et U est une matrice triangulaire supérieure comme représenté dans la Figure 4.4. Soit L soit U possède des 1 sur sa diagonale.

La Figure 4.5 montre l'algorithme séquentiel de la factorisation LU. L'objectif de cet algorithme est d'introduire des 0 sous la diagonale afin de transformer une matrice A dans la matrice U. Au final, on observe que l'opération se compose d'un produit scalaire en ligne 4 et d'un produit dyadique ("outer product", opération "ger") en ligne 5. Le nombre d'opérations de cet algorithme pour une matrice de taille $m \times n$ est $mn^2 - 1/3n^3 - 1/2n^2 + 5/6n$. La parallélisation de cet algorithme est assez proche de la factorisation de Cholesky. On partitionne la matrice en deux dimensions afin de former des blocs et l'on applique des opérations d'algèbre linéaire de base sur ces tuiles comme montré

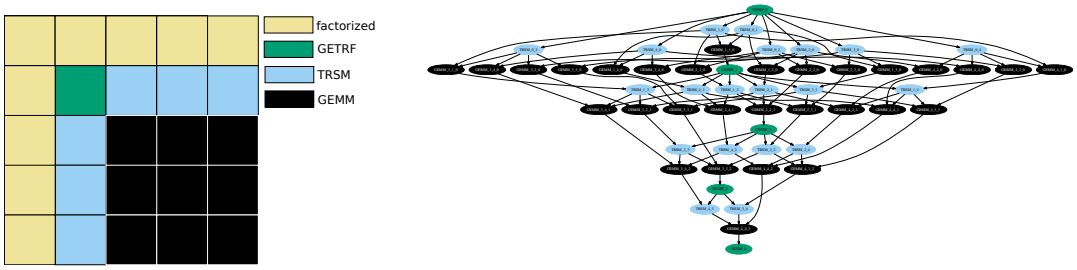
$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{2,1} & 1 & 0 & 0 \\ l_{3,1} & l_{3,2} & 1 & 0 \\ l_{4,1} & l_{4,2} & l_{4,3} & 1 \end{bmatrix} \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & u_{1,4} \\ 0 & u_{2,2} & u_{2,3} & u_{2,4} \\ 0 & 0 & u_{3,3} & u_{3,4} \\ 0 & 0 & 0 & u_{4,4} \end{bmatrix}$$

FIGURE 4.4 – Factorisation $A = LU$ avec une matrice carrée de taille 4×4 éléments

dans la Figure 4.6a. Ici, les opérations utilisées sont GETRF pour factoriser la tuile diagonale, TRSM pour résoudre les tuiles de la colonne et de la ligne de la diagonale factorisée, puis des GEMM pour mettre à jour les valeurs du reste de la matrice. On observe dans la Figure 4.6b que cette opération est très parallèle et génère beaucoup de tâches de type GEMM, plus encore que la factorisation de Cholesky.

```
1  U = A, L = I
2  for k : 1 ⇒ m − 1
3     for j : k + 1 ⇒ m
4        ℓ_{j,k} = u_{j,k} / u_{k,k}
5        u_{j,k:m} = u_{j,k:m} − ℓ_{j,k} u_{k,k:m}
6     end_for
7  end_for
```

FIGURE 4.5 – Algorithme séquentiel de la factorisation $A = LU$.



(a) Découpage de la matrice en blocs.

(b) Graphe de tâches.

FIGURE 4.6 – Schémas de parallélisme de la factorisation $A = LU$ avec une matrice de taille 5×5 tuiles

Une opération critique de la factorisation $A = LU$ est la division par l'élément diagonal $u_{k,k}$ à la ligne 4 de l'algorithme 4.5. Ainsi, si sur la diagonale

un des éléments est un 0 la factorisation échoue. Un second problème apparaît si l'élément diagonal est un chiffre de très faible valeur (*e.g.* de l'ordre de 10^{-16}) alors la factorisation devient instable numériquement (*cf.* [1]) puisque l'arithmétique des chiffres flottants a une précision $\epsilon_{machine} \approx 10^{-16}$.

4.2.2 Présentation de l'algorithme $PA = LU$

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix} = \begin{bmatrix} a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \end{bmatrix}$$

FIGURE 4.7 – Exemple de permutation $PA = A'$ d'une matrice carrée de taille 4×4 éléments

Afin de contrôler l'instabilité numérique de cet algorithme, il est nécessaire d'ajouter une étape de permutation. En plus de la factorisation présentée précédemment, cet algorithme génère une matrice P de permutation de A de sorte à ce que $PA = A'$. Les lignes de A sont permutées afin d'obtenir une matrice A' qui possède sur sa diagonale le maximum de la colonne. La Figure 4.7 montre un exemple de permutation d'une matrice A. Les lignes 2 et 1 de la matrice sont permutées ainsi que les lignes 3 et 4. En supposant que ces permutations placent l'élément maximum de chaque colonne sur chaque diagonale, la factorisation LU appliquée à une telle matrice devient plus stable. On note ainsi cette factorisation avec permutation $PA = LU$.

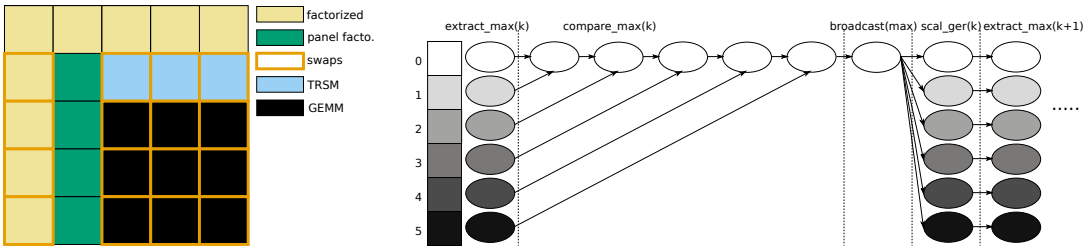
Plusieurs techniques de pivotage sont présentées dans l'article [46]. La première stratégie est de rechercher le meilleur élément (*i.e.* le plus grand) parmi la sous-matrice $A_{k:m,k:m}$ d'une matrice A de taille $m \times m$ pour remplacer l'élément $a_{k,k}$. Cette opération est très stable numériquement. Il a été montré que la stabilité dépend du facteur d'agrandissement des valeurs $\rho = max_{i,j}|u_{i,j}|/max_{i,j}|a_{i,j}|$ et pour le pivotage complet celui-ci est en moyenne de $n^{1/2}$. Cependant, cette opération est très cher et peu utilisée en pratique car il est nécessaire de réaliser de l'ordre de $\mathcal{O}(m^3)$ comparaisons avec cet algorithme. Ainsi, la plupart des versions de la factorisation LU recherchent le maximum uniquement dans une dimension de la matrice, *e.g.* dans la colonne.

La factorisation LU avec pivotage incrémental [32, 73] permet de changer la façon dont la colonne de la tuile diagonale est factorisée et d'accélérer la recherche de maximum. La technique utilisée est de factoriser la tuile diagonale puis de la combiner successivement avec chacune des tuiles de la colonne et de les refactoriser. Ceci permet de réaliser cette opération en gardant un bon

parallélisme global, notamment sur la progression des mises à jours (GEMM) au fur et à mesure de la factorisation de la colonne de la tuile diagonale (ci-après nommé panneau). Le problème principal de cette version est que le facteur d'agrandissement des valeurs est relativement élevé (en moyenne n ou $> n$), ce qui impacte sur la stabilité de cette version comme montré dans [46].

Plusieurs autres versions de factorisation LU ont été proposées, notamment une version intéressante est l'utilisation de pivotage avec un système de tournois développé d'abord en version distribuée [43, 58] puis en adapté en mémoire partagée [45]. Dans cette section, on se concentre sur la factorisation LU la technique dite de pivotage partiel présenté entre autre dans [1]. Le principe de cet algorithme est de procéder à des recherches de maximum dans toute la colonne $a_{k:m,k}$ pour chaque élément diagonal $a_{k,k}$ avant de factoriser la colonne en question, puis la même opération est réalisée pour la colonne $k+1$. Cet algorithme propose un bon compromis entre le pivotage complet et les autres versions en terme de stabilité numérique, en effet son facteur d'agrandissement moyen des valeurs est de $n^{2/3}$ et le nombre de comparaisons à réaliser pour cette version est de $\frac{1}{2}n^2 - \frac{1}{2}n$ ce qui est beaucoup plus faible que pour le pivotage complet.

4.2.3 Difficultés d'implantation de $PA = LU$ à base de tâches



(a) Découpage d'une matrice en tâches de haut niveau pour réaliser une factorisation $PA = LU$.

(b) Une étape de la factorisation du panneau en tâche pure avec un panneau de 6 éléments en dimension m sans optimisations.

FIGURE 4.8 – Schémas algorithmiques de la factorisation $PA = LU$

On présente ici à la réalisation de l'algorithme $PA = LU$ avec un pivotage partiel à base de tâches. Afin d'obtenir un maximum de parallélisme, il est judicieux d'utiliser un blocage à deux dimensions sur la matrice A. À haut niveau, l'algorithme peut alors être représenté comme sur la Figure 4.8a. Dans cette figure, la première colonne et la première ligne sont déjà factorisées. En vert est représenté un ensemble de tâches `facto panel` qui travaillent sur

toute la colonne et factorisent la tuile diagonale actuelle ainsi que les tuiles de la même colonne. Cette étape sera expliquée plus en détail par la suite. Une fois la factorisation réalisée, les lignes de la matrice ont possiblement pivotées à cause de la nature de l'algorithme. Il est donc nécessaire d'appliquer des tâches de SWAPs sur les parties de la matrice concernées. Ces tâches s'appliquent aux tuiles entourées de bordure verte. Une fois cette opération réalisée, il devient possible d'appliquer les tâches de TRSM, puis de GEMM.

La Figure 4.8b montre une implantation naïve de la factorisation du panneau (ensemble de tâches de la facto panel du schéma 4.8a) à base de tâches opérant sur des tuiles ou sous-parties du panneau à factoriser. L'algorithme doit traiter en séquentiel les colonnes de la matrice comprises sous la tuile diagonale et se positionne ici à la colonne k. La première étape à réaliser est que chaque tuile recherche localement le maximum de la colonne considérée (opération extract_max). Ensuite, il est nécessaire de comparer chacun des maximum locaux afin d'identifier le maximum réel de la colonne (opération compare_max). Une fois ce maximum trouvé, l'opération de broadcast permet d'échanger la ligne diagonale avec la ligne contenant le maximum et indique aux autres tuiles la terminaison de cette opération à l'intérieur du panneau. Elle s'assure aussi de sauvegarder l'indice de la ligne échangée afin d'appliquer la même opérations aux autres tuiles de la matrice. Enfin, l'opération scal_ger peut être réalisée en parallèle. Cette opération sert à factoriser la colonne k et mettre à jour les coefficients du reste du panneau. Une fois ces opérations réalisées, l'algorithme peut factoriser de façon similaire la colonne $k+1$ du panneau.

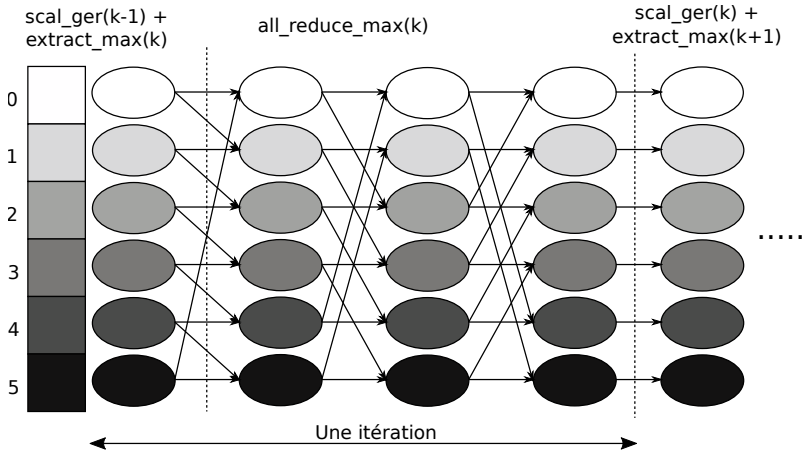


FIGURE 4.9 – Une étape de la factorisation du panneau en tâche pure avec un panneau de 4 éléments en dimension m avec optimisations.

Plusieurs optimisations de cet algorithme de factorisation du panneau à base de tâches pures sont montrées en Figure 4.9. En combinant la factorisation

de la colonne $k-1$ avec la recherche du maximum pour la colonne k du panneau il est possible d'économiser en nombre de tâches et d'augmenter la granularité de celles-ci. De même, il est possible d'utiliser un schéma "all reduce" afin de réaliser la comparaison et l'opération de broadcast plus efficacement.

Le problème majeur de l'utilisation d'un modèle de tâches pures pour réaliser l'algorithme de factorisation $PA = LU$ est qu'il génère un nombre conséquent de tâches de granularité faible pour chaque colonne du panneau à factoriser. Cette opération de factorisation se trouve sur le chemin critique et il est avantageux de l'optimiser. Une autre version proposée dans [48] permet de résoudre ce problème. La proposition est de réaliser toutes ces opérations de recherche de maximum, broadcast et factorisation du panneau dans une seule et même tâche appliquée à tout le panneau. Cette méthode présente l'avantage de limiter fortement le nombre de tâches générées pour la factorisation du panneau. Cette factorisation du panneau en une seule tâche rend l'utilisation de tâches parallèles critique. En effet, puisque ce calcul est situé sur le chemin critique de l'algorithme il est judicieux d'assigner plusieurs processus légers au calcul de cette tâche comme le propose [48].

Difficultés d'adaptation de l'algorithme aux caractéristiques de la matrice

Une seconde difficulté de cet algorithme est que le graphe de tâches n'est pas le même selon la matrice considérée. Prenons l'exemple d'une matrice avec une diagonale strictement dominante (DSD). Dans une telle matrice la valeur de l'élément de la diagonale est supérieur ou égal à la somme de tous les éléments de la colonne. Autrement dit, soit une matrice $A = ((a_{i,j})_{i,j\in[1,n]})$, alors $\forall i \in [\![1, n]\!], |a_{i,i}| > \sum_{\substack{j=1 \\ j\neq i}}^{n} |a_{i,j}|$. Dans une telle matrice, aucun échange de ligne n'est nécessaire, car la recherche de maximum ne trouvera aucun élément supérieur à celui se trouvant sur la diagonale. À l'inverse, si la matrice considérée est une matrice générée aléatoirement, il est probable qu'un échange de ligne soit nécessaire pour chaque élément de la diagonale afin d'améliorer la stabilité de la factorisation.

Pour une efficacité optimale, on se doit donc d'adapter le graphe de tâches et les tâches générées afin de ne générer des tâches de permutation de lignes uniquement lorsque nécessaire. Ceci est un challenge car la plupart des supports d'exécution ne permettent pas de mécanisme de génération de tâches avec une condition sur les données en entrée. Avec les deux types de matrice présentés précédemment, pour une matrice DSD aucune tâche d'échange de données ne doit être généré alors que dans le cas d'une matrice aléatoire on est susceptible de réaliser des échanges de lignes pour toutes les lignes de la matrice.

Difficultés du code avec accélérateur

Plusieurs difficultés apparaissent à l'implantation efficace de cet algorithme lorsque l'on considère une machine dotée d'accélérateurs. Comme présenté, l'algorithme réalisé à base de tâches éventuellement parallèles correspond à celui de la Figure 4.8a. Cet algorithme expose plusieurs grains de parallélisme. Il y a la tâche de calcul du panneau qui se doit d'être exécuté en parallèle sur plusieurs cœurs simultanément et possède un grain de calcul supérieur aux autres tâches. De plus, il est aussi avantageux de concilier ces différences de granularité des tâches avec l'utilisation d'accélérateurs, ce qui créé en plus de cela une différence de granularité des ressources. Le contrôle de ces deux différences de granularités semble donc nécessaire pour une implantation efficace de cet algorithme.

De plus, les tâches d'échanges de lignes ne génèrent pas de transfert de donnée dans le cas d'une machine homogène, cependant dans le cas d'une machine hétérogène il sera nécessaire de limiter les échanges afin de ne pas surexploiter les capacités de la machine.

4.2.4 L'algorithme $PA = LU$ sur un support d'exécution

Dans la suite de cette section, on propose d'implanter l'algorithme $PA = LU$ au dessus de StarPU à l'aide de tâches parallèles en utilisant le noyau de factorisation de panneau proposé par [48]. Plusieurs autres améliorations à l'algorithme $PA = LU$ sont explorées afin de résoudre les problèmes exposés précédemment et rendre cette opération efficace sur des machines manycore et hétérogènes au dessus de support d'exécution.

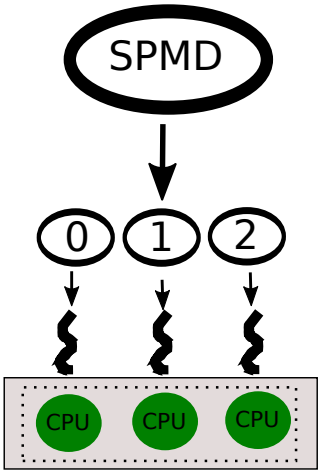
Utilisation des tâches parallèles

```
1  for k:0->NT
     submit(facto_panel, A(k):RW, ipiv(k):W) //colonne k
3    for n:k+1->NT
       submit(SWAPs, A(n):RW, ipiv(k):R)
5
       submit(DTRSM, A(k,k):R, A(k,n):RW)
7
       for m:k+1->MT
9        submit(DGEMM, A(m,k):R, A(k,n):R, A(m,n):RW)
       end_for
11   end_for

13   for n:0->k
       submit(SWAPs, A(n):RW, ipiv(k):R)
15   end_for
   end_for
```

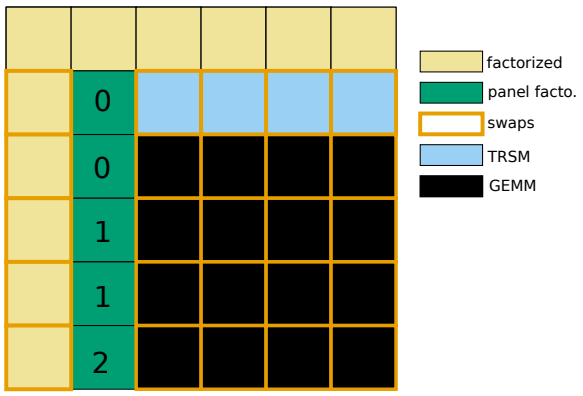
FIGURE 4.10 – Schéma algorithmique de la factorisation $PA = LU$ avec une tâche parallèle.

Le pseudo-code 4.10 montre le schéma algorithmique de la factorisation

$PA = LU$ avec une tâche parallèle pour factoriser le panneau. Contrairement à ce qui est observé dans la Figure 4.9 où plusieurs tâches sont générées pour chaque colonne du panneau, une seule tâche est créée pour la factorisation du panneau et le calcul des pivots. La tâche parallèle utilisée prend une sous-matrice avec un accès en lecture et écriture (RW). Autrement dit du point de vue du support d'exécution, il est nécessaire d'exprimer des dépendances prenant des tuiles évoluant selon l'itération considérée. Un tel mode est disponible dans StarPU où il est possible de passer un nombre variable de données en entrée au travers du mode `STARPU_VARIABLE_NBUFFERS`.



(a) Duplication de tâches et attribution de rang pour la tâche SPMD du panneau.



(b) Utilisation des rangs pour assigner le traitement de tuiles aux pthreads.

FIGURE 4.11 – Schéma de parallélisation du panneau à l'aide de tâche parallèle de type SPMD.

Afin d'exécuter le panneau comme une tâche parallèle, le mode de tâches SPMD est utilisé. La Figure 4.11a montre la duplication de la tâche et l'attribution de rangs associés à la tâche SPMD représentant la factorisation du panneau, comme présenté précédemment en Section 3.2.2. Cette étape est réalisée de façon transparente par le support d'exécution avant l'exécution des tâches dupliquées de cette tâche parallèle. Lors de l'appel de la tâche, un rang est assigné au worker permettant au programmeur de noyau de différencier les appels. La Figure 4.11b montre la distribution du parallélisme à l'intérieur du panneau. La méthode choisie est de répartir les tuiles du panneau entre les différents processus légers de façon équitable. Chaque processus léger se charge de rechercher les maximums locaux dans ses tuiles avant de les partager grâce à une réduction, puis la factorisation peut avoir lieu. Toutes les synchronisations sont exprimées à l'intérieur d'une seule tâche ce qui est plus efficace que l'utilisation du modèle de tâches pour représenter de nombreuses opérations

successives sur chaque colonne du panneau comme dans la Figure 4.9.

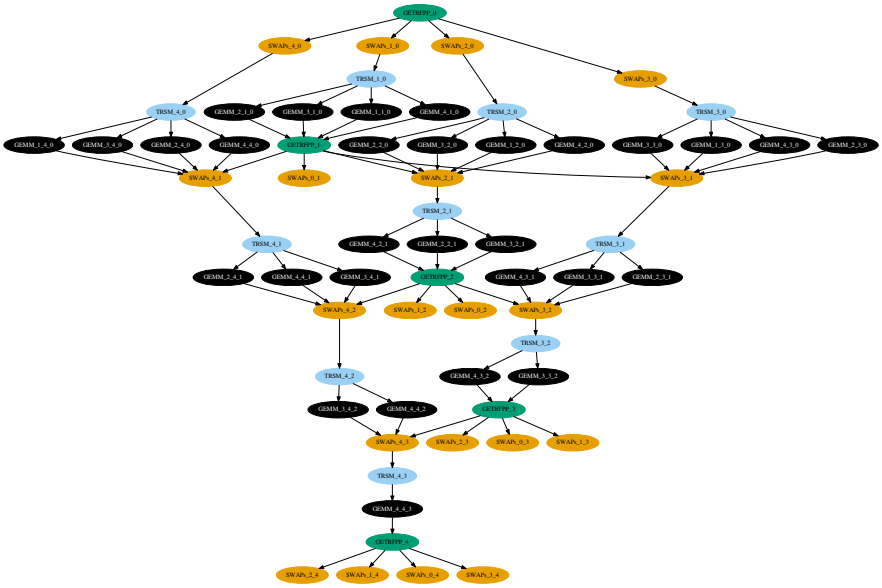
L'algorithme présenté soumet des tâches de swaps pour toutes les colonnes de la matrice de taille équivalente au panneau, même si cela n'est pas nécessaire notamment dans le cas d'une matrice DSD. La prochaine version permet d'outrepasser cette limite.

Contrôle de flux de soumission

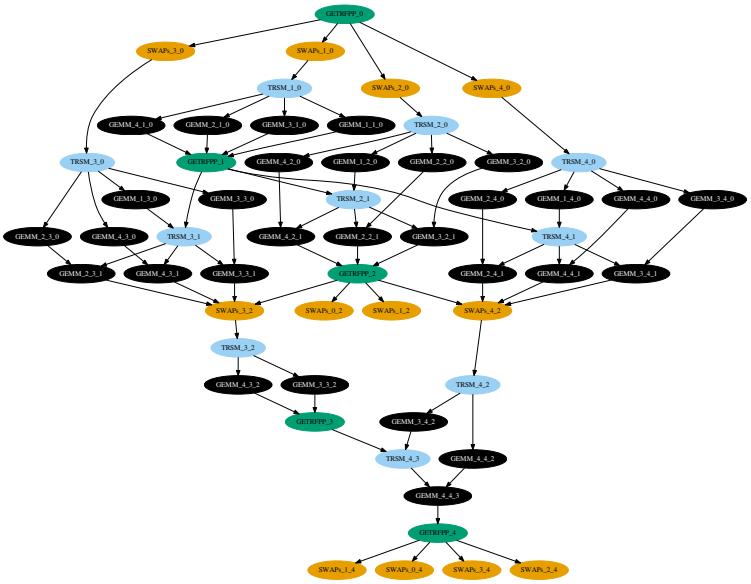
```
submit(facto_panel, A(0):RW, ipiv(0):W)
for k:0->NT
  acquire(ipiv(k):R) // arret du flux de soumission
  analyse_pivots()
  release(ipiv(k))
  for n:k+1->NT
    if needed
      submit(SWAPs, A(n):RW, ipiv(k):R)
    end_if

    submit(DTRSM, A(k,k):R, A(k,n):RW)

    for m:k+1->MT
      submit(DGEMM, A(m,k):R, A(k,n):R, A(m,n):RW)
    end_for

    if n is k+1
      submit(facto_panel, A(k+1):RW, ipiv(k+1):W)
    end_if
  end_for

  for n:0->k
    submit(SWAPs, A(n):RW, ipiv(k):R)
  end_for
end_for
```

FIGURE 4.12 – Schéma algorithmique de la factorisation $PA = LU$ avec contrôle du flux de soumission, anticipation du chemin critique et tâche parallèle pour la factorisation du panneau.

La Figure 4.12 montre deux améliorations par rapport à l'algorithme précédent. La première consiste à anticiper le chemin critique dans la soumission grâce à la soumission de la tâche de factorisation du panneau au plus tôt. Le deuxième point corrige un problème de la version proposée dans la Figure 4.10 où la création de tâches d'échanges de lignes pour toute la matrice, même lorsque cela n'est pas nécessaire. Dans l'algorithme de la Figure 4.12 ce problème est corrigé grâce à l'utilisation de contrôle de flux de soumission au biais d'un outil du support d'exécution que l'on nomme `acquire-release`. Cette fonction suspend le processus léger de soumission de tâches jusqu'à ce que la donnée demandée, ici `ipiv(k)` produite par la factorisation du panneau, soit disponible en lecture. Lorsque le processus léger de soumission est débloqué à nouveau, la tâche de factorisation du panneau est nécessairement terminée et la donnée produite, le vecteur de pivots, est disponible en lecture pour le processus léger de soumission. Le processus léger de soumission analyse alors ce vecteur avant de relâcher la donnée. Grâce à cela, il est possible de soumettre uniquement les tâches d'échange de lignes requises.



(a) Si les tâches d'échange de ligne sont toujours soumises.



(b) Si les tâches d'échange de ligne ne sont pas systématiquement soumises.

FIGURE 4.13 – Exemples de graphes de tâches de la factorisation $PA = LU$ pour une matrice carrée de 5 tuiles de côté qui génère des échanges de lignes que pour une itération sur deux.

La Figure 4.13 montre deux exemples de graphes de tâches de la factorisation $PA = LU$ pour une même matrice de taille 5×5 tuiles. Dans cet exemple, il est considéré que seulement une instance de la tâche `GETRFPP` de factorisation du panneau nécessite d'échanger les lignes de la matrice. Le code couleur utilisé est le même que dans la Figure 4.11, c'est à dire : `GETRFPP`, `SWAPs`, `TRSM`, `GEMM`. On remarque sur la Figure 4.13a que la présence de tâches `SWAPs` créé beaucoup de points de synchronisations, sous formes de barrières oranges. Ceci est d'autant plus problématique que ces tâches génèrent des surcoûts importants, surtout en mouvement de données.. Au contraire dans la Figure 4.13b, la soumission d'échanges de lignes que lorsqu'ils sont nécessaires pour ce même cas permet de s'affranchir des pseudo synchronisations forcées par les échanges de lignes et obtenir des zones de parallélisme dense autour du deuxième appel à la tâche `GETRFPP` notamment. Cependant, puisque cette technique repose sur la suspension du processus léger de soumission il faut s'assurer que cela n'impacte pas négativement les performances de l'algorithme. En effet, il est possible de recouvrir la suspension du processus léger de soumission, dû à l'attente du tableau des pivots produit par la tâche parallèle de factorisation, avec du calcul. Ceci est possible notamment grâce à la soumission anticipée de la tâche de factorisation et l'utilisation de priorité suffisamment grande pour favoriser l'exécution de cette tâche.

Une autre raison de ne pas soumettre toutes les tâches d'échange de lignes est la très faible efficacité et les surcoûts des tâches d'échange de lignes sur un support d'exécution à base de tâches. En effet, si ces échanges de lignes ne sont pas optimisés lorsque l'on se situe sur une machine hétérogène le volume de communication généré sera de l'ordre de toute la matrice, ce qui remet en question les décisions précédente d'ordonnancement ainsi que la localité des données sur la machine et cela peut limiter les performances de cet algorithme selon l'efficacité des communications.

Complexité des swaps et types de swaps implantés

Numéros de ligne	0	1	2	3	4
Numéros du pivot	17	6	6	3	4

FIGURE 4.14 – Pivots choisis pour la factorisation du panneau de la Figure 4.15

Plusieurs schémas d'échanges de lignes sont disponibles avec chacun leurs avantages, inconvénients et limites. On peut identifier trois types principaux de tâches d'échange de lignes selon la granularité des échanges de donnés à réaliser : la colonne entière, des paires de tuiles, des groupes de lignes à échanger. La Figure 4.15 explicite l'utilisation de ces trois techniques pour implanter la

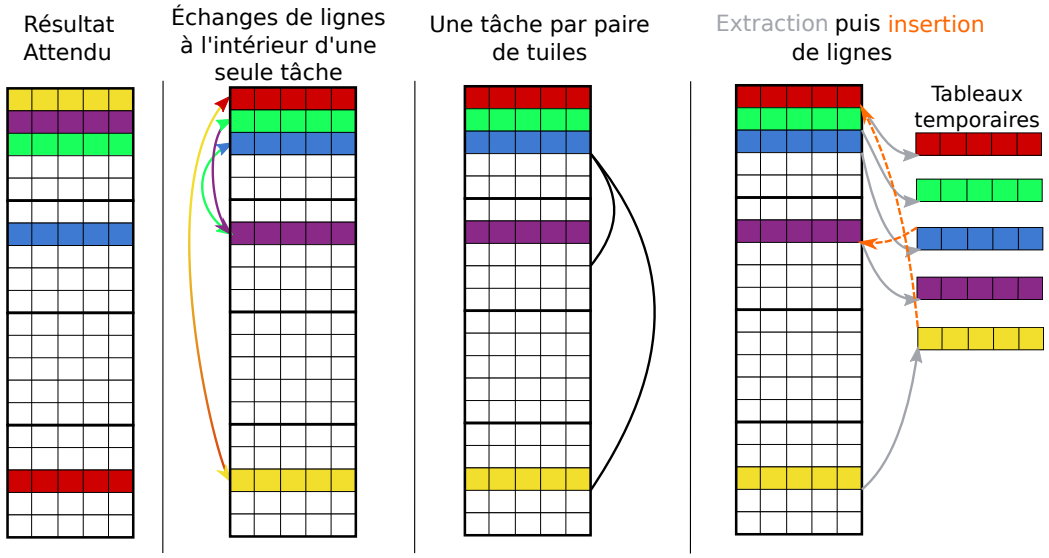


FIGURE 4.15 – Différentes façons d'implanter l'échange de lignes pour une même matrice

tâche d'échange de ligne SWAP à travers un exemple. Le Tableau 4.14 montre les pivots choisis lors de l'étape de factorisation du panneau précédant dans cet exemple. Dans cet exemple, trois lignes doivent être échangés dans une séquence précise avec deux autres lignes de la matrice. Une façon de représenter ces échanges est d'utiliser un vecteur de pivots, qui est représenté dans le Tableau 4.14. Pour chaque étape d'échange de ligne suivant la factorisation, seulement 5 lignes peuvent être échangées avec d'autres car seul la factorisation du panneau ne s'applique que sur une tuile. Afin de faciliter la compréhension, le résultat attendu d'après le vecteur de pivots est aussi montré à gauche dans la Figure 4.15.

Une tâche d'échange de lignes sur toute la colonne On s'intéresse ici à la deuxième partie de la Figure 4.15 et le Tableau 4.14. Cette première version ne créé qu'une seule tâche qui opère en écriture sur toute la colonne ce qui réalise donc un échange de ligne à *gros grain*. L'avantage de cette technique est que l'écriture du noyau est simple. Ce noyau consiste à une simple itération sur les lignes de la tuile qui a été précédemment factorisée (la tuile 1 ici) et prenant en paramètre le vecteur de pivots ainsi que toutes les tuiles de la colonne. Ici les échanges à réaliser sont les suivants : la ligne 1 est échangée avec la ligne 5, la ligne 2 est échangée avec la ligne 4 puis la ligne 3 est à nouveau échangée avec la **ligne 4**, qui est désormais la ligne 2 à cause de l'échange précédent. Ceci est explicité avec le résultat attendu dans la partie la plus à gauche de la Figure 4.15. L'utilisation d'une seule tâche pour tous les échanges de ligne de la colonne possède plusieurs inconvénients bien que

le nombre de tâches générées est faible (une tâche par colonne par itération). Le premier inconvénient est que cela créé plusieurs points de synchronisation comme observé dans le graphe 4.13a. Le second inconvénient, est la quantité de mémoire déplacée lorsque l'on se trouve sur une machine hétérogène. En effet, dans une telle situation les tuiles peuvent être présentes dans n'importe quelle mémoire de la machine et l'accès à toutes ces tuiles en écriture nécessite le rapatriement complet des tuiles de la colonne au même endroit. Au final, dans le cas où toutes les données sont présentes sur un autre nœud mémoire, à l'itération 0 pour une matrice de taille $n \times n$ avec des blocs de taille $n_b \times n_b$, il est nécessaire de rapatrier $n^2 - n_b * n$ éléments à cause des échanges de lignes, soit toute la matrice sauf la colonne factorisée. Ceci peut aussi avoir un impact négatif sur l'ordonnancement puisque cela remet sans cesse en cause la répartition des donnés actuelles.

Une tâche d'échange de lignes entre paires de tuiles À titre d'information, une deuxième version de l'échange de lignes est représentée dans la troisième partie de la Figure 4.15. Le travail a réaliser est le même que précédemment, mais ici pour réaliser cette opération une tâche est émise par paire de tuile. La paire de tuile en question comprend toujours la tuile située le plus haut dans la matrice, qui est en train d'être factorisée, et une tuile inférieure qui peut fournir de bons pivots à celle-ci. Cette approche permet de s'adapter à la forme de la matrice et de transférer moins de données dans certains cas, comme on peut l'observer dans la Figure 4.15. Cependant, plusieurs points négatifs peuvent être remarqués. Le nombre de tâches dans le pire des cas, *i.e.* avec une matrice aléatoire et des échanges de ligne entre toutes les tuiles, est à l'itération k de $2(n_t - k) * (n_t - 1)$ où $n_t = (n/n_b)$, le nombre de tuile dans une dimension de la matrice. De plus, bien que dans les cas où l'on a un nombre intermédiaire d'échange de lignes (ou bien réparti entre quelques tuiles) le volume de données transférées sur une machine hétérogène est moindre que la version par colonne, dans le cas extrême d'une matrice aléatoire le volume de données transférées est le même que la version colonne. Enfin, ce type d'échange de lignes centralise toujours les données de la matrice sur un seul nœud dans le cas d'une machine hétérogène. En effet, la tuile en cours de factorisation ("supérieure") à l'étape considérée doit recevoir les écritures de potentiellement toutes les autres. Cet algorithme peut cependant être parallélisé au travers d'écritures concurrentes dans la tuile "supérieure", mode qui n'est pas supporté dans la majorité des supports d'exécution. Cette version ne sera donc pas utilisée pour les expériences.

Échange de lignes à travers des tableaux temporaires Afin d'améliorer le volume de données transférées sur une machine hétérogène, une technique d'échange de ligne faisant usage de tableaux temporaires est proposée. Cette version contrairement aux autres réalise donc un échange de ligne à *grain fin*,

contrairement à la version par colonne qui elle est à *gros grain*. Le principe est de créer un tableau temporaire par groupe de lignes ayant la même tuile d'origine (source) et la même tuile destination. Cette technique peut donc être assimilée à l'échange de ligne à base de paire de tuiles. Deux tâches sont proposées, les tâches `d'extraction` de lignes qui permettent d'allouer (grâce au support d'exécution) et remplir les tableaux temporaires, et les tâches `d'insertion` qui insèrent ces lignes dans la tuile correspondante. La dernière partie de la Figure 4.15 exemplifie les tâches et échanges générés pour l'exemple considéré. Cependant, afin de pouvoir appliquer cette technique il est nécessaire d'identifier et de "démêler" les cycles du tableau des pivots, comme montré dans le Tableau 4.16. Afin de garder une interface commune avec les noyaux précédents et la norme LAPACK cette étape est ici réalisée en dehors du noyau de factorisation `GETRFPP`, mais il serait possible de définir un noyau de factorisation, avec une nouvelle interface, qui transforme le vecteur de pivots en vecteurs de permutations.

Numéros de ligne	0	1	2	3	4
Numéros du pivot	17	6	6	3	4
inf_vers_sup	17	6	1	3	4
sup_vers_inf	17	2	6	3	4

FIGURE 4.16 – Identification et suppression des cycles dans les choix de pivots pour la factorisation du panneau de la Figure 4.15

Deux sens sont identifiés pour l'échange de ligne, l'échange de ligne de la tuile supérieure vers les tuiles inférieures ("sup_vers_inf") et le sens inverse, "inf_vers_sup". Dans le cas des échanges "sup_vers_inf", on utilise la ligne de pivots associés et on extrait les lignes i que l'on insère dans $sup\_vers\_inf(i)$, si $sup\_vers\_inf(i) \in t_{inf}$ avec t_{inf} la tuile "inférieure" en question. La ligne de pivots "inf_vers_sup" permet de réaliser la contribution des tuiles inférieures dans la tuile supérieure. On extrait ici la ligne $inf\_vers\_sup(i)$ que l'on insère dans i.

Au final, le schéma algorithmique de la factorisation $PA = LU$ avec échange de lignes à travers l'utilisation de tableaux temporaire comme schéma de permutation en plus des autres optimisations proposées est montré dans la Figure 4.17. On procède d'abord par l'extraction des échanges de la tuile "supérieure" vers elle-même. Ensuite, les extractions des deux sens sont soumises puis les insertions. Enfin, on soumet l'insertion de la tuile supérieure vers elle même.

Cette version d'échange de lignes permet de fortement limiter le volume de données transférées. En effet, dans le cas d'une matrice allouée aléatoirement,

```
submit(facto_panel, A(0):RW, ipiv(0):W)
for k:0->NT
  acquire(ipiv(k):R)
  identify_cycles(ipiv(k), source(k), dest(k))
  release(ipiv(k))
  for n:k+1->NT
    if needed
      // the runtime may allocate all temporary buffers
      submit(extract_rows, A(k,n):R, inf_to_sup(k):R, upper_rows:W)
      for m:k+1->MT
        submit(extract_rows, A(k,n):R, sup_to_inf(k):R, up_to_bot_rows:W)
        submit(extract_rows, A(m,n):R, inf_to_sup(k):R, bot_to_up_rows:W)

        submit(insert_rows, sup_to_inf_rows:R, sup_to_inf(k):R, A(m,n):W)
        submit(insert_rows, inf_to_sup_rows:R, inf_to_sup(k):R, A(k,n):W)
      end_for
      submit(insert_rows, A(k,n):W, inf_to_sup(k):R, upper_rows:R)
    end_if

    submit(DTRSM, A(k,k):R, A(k,n):RW)

    for m:k+1->MT
      submit(DGEMM, A(m,k):R, A(k,n):R, A(m,n):RW)
    end_for

    if n is k+1
      submit(facto_panel, A(k+1):RW, ipiv(k+1):W)
    end_if
  end_for

  for n:0->k
    if needed
      submit(extract_rows, A(k,n):R, inf_to_sup(k):R, upper_rows:W)
      for m:k+1->MT
        submit(extract_rows, A(k,n):R, sup_to_inf(k):R, up_to_bot_rows:W)
        submit(extract_rows, A(m,n):R, inf_to_sup(k):R, bot_to_up_rows:W)

        submit(insert_rows, sup_to_inf_rows:R, sup_to_inf(k):R, A(m,n):W)
        submit(insert_rows, inf_to_sup_rows:R, inf_to_sup(k):R, A(k,n):W)
      end_for
      submit(insert_rows, A(k,n):W, inf_to_sup(k):R, upper_rows:R)
    end_if
  end_for
end_for
```

FIGURE 4.17 – Schéma algorithmique de la factorisation $PA = LU$ avec échange de ligne à travers des tableaux temporaires comme schéma de permutation, contrôle du flux de soumission et tâche parallèle.

i.e. où chaque ligne nécessite une permutation, le volume de donnée transférée à l'itération 0 est de $2n_b * n_b * (n_t - 1)$ éléments. Cela correspond à deux tuiles sur toute une ligne de la matrice (moins une colonne), comparé à la version colonne qui transfert toute la matrice (moins une colonne) à cette même itération. Cette version permet aussi d'extraire les lignes à transférer en parallèle, et permet d'insérer en parallèle sur les tuiles inférieures. Il serait aussi possible avec l'utilisation d'une écriture concurrente d'insérer en parallèle les lignes dans la tuile supérieure, puisqu'on est certain de ne pas écrire au même endroit. Cette version possède cependant une forte limite : le nombre de tâches soumises.

La Figure 4.18 représente l'évolution du nombre de tâches de la factorisation $PA = LU$ en fonction de la taille de la matrice avec une taille de tuiles de 960 en utilisant des tâches d'extraction et d'insertion de lignes dans le pire des cas (960 échanges de lignes par colonne et par itération k de la factorisation). Comme montré par cette figure, le nombre de tâches d'échange de ligne est

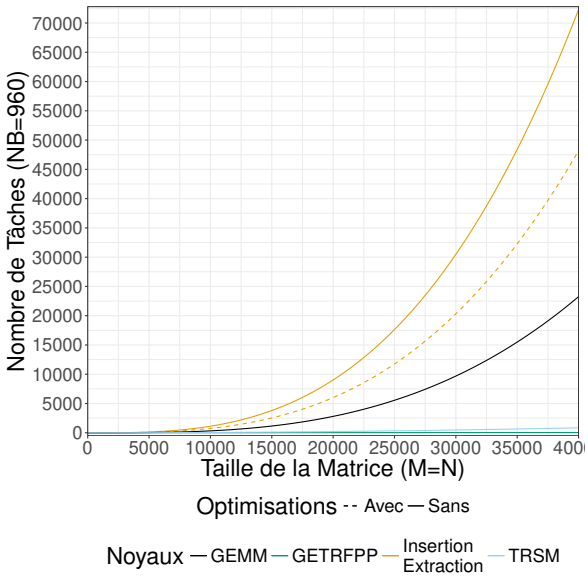


FIGURE 4.18 – Comparaison du nombre de noyaux soumis au total en fonction de la taille de la matrice.

très important, avec n_t le nombre de tuile dans une dimension, on a ${n_t}^3 - {n_t}^2$ extractions générées au total, et autant d'insertions. Il est possible d'optimiser cela en utilisant la version d'échange de lignes par colonne pour les permutations de ligne non critiques se situant en avant de la factorisation du panneau, représenté par la version avec tirets (optimisée) dans la Figure 4.18. Dans ce cas, le nombre d'insertions et d'extraction est de $2\frac{{n_t}^3}{3} - 2\frac{n_t}{3}$. C'est cette version qui est utilisée en pratique dans la partie expérimentale pour l'évaluation du pivotage par échange de ligne.

4.2.5 Évaluation expérimentale

L'évaluation de la factorisation $PA = LU$ présentée précédemment est conduite sur deux machines modernes de la plateforme PlaFRIM, l'Intel KNL 64 cœurs et une machine hétérogène composée de GPUs NVidia K40 et de 2 processeurs Intel Xeon E5-2680 v3 pour un total de 24 cœurs. Les expériences permettent de comparer plusieurs versions préliminaires de l'algorithme implantées dans Chameleon de la factorisation $PA = LU$ avec pivotage partiel avec plusieurs versions de référence comme représenté dans le Tableau 4.1. La principale version de référence est l'algorithme *nopiv* de Chameleon qui considère qu'aucun pivotage n'est à réaliser et n'utilise pas de panneau de factorisation. L'algorithme nopiv factorise la tuile diagonale, à la façon d'une factorisation de Cholesky, puis utilise des tâches TRSM sur la colonne et la ligne courante avant de mettre à jour le reste de la matrice avec des GEMM.

C'est donc un algorithme fortement parallèle qui n'est pas réaliste dans le cas d'une matrice à diagonale non strictement dominante. Dans le cas du KNL, ces versions sont aussi comparées à l'implantation DGETRF de la bibliothèque *MKL* et pour la machine hétérogène les résultats sont comparés à *MAGMA*.

Version	Type de matrice	Tâche parallèle	Acquire	Pivotage
MKL	aléatoire	?	?	oui
MAGMA	aléatoire	oui ? (panneau MKL)	non	oui
nopiv	DSD	non	non	non
pp-dsd-noacquire	DSD	oui	non	non
pp-dsd	DSD	oui	oui	non
pp-dsd-line	DSD	oui	oui	oui (non soumis)
pp-column	aléatoire	oui	oui	gros grain : toute la colonne
pp-line	aléatoire	oui	oui	grain fin : extraction/insertion de ligne

TABLE 4.1 – Caractéristiques des différentes versions de la factorisation $A = LU$ ou $PA = LU$

L'objectif de ces expériences est d'étudier les performances de l'approche et les limites des implantations préliminaires actuelles vis à vis des références établies. De plus, l'objectif de cette étude expérimentale est de réaliser une étude des impacts successifs sur les performances de l'utilisation de matrices de type diagonale strictement dominante ne nécessitant aucun pivotage (*dsd*) ou aléatoire (aucune mention), de l'utilisation de la tâche parallèle pour la réalisation du pivotage partiel (notée *pp*), de l'arrêt du flux de soumission des tâches (*acquire*) et de l'utilisation des tâches de pivotage à gros grain (*column*) ou à grain fin (*line*). Les versions de pivotage font référence aux versions expliquées dans la Figure 4.15. La version par colonne ne soumet qu'une seule tâche par colonne pour réaliser le pivotage, alors que la version par ligne fonctionne avec deux tâches, une d'insertion et l'autre d'extraction de lignes pour chaque tuile de la matrice. Il est important de noter que pour optimiser cette version, un mode de localité a été intégré dans StarPU qui force les tâches d'extraction et d'insertion à s'exécuter sur le nœud qui possède la tuile dont les lignes sont extraites/insérées. De cette façon, seuls de petits tableaux temporaires contenant les données à extraire ou insérer transitent au sein de la machine ce qui permet l'obtention d'un volume de données optimal.

KNL

La Figure 4.19 montre les performances de plusieurs versions de la factorisation $PA = LU$ avec pivotage partiel contre les versions de référence *nopiv* et MKL. La performance est montrée pour deux tailles de tuiles différentes pour tous les algorithmes sauf la MKL, 480 et 960. Le premier point important à remarquer est la performance impressionnante de la factorisation de MKL.

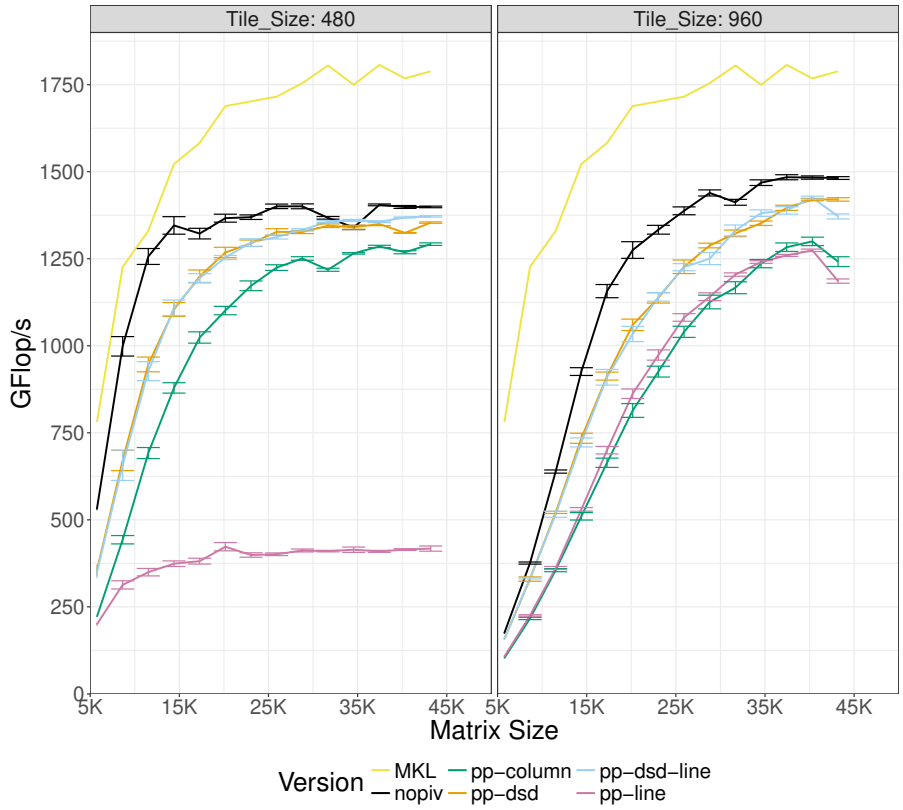


FIGURE 4.19 – Performances de la factorisation LU implantée dans `Chameleon` pour plusieurs variantes sur une Intel KNL. La version *nopiv* sert de référence.

Pour les tailles de matrices inférieures à 15K la version nopiv est capable de se rapprocher de la performance de la MKL avec une taille de tuile de 480 cependant pour les tailles de matrices plus grandes la MKL gagne un peu plus de 250 GFlop/s sur les autres versions, notamment la version *nopiv*. En ce qui concerne les autres versions, la version nopiv est constamment meilleure ce qui est normal car comme décrit précédemment, cette version fournit un parallélisme optimal en prenant pas en compte la recherche d'éléments maximum et l'échange de ligne pour augmenter la stabilité numérique de l'algorithme. De façon générale, utiliser une taille de tuile de 480 fournit des performances intéressantes pour les petites tailles de matrices et une performance maximale de 1400 GFlop/s, alors qu'augmenter la taille de tuile à 960 permet d'augmenter la performance maximale à 1480 GFlop/s.

La version la plus proche de *nopiv* utilisant une tâche parallèle pour le panneau, *pp-dsd* obtient des performances légèrement inférieures à *nopiv* notamment pour des matrices de taille faible à intermédiaire, mais la meilleure performance obtenue est de 1420 GFlop/s soit une différence d'environ 4.5%. L'utilisation du pivotage par ligne dans le cadre d'une matrice à diagonale stric-

tement dominante, *pp-dsd-line*, fourni des performances similaires à la version sans pivotage car aucune tâche de pivotage n'est soumise puisque ce n'est pas nécessaire. Ceci est possible grâce à l'arrêt du flot de soumission de tâches qui permet à l'algorithme de décider de la nécessite ou non de soumettre ces tâches de pivotage. Cependant, lors de l'utilisation de cet algorithme dans le cadre d'une matrice aléatoire nécessitant du pivotage pour chaque ligne de la matrice, les versions ligne comme colonne perdent environ 125 GFlop/s de performance. Ce phénomène est plus prononcé pour la version ligne avec une taille de tuile de 480 qui stagne en performances, peut être à cause des surcoûts de cette version et du nombre de tâches générées pour le pivotage.

Hétérogène

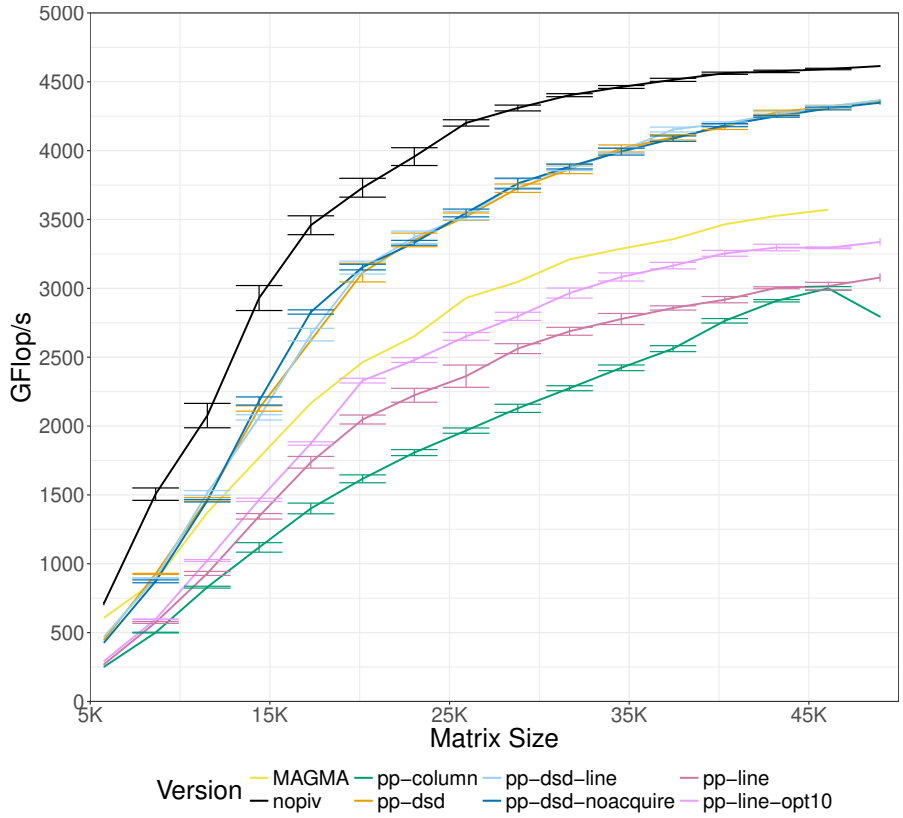


FIGURE 4.20 – Performances de la factorisation LU implantée dans `Chameleon` pour plusieurs variantes sur une machine hétérogène équipée de 4 GPUs. Les versions nopiv et MAGMA servent de référence.

La Figure 4.20 montre les performances de plusieurs versions de la factorisation $PA = LU$ avec pivotage partiel contre les références *nopiv* et *MAGMA*.

La version *nopiv*, puisqu'elle est créée pour l'obtention d'un parallélisme massif, fourni le plus de performances et obtient une performance maximale de 4613 GFlop/s. Au contraire, MAGMA est l'algorithme de pivotage partiel qui obtient la meilleure performance, avec un maximum de 3570 GFlop/s. En ce qui concerne les versions du pivotage partiel implantées, la version *pp-dsd-noacquire* qui ne réalise pas la suspension du processus léger de soumission des tâches et obtient une performance équivalente à *pp-dsd* qui a pour seule différence de réaliser cette suspension. On observe donc qu'il n'y a aucun surcoût observable dans ce cas à suspendre le processus léger de soumission pour décider de la soumission ou non de tâches de pivotage. Ceci a un fort intérêt, car dans le cadre d'une matrice à diagonale strictement dominante la version *pp-dsd-line* ne soumet aucune tâche de pivotage et obtient la même performance que les autres versions. Au final, les versions utilisant la tâche parallèle sans pivotage obtiennent une performance maximale de 4360 GFlop/s soit 5.5% de moins que la version *nopiv*. Lorsque l'on considère une matrice aléatoire nécessitant de nombreux pivotage, la version par colonne obtient la plus mauvaise performance car elle génère de très nombreux transferts de données et la version ligne sans optimisation obtient de meilleures performances. Cependant, la version ligne souffre d'un autre problème : un nombre important de tâches est créé comme observé précédemment dans la Figure 4.18 et sa performance stagne à partir d'un certain point (environ $30K$). Il est ainsi bénéfique d'optimiser cette version en utilisant du pivotage par colonne pour les parties distantes de la diagonale de la matrice. Dans ce cas précis, toutes les colonnes de tuiles situées à une distance de plus de 10 colonnes de la diagonale sont réalisées avec une seule tâche de pivotage. Les colonnes situées plus près de la diagonale sont réalisées par extraction et insertion de lignes. Ceci permet de fortement réduire les surcoûts et obtenir de meilleures performances, proches de celles de MAGMA avec un maximum obtenu de 3336 GFlop/s soit 6.6% de moins que MAGMA.

Un point critique de cet algorithme est de maîtriser le volume de données communiquées lors de la réalisation du pivotage. On observe dans la Figure 4.21 le volume total de données transféré au cours d'une exécution de plusieurs versions de la factorisation LU. Les versions *nopiv*, *pp-dsd* et *pp-line* communiquent avec 4 GPUs pour une matrice de taille $26K$ environ 30 Go de données au cours de la factorisation alors que la version colonne communique près de trois fois plus de données. Le pivotage par ligne permet donc de fortement optimiser les communications grâce à l'extraction et l'insertion précise des données utiles au pivotage. Pour rappel, les tuiles de la matrice sont fixées sur la machine pendant la durée de l'exécution de l'extraction et de l'insertion de ligne grâce à un mode de localité implanté spécialement dans StarPU pour cela. Cette conservation de la bande passante semble avoir un effet néfaste sur l'ordonnancement : bien que les données sont fixées sur la machine uniquement pour les tâches d'échange de lignes, ceci empêche StarPU et notamment son

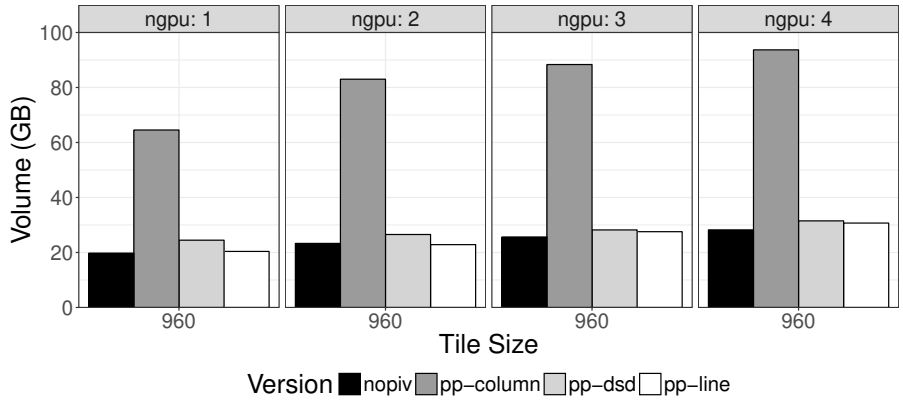


FIGURE 4.21 – Bande passante consommée par plusieurs variantes d'algorithmes de la factorisation LU pour une matrice de taille 25920×25920.

ordonnanceur, HEFT (dmda), de remettre en cause la répartition des données à certains points de l'algorithme d'ordonnancement et créé un déséquilibre de charge en faveur des GPUs ainsi qu'une légère baisse de la consommation en bande passante agrégée comparé à *pp-dsd* puisque les CPUs reçoivent un peu moins de travail dans la version *pp-line*.

La Figure 4.22 montre des histogrammes des durées des noyaux d'extraction et d'insertion de lignes sur CPUs et GPUs pour le pivotage. On observe d'après ces histogrammes que les noyaux obtiennent un temps d'exécution moyen de $1.6ms$ sur CPU et de $60\mu s$ sur GPU. Ceci explique l'existence de surcoûts à l'utilisation de cette version, car non seulement un nombre important de tâches est généré mais en plus ces tâches ont une durée extrêmement faible sur GPU. En effet, la documentation de StarPU recommande d'utiliser des tâches avec un temps d'exécution de l'ordre de $1ms$ pour permettre aux ordonnanceurs et à StarPU de passer à l'échelle peu importe le nombre de tâches. C'est pourquoi l'utilisation d'une simple optimisation comme présenté précédemment mélangeant pivotage par colonne et par ligne permet des gains de performance significatifs, car à partir d'un certain nombre de tâches de la sorte les surcoûts deviennent détrimentaux pour les performances.

4.2.6 Discussion

L'implantation efficace de l'algorithme $PA = LU$ n'est pas naturelle en utilisant un modèle de tâches pures. En effet, la factorisation du panneau est critique pour les performances, et nécessite des recherches de maximum dans les éléments sous-diagonaux de la matrice ce qui rend son implantation inefficace avec des tâches pures ou sur accélérateur tel un GPU. De plus, les échanges de lignes générés par les permutations sont coûteux et créent des points de

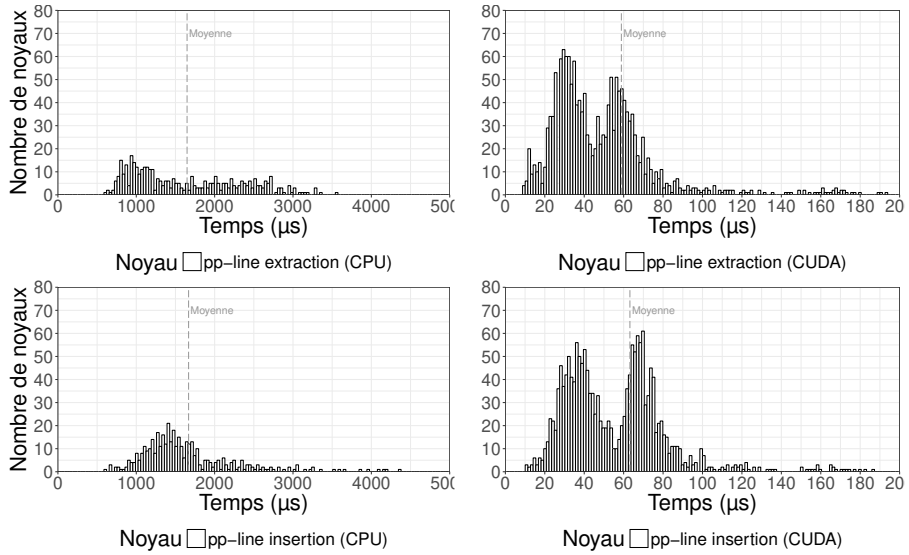


FIGURE 4.22 – Nombre de noyaux d'extraction et d'insertion de ligne avec une certaine durée (en μs) pour la factorisation $PA = LU$ avec échange de lignes pour une matrice de taille 14400×14400.

synchronisations importants dans le graphe de tâches.

L'implantation de cet algorithme permet de mettre en valeur plusieurs nouveautés de StarPU. Pour factoriser efficacement le panneau, une implantation en tâche parallèle du panneau est utilisée. L'utilisation de tâches parallèles avec plusieurs synchronisations internes et échanges de données internes, combinées à un nombre variable de dépendances permet de dépasser les limites du mode de tâches pures. De plus, pour le calcul des échanges de lignes, un système "d'acquire-release" permet de soumettre ceux-ci uniquement lorsque nécessaire. Les échanges de lignes sont optimisés grâce à l'ajout d'un système forçant l'exécution des tâches d'échanges de lignes sur le nœud mémoire où est localisé la tuile de la matrice source ou destination des contributions.

Les performances obtenues montrent que l'utilisation de ce système "d'acquire-release" n'est pas pénalisant pour l'algorithme. L'utilisation de tâches parallèles dans sa forme actuelle implique une légère perte de performances par rapport à l'algorithme de référence *nopiv* cependant les performances obtenues avec pivotage sur la machine hétérogène sont similaires à *MAGMA*, ce qui montre l'intérêt de ces techniques. Enfin, lorsque cet algorithme est utilisé sur une matrice ne nécessitant aucun échange de lignes, les performances obtenues sont les mêmes que dans le meilleur des cas, prouvant la bonne capacité d'adaptation de cette algorithme aux caractéristiques de la matrice factorisée. Cependant, plusieurs améliorations pourraient être réalisées afin d'augmenter les performances obtenues.

Amélioration de la tâche parallèle

Une première amélioration possible pour augmenter les performances de l'algorithme consiste en l'optimisation de la tâche parallèle réalisant le calcul du panneau, en effet cette tâche est implantée en pthreads et assigne un processus léger à chaque tuile. Cette parallélisation à très gros grain est problématique car si le panneau n'est pas assez grand, ce qui arrive lorsque l'on considère des matrices de petites tailles ou bien la partie finale de l'exécution de l'algorithme, il n'y a pas assez de tuiles pour utiliser tous les processus légers. Ainsi, certaines ressources deviennent inactives et le panneau n'est pas autant accéléré qu'il pourrait l'être ce qui peut être pénalisant car cette tâche est fortement critique pour cet algorithme. D'autre part, l'implantation actuelle utilise de l'algorithme un "look-ahead", une anticipation de la soumission de la tâche de factorisation du panneau, de niveau 1, augmenter ce niveau pourrait être également améliorer les performances.

Perspectives sur le contrôle du graphe de tâches

L'utilisation de la technique "acquire-release" force la suspension du processus léger de soumission, cela peut être à l'origine d'une pénurie de tâches prêtes. Dans l'exemple du $PA = LU$ cet effet est compensé par l'utilisation de technique d'anticipation ("lookahead"), et les expériences montrent que l'impact reste limité. Cependant, de façon générale, la technique "acquire-release" nuit à la soumission des tâches futures et oblige donc à optimiser l'ordre de soumission des tâches sous peine de perte de performances. Ceci met en avant l'intérêt de mettre au point de nouvelle techniques de contrôle et d'adaptation du graphe de tâches plus souples. Une solution à ce problème pourait être l'utilisation de tâches hiérarchiques. Ces tâches permettent de soumettre des tâches qui à l'exécution soumettront de nouvelles tâches. Utiliser ce modèle permettrait d'adapter le graphe soumis selon les données et ainsi obtenir une adaptation du graphe de tâches plus naturelle sans impacter la soumission des tâches.

Perspectives sur l'accès aux données

Un point clé de cet algorithme comme le montre les performances, surtout sur machine hétérogène, est l'implantation efficace de techniques de pivotage. Les techniques utilisées ont toutes leurs limites : le pivotage par colonne génère peu de tâches mais n'est pas parallélisé et requiert de nombreux transferts de données, alors que le pivotage par ligne est très efficace en temps et en volume de données mais génère de nombreux surcoûts à cause de son temps trop court et du nombre de tâches générées. Accessoirement, de nombreux tableaux temporaires doivent être alloués, enregistrés à StarPU puis désenregistré ce qui peut avoir un coût important et complexifie l'algorithme. Il serait intéressant

que le support d'exécution fournisse un moyen d'exprimer cette opération de façon plus naturelle. Il peut être par exemple envisagé, pendant le temps de l'opération, de fixer les données sur la machine (de la même façon que la localité des échanges de ligne est forcée actuellement) et de réaliser pour les contributions dirigées vers la tuile supérieure une opération de type "gather" sur chaque nœud mémoire, avant de rassembler ces données partielles dans la donnée finale. Il serait même possible d'utiliser un mécanisme d'écriture concurrente pour cette opération finale, puisque les lignes à échanger ne touchent pas aux mêmes portions de la tuile de destination des contributions.

4.3 Synthèse et perspectives pour le MPI

Dans ce chapitre, on a étudié l'utilisation de tâches parallèles dans le cadre de deux applications, l'application FLUSEPA un code de mécanique des fluides développé par Airbus DS et la factorisation $PA = LU$ avec pivotage partielle au dessus du support d'exécution StarPU. Il a été observé comment l'utilisation de tâches parallèles permet à FLUSEPA de jouer sur deux paramètres, la granularité des tâches et la granularité des ressources, simultanément pour obtenir de meilleures performances dans tous les cas. L'utilisation d'une tâche parallèle pour l'algorithme $PA = LU$ permet de réaliser les nombreuses synchronisations nécessaires à la factorisation du panneau de la matrice dans une seule tâche, ce qui permet d'alléger le travail du support d'exécution. Cette implantation permet l'obtention de performances proches de la version tuilée de référence sans pivotage de l'algorithme et permet de plus de s'adapter à la structure de la matrice factorisée et d'obtenir des performances compétitives avec MAGMA dans le cadre d'une plateforme hétérogène. Plusieurs améliorations ont été proposées pour rendre cet algorithme plus compétitif et augmenter les performances obtenues.

Un point de discussion central, notamment pour l'algorithme $PA = LU$ concerne le passage à l'échelle en MPI de cet algorithme. Ceci nécessite de nouvelles innovations qui restent à explorer pleinement. Le problème principal concerne à nouveau la tâche parallèle calculant le panneau. Comme observé précédemment, il est illusoire de réaliser la factorisation du panneau avec un modèle de tâches pures du fait des très nombreuses synchronisations requises pour chaque colonne du panneau et encore moins lorsque ceci se passe au dessus de plusieurs nœuds, nécessitant l'utilisation d'appels MPI pour chaque tâche et de nombreuses réductions.

L'extension du modèle de tâches parallèles au distribué peut rendre possible une factorisation efficace du panneau dans un contexte distribué. De plus, des implantations très performantes en mode distribué tel que HPL [2] pourraient être utilisées comme implantation de la tâche parallèle distribuée. Plusieurs aspects intéressants sont alors à étudier. Comment contraindre la bibliothèque

HPL à s'exécuter uniquement sur certains nœuds, et composer son exécution avec d'autres tâches et supports d'exécution, éventuellement impliquant l'utilisation de bibliothèques MPI différentes ? Quel surcoût pour la version tâche parallèle distribuée par rapport à la non distribuée, et quand est-il bénéfique d'utiliser l'une ou l'autre version. Enfin, quelle distribution des données sur les nœuds est optimale et quel impact a le mode "2D bloc cyclique" sur les performances d'un panneau implanté grâce à une tâche parallèle distribuée.

Chapitre 5

Plateformes pour l'ordonnancement de tâches moldables

Sommaire

Dans ce chapitre est présenté plusieurs travaux autour de l'ordonnancement de tâches parallèles moldables. L'objectif de ce chapitre est d'étudier la pertinence de la composition de groupes de cœurs de tailles différentes. Cette

étude est conduite en deux temps, 1) dans le cadre de groupes de cœurs fixes en Section 5.1 et 2) dans le cadre de groupes de cœurs hétérogènes dynamiques dans les sections suivantes.

Dans le cadre de l'étude des tâches moldables avec groupes de cœurs hétérogènes fixes en Section 5.1, l'ordonnanceur HeteroPrio présenté en Section 2.2.2 est adapté à l'utilisation de groupes de cœurs. Pour cela une nouvelle métrique est présentée, le score d'accélération, une généralisation du concept de facteur d'accélération. Une étude exhaustive est ensuite conduite pour étudier les comportements des ordonnanceurs Heft et HeteroPrio lors de l'utilisation de groupes de cœurs non homogènes en simulation et ces résultats sont validés en exécution réelle.

La seconde contribution concerne la réalisation d'ordonnanceurs dynamiques (online) pour décider de l'attribution de plusieurs ressources aux tâches. Pour cela, ces ordonnanceurs doivent décider dynamiquement de quelle quantité de ressources assigner à chaque tâche avant d'assigner cette tâche sur plusieurs ressources. En Section 5.2 sont proposés des outils pour l'expression de ces nouveaux types d'ordonnanceurs. Ensuite, plusieurs stratégies d'ordonnancement possibles sont exposées dans la Section 5.3.

5.1 Adaptation d'ordonnanceurs basés sur les facteurs d'accélération

Habituellement, les supports d'exécutions et notamment StarPU dépendent de stratégies gloutonnes pour l'ordonnancement de tâches comme présenté précédemment. Nous avons pu observer dans le Chapitre 3 que ces types d'ordonnanceurs ont tendance à trop favoriser les GPUs et que rendre les CPUs compétitifs avec l'utilisation de ressources parallèles permet une meilleure utilisation de celles-ci. En effet, quand le facteur d'accélération est élevé, les tâches prêtes ont tendance à être toujours allouées sur les accélérateurs, même quand elles sont éloignées du chemin critique et pourraient être exécutées sur une ressource plus lente. Ce même constat a aussi été réalisé dans [10]. Afin de contrer ce problème un nouvel ordonnanceur nommé HeteroPrio a été proposé [16], celui-ci repose sur l'affinité entre les tâches et les ressources. Cet ordonnanceur a aussi été amélioré dans [10] avec plusieurs corrections. Il a été démontré que, quand elle est bien paramétrée, cette stratégie d'ordonnancement est plus efficace que les stratégies de type glouton en présence de CPUs et GPUs pour la factorisation de Cholesky. Son désavantage majeur est que cela est limité à deux types de ressources, *i.e.* une ressource lente de type CPUs et une ressource rapide de type GPUs. Dans un premier temps, on présente plus en détail HeteroPrio en section 5.1.1. Dans un second temps, on présente le score d'accélération et la généralisation de HeteroPrio à plus de deux classes de ressources en section 5.1.2. Enfin, des expériences sont réalisées en

section 5.1.3 afin de démontrer la pertinence de l'approche et comparer HEFT à HETEROPRIO avec et sans groupes de cœurs. Il est notamment intéressant de combiner et comparer l'utilisation de HETEROPRIO avec l'utilisation de tâches parallèles puisque ces deux techniques s'attaquent au même problème. Ces travaux résultent d'une collaboration avec Suraj Kumar qui a soutenu sa thèse le 12 avril 2017 [77].

5.1.1 HETEROPRIO, un ordonnanceur basé sur l'affinité entre les tâches et les ressources

Une version simplifiée de l'ordonnanceur HETEROPRIO a été présentée en Section 2.2.2. Cet algorithme d'ordonnancement a notamment reçu plusieurs améliorations dans [10] qui seront utilisées dans le cadre de cette étude. Tout d'abord, afin d'éviter de créer du délai pour les tâches sur (ou proches) du chemin critique, il est important s'assurer que la majeur partie de celles-ci sont accélérées en les exécutant sur la meilleure ressource. Ceci est fait en triant chaque file de tâches par priorités, calculées comme la distance entre cette tâche et le nœud de sortie du graphe. Selon le facteur d'accélération sur GPUs, les plus grandes priorités sont généralement assignées au GPU tandis que les plus faibles sont données aux CPUs afin de ne pas créer de délais dans le traitement des tâches critiques. Ce compromis entre l'affinité et la priorité des tâches est renforcé par une seconde amélioration : les files des GPUs sont fusionnées lorsque les facteurs d'accélération sont proches, afin que l'algorithme favorise mieux sur les tâches plus prioritaires. Par exemple pour la factorisation de Cholesky, les files des noyaux DGEMM et DSYRK sont en pratique fusionnées pour les GPUs car les facteurs d'accélération pour ces deux tâches sont similaires. Pour les CPUs, quatre files sont créées, soit une pour chaque noyau *i.e.* DGEMM, DSYRK, DTRSM et DPOTRF.

Enfin, la dernière optimisation concerne l'ajout d'un mécanisme de spoliation. À chaque fois qu'un GPU est inactif alors que des tâches possédant un facteur d'accélération très élevé (*i.e.* DGEMM et DSYRK), le GPU redémarre l'exécution de cette tâche si cela lui permet de la finir plus rapidement. Dans la suite on utilise en pratique cette version modifiée de HETEROPRIO.

Implantation de HETEROPRIO modifié grâce à une simulation hors-ligne

En pratique, arrêter l'exécution des noyaux, comme le requiert l'utilisation de la spoliation, peut être une difficulté technique particulièrement au niveau de la cohérence des données. Afin de pouvoir utiliser ce type d'ordonnancement en pratique, une simulation est utilisée pour calculer un ordonnancement hors-ligne. Cette technique est mise en œuvre en passant un graphe de tâches et les temps des noyaux en entrée d'un programme qui implante l'ordonnanceur

lui-même. La sortie spécifie quelle ressource précise a exécuté quelle tâche, l'ordre entre ces tâches ainsi qu'une estimation de la performance. Pour exécuter cet ordonnancement calculé hors-ligne en pratique dans StarPU et explorer le comportement réel de celui-ci on utilise la sortie de la simulation, *i.e.* l'ordre d'exécution des tâches et les ressources sur lesquelles elles ont été exécutées, et l'on utilise cela comme entré d'un ordonnanceur chargé de respecter cette description. Cet ordonnanceur permettant de respecter la spécification calculée hors-ligne est présenté dans la section expérimentale 5.1.3.

L'utilisation de simulations a plusieurs avantages mais aussi des inconvénient. Le premier avantage est que l'utilisation de simulations permet de réaliser la correction nécessaire de l'ordonnancement grâce au mécanisme de spoliation présenté précédemment, qui n'est pas réalisable facilement en pratique. Le second avantage est qu'il est possible d'explorer rapidement beaucoup d'ordonnancements et de configurations de machines et d'en obtenir une estimation de la performance puisque le graphe n'est pas exécuté réellement. Le principal inconvénient est que la simulation utilisée ne permet pas de prendre en compte les communications et manque de précision dans les cas où ces communications sont critiques car non recouvertes (notamment pour de petites tailles de matrices avec la factorisation de Cholesky), ceci impliquera plusieurs adaptations au moment de l'exécution réel de l'ordonnancement pré-calculé.

5.1.2 Scores d'accélération pour l'adaptation de HeteroPrio à plus de deux types de ressources

Adapter cet algorithme au cas où l'on utilise plus de deux ressources n'est pas évident, en particulier à cause de la notion centrale dans cet algorithme de facteur d'accélération qui n'a plus de sens lorsque l'on utilise plus de deux ressources. Il est alors nécessaire d'identifier une nouvelle façon de décider des tâches qui doivent être favorisées pour être exécutées sur chacune des ressources en présence. Dans cette section, on présente deux façons possibles de calculer des scores dont l'objectif est la généralisation du concept de facteur d'accélération. Le principe général de HeteroPrio reste inchangé, dès qu'une ressource est libre, elle pioche une tâche prête parmi le type de tâches qui possède le meilleur score.

Le score d'`Aire`

On nomme le premier système de score d'`Aire` car il repose sur la généralisation du concept nommé *borne d'aire* dans le cas d'une machine homogène. L'idée est de calculer l'allocation des tâches qui minimise le temps d'exécution global lorsque l'on ignore les dépendances, en assumant que toutes les unités de calcul travaillent sans temps d'inactivité ni surcoût. Cette allocation peut être obtenue grâce à la résolution d'un petit programme linéaire [12] et créée

une façon générique de détecter quelles tâches sont les plus adaptées à quelles ressources. Dans le système `Aire`, le score de chaque type de tâches t pour une ressource r est simplement la proportion de tâches de type t que la ressource r exécuterait dans ce cadre idéal. Dans le cas avec deux ressources, les proportions sont assignées selon l'ordre fourni par le facteur d'accélération entre ces deux ressources. Ainsi, ce système de score généralise le comportement d'origine de HeteroPrio.

Le score *Indice d'Hétérogénéïté*

Le second système de score se nomme *Indice d'Hétérogénéïté* (`Indice H.`) et il se calcule de la façon suivante. Soit T l'ensemble des types de tâches, R l'ensemble des ressources, et $E(t, r)$ le temps d'exécution d'une instance de tâche de type t sur la ressource r, avec $t \in T$ et $r \in R$. Pour chaque type de tâches t, on note le temps d'exécution maximal $E_{\max} = \max_{i \in R} E(t, i)$ et le temps d'exécution minimal $E_{\min} = \min_{i \in R} E(t, i)$. On défini $\texttt{IndiceH.}(t, r) = \frac{E_{\max} \times E_{\min}}{E(t,r)^2} = \frac{E_{\max}}{E(t,r)} \times \frac{E_{\min}}{E(t,r)}$ et on utilise $\texttt{IndiceH.}(t, r)$ comme un score pour décider quelle type de tâches favoriser pour la ressource r. L'idée amenant à cette définition est que le premier terme ($\frac{E_{\max}}{E(t,r)}$) représente à quel point cette ressource est "meilleure" comparée à la pire possible, et le second terme quantifie à quel point cette ressource est "mauvaise" comparée à la meilleure. Ce score est aussi une généralisation du facteur d'accélération : avec des GPUs et CPUs uniquement, l'indice d'hétérogénéïté des GPUs est égal au facteur d'accélération, et pour les CPUs celui-ci est égal à l'inverse du facteur d'accélération.

Considérations sur l'adaptation de HeteroPrio

Comme mentionné précédemment, il est important de prendre en compte les priorités des tâches en s'assurant que les ressources dites "rapides" prennent en charge les tâches avec les priorités les plus élevées. Caractériser ce que "rapide" signifie est évident dans le cas où il n'y a que deux ressources. Pour généraliser ceci au cas où il y a plus de deux ressources, on propose l'approche suivante.

Pour une ressource r, on calcul la moyenne géométrique μ_r des temps d'exécution de toutes les tâches sur cette ressource ($\mu_r = \left(\prod_{t \in T} E(t, r)\right)^{\frac{1}{|T|}}$). Cette moyenne géométrique mesure la vitesse agrégée de tous les types de tâches pour la ressource r. Ensuite, on calcule la moyenne (arithmétique) de ces μ_r de chaque ressource et l'on classifie une ressource comme "rapide" si sa valeur μ_r est inférieure à la moyenne, et dans le cas contraire on la qualifie comme "lente". Les ressources dites "rapides" prennent en charge les tâches avec les priorités les plus élevées, et dans le HeteroPrio considéré peuvent aussi utiliser le mécanisme de spoliation sur les ressources "lentes".

De plus, comme mentionné précédemment, dans HeteroPrio les priorités élevées sont favorisées grâce à la fusion de files des types de tâches possédant un facteur d'accélération similaire sur les GPUs. Ce principe est généralisé sur les ressources "rapides", en fusionnant les files des tâches avec des scores similaires. Il a été observé de façon empirique que la meilleur valeur pour ce paramètre est de fusionner les files lorsque la différence de leurs scores est de moins de 25% du plus grand.

Utilisation des deux scores avec un exemple

Afin de comprendre le principe de fonctionnement des deux systèmes de scores (`Aire` et `Indice H.`) et d'exposer leur différence, considérons plusieurs instances de deux types de tâches (*T1* et *T2*) sur trois ressources différentes (*R1*, *R2* et *R3*). La Table 5.1 montre les temps d'exécutions des deux types de tâches sur toutes les ressources (Table 5.1a). Elle montre aussi les scores obtenus avec les systèmes `Aire` (Table 5.1b) et `Indice H.` (Table 5.1c) pour les deux types de tâches sur toutes les ressources.

	T1	*T2*
R1	100	200
R2	120	60
R3	200	75

(a) Temps d'exécution

	T1	*T2*
R1	60	0
R2	40	20
R3	0	80

(b) Score `Aire`

	T1	*T2*
R1	2.0	0.3
R2	1.4	3.3
R3	0.5	2.1

(c) Score `Indice H.`

Table 5.1 – Temps d'exécution en utilisant les scores Aire et `Indice H.` sur différentes ressources pour différents types de tâches.

Sur la ressource R1, pour les deux systèmes de scores, le score des tâches de type T1 est plus élevé que celui des tâches de type T2, ainsi R1 préfère les tâches de type T1 pour les deux systèmes de scores. De façon similaire, les tâches de type T2 ont un score plus élevé que celles de type T1 sur la ressource R3, et donc cette ressource préfère les tâches de type T2. Cependant, dans le système de score `Aire`, la ressource R2 préfère les tâches de type T1 mais `Indice H.` favorise les tâches de type T2 car il leur attribue un score `Indice H.` plus élevé.

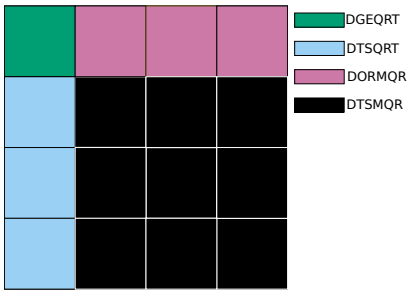
5.1.3 Étude expérimentale, Heft vs HeteroPrio

Afin d'évaluer les performances des heuristiques proposées on propose plusieurs expériences. Tout d'abord, la plateforme considérée est composée de deux processeurs Haswell Intel® Xeon® E5-2680 possédant 12 cœurs chacun

ainsi que quatre GPUs NVIDIA® K40-M. Comme observé précédemment, la plupart des supports d'exécution utilisent un cœur dédié pour la gestion de chaque GPU. Par conséquent, on peut considérer cette machine comme ayant 20 workers CPUs et 4 workers GPUs de disponibles. Pour les expériences présentées, les résultats ont été obtenus courant 2016 et sont moins récents que les résultats du Chapitre 3 ce qui peut expliquer certaines petites différences de comportement. Les résultats sont évalués sur deux opérations, la factorisation de Cholesky et la factorisation QR de Chameleon [15] qui s'exécutent au dessus du support d'exécution StarPU. La factorisation de Cholesky a déjà été présentée dans les Section 2.3.1 et section 3.3. La factorisation QR de Chameleon utilisée est la méthode QR "Householder" qui est composée de quatre noyaux, DGEQRT, DTSQRT, DORMQR, DTSMQR, dont l'utilisation est expliquée dans la Figure 5.1. La Figure 5.2 présente les graphes de tâches pour les deux applications Cholesky et QR. Cette étude expérimentale est composée de trois étapes : d'abord, on présente le comportement des noyaux (notamment des noyaux QR), ensuite, on évalue les différentes heuristiques d'ordonnancement en utilisant une simulation et enfin, on observe la performance des meilleures configurations identifiées sur de vraies exécutions.

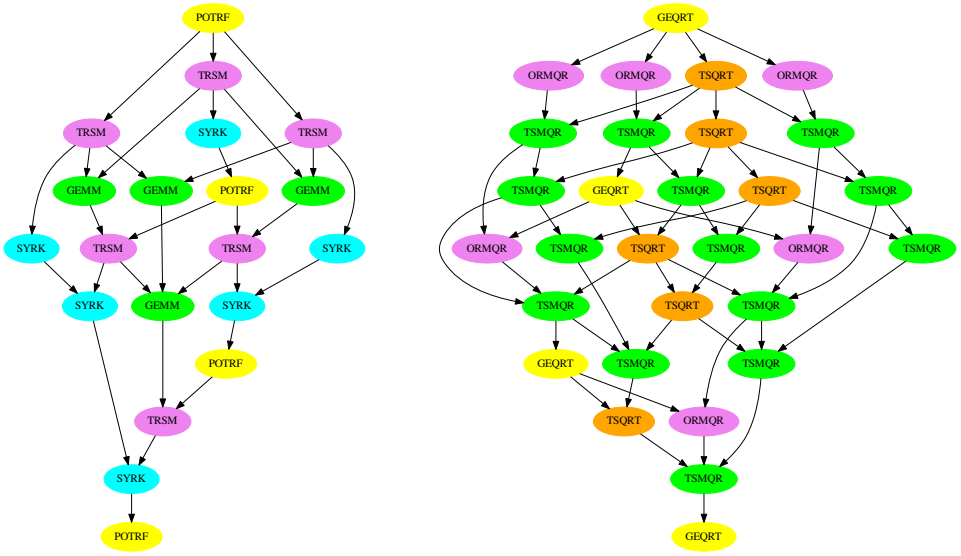
```
for k : 0 → min(MT-1, NT-1)
  submit(DGEQRT, A_{k,k}:RW, T_{k,k}:RW)
  for n : k+1 → NT-1
    submit(DORMQR, A_{k,k}:R, T_{k,k}:R,  A_{k,n}:RW)
  for m : k+1 → MT-1
    submit(DTSQRT, A_{k,k}:RW, A_{m,k}:RW, T_{m,k}:RW)
    for n : k+1 → NT-1
      submit(DTSMQR, A_{k,n}:RW, A_{m,n}:RW, A_{m,k}:R, T_{m,k}:R)
```

(a) Algorithme de la factorisation QR utilisée



(b) Factorisation QR d'une matrice à l'étape $k = 0$

FIGURE 5.1 – Présentation de la factorisation QR utilisée



(a) DAG de la factorisation de Cholesky

(b) DAG de la factorisation QR

FIGURE 5.2 – DAG de tâches des deux applications pour une matrice de taille 4×4 tuiles.

Étude des noyaux des factorisations Cholesky et QR

La Table 5.2 montre la performance des noyaux composant les factorisations de Cholesky et QR. On observe d'après ce tableau que le GPU est plus approprié pour certains types de tâches (*e.g.* DGEMM et DTSMQR) que d'autres, notamment les tâches DPOTRF, DGEQRT et DTSQRT ont une faible accélération sur GPU (inférieure à 2), ce qui rend la plateforme fortement hétérogène du point de vue des ordonnanceurs. Cependant, comme présenté au Chapitre 3, il est possible de rendre la plateforme moins hétérogène via l'utilisation de tâches parallèles, c'est à dire en assignant plusieurs ressources à un même noyau. On observe d'après la Table 5.2 que certains noyaux sont bien scalables (*i.e.* DGEMM, DSYRK, DTSMQR), certains ont une scalabilité acceptable (*i.e.* DPOTRF, DTRSM, DORMQR) et finalement d'autres ont une mauvaise scalabilité (*i.e.* DGEQRT, DTSQRT). L'explication de ces problèmes de scalabilité est la façon dont les noyaux QR sont implémentés dans Chameleon, en effet les noyaux de cette factorisation QR sont sous-découpés en tuiles internes (IB) et le meilleur IB trouvé pour la performance et la scalabilité de chaque noyaux est de 128. En interne, plusieurs appels BLAS de cette taille sont réalisés ce qui limite fortement la scalabilité des noyaux en version parallèle et donc la taille des groupes de cœurs. Cependant, on remarque que pour la factorisation

de Cholesky on est plus flexibles sur l'utilisation de tâches parallèles car tous les noyaux ont une scalabilité moyenne à bonne.

	Cholesky				QR (IB=128)			
	DPOTRF	DTRSM	DSYRK	DGEMM	DGEQRT	DORMQR	DTSQRT	DTSMQR
1 core (GFlop/s)	27.78	34.42	31.52	36.46	22.08	33.78	17.63	32.93
GPU / 1 core	1.72	8.72	26.96	28.80	1.91	15.90	1.87	14.64
10 cores / 1 core	5.55	6.75	6.90	7.77	1.65	4.10	0.73	6.94
5 cores / 1 core	4.20	4.50	4.66	4.49	1.67	3.30	1.25	4.05
2 cores / 1 core	1.88	1.95	1.93	1.94	1.33	1.77	1.16	1.91

TABLE 5.2 – Facteurs d'accélération des noyaux composant les factorisations de Cholesky et QR considérées. La performance des noyaux est rapportée à la performance d'un cœur et la taille de tuile est de 960.

Comparaison exhaustive des ordonnanceurs HEFT et HETEROPRIO en simulation avec les factorisations de Cholesky et QR

On conduit ici une exploration exhaustive de toutes les façons possible de créer des groupes de cœurs et on présente leur performance sur plusieurs ordonnanceurs. Les résultats obtenus sont montrés dans les Figures 5.3 et 5.4 pour les factorisations de Cholesky et QR. Chaque colonne représente un algorithme d'ordonnancement particulier, et chaque ligne correspond à une taille de matrice, exprimé comme le nombre de tuiles de taille 960 dans chaque colonne et dans chaque ligne.

En ordonnée on trouve les performances obtenues, exprimées en GFlop/s et en abscisse on représente les groupes de cœurs donnant un certain nombre de ressources, compris entre 2 et 20 groupes de cœurs au total. Pour 2 ressources, la configuration représentée est 2 groupes de cœurs de taille 10 (2×10, ou encore $(10)-(10)$) et pour 20 la configuration est 20×1, autrement dit les cœurs sont utilisés de façon indépendante. Prenons le cas où l'on a 3 ressources au total, on observe plusieurs groupes de cœurs donnant ce résultat tel que 2 groupes de cœurs de 5 cœurs chacun et un groupe de cœurs de 10 cœurs $(5,5)-(10)$ ou encore $(6,4)-(10)$ ou bien $(1,9)-(10)$, ... La seule règle suivie pour créer les configurations de groupes de cœurs est que cette configuration doit avoir du sens du point de vue de la topologie de la machine : on a deux processeurs 10 cœurs disponibles et l'on interdit à un même groupe de cœurs de s'exécuter sur deux processeurs simultanément, ainsi des configurations comme $(16)-(2,2)$ ou $(7)-(6,6)$ ne sont pas représentées puisque dans ces cas un groupe de cœurs empiète sur deux processeurs. Pour faciliter la comparaison, on ajoute une ligne sur chaque graphe montrant la performance où les cœurs des CPUs sont utilisés de façon indépendante (20×1).

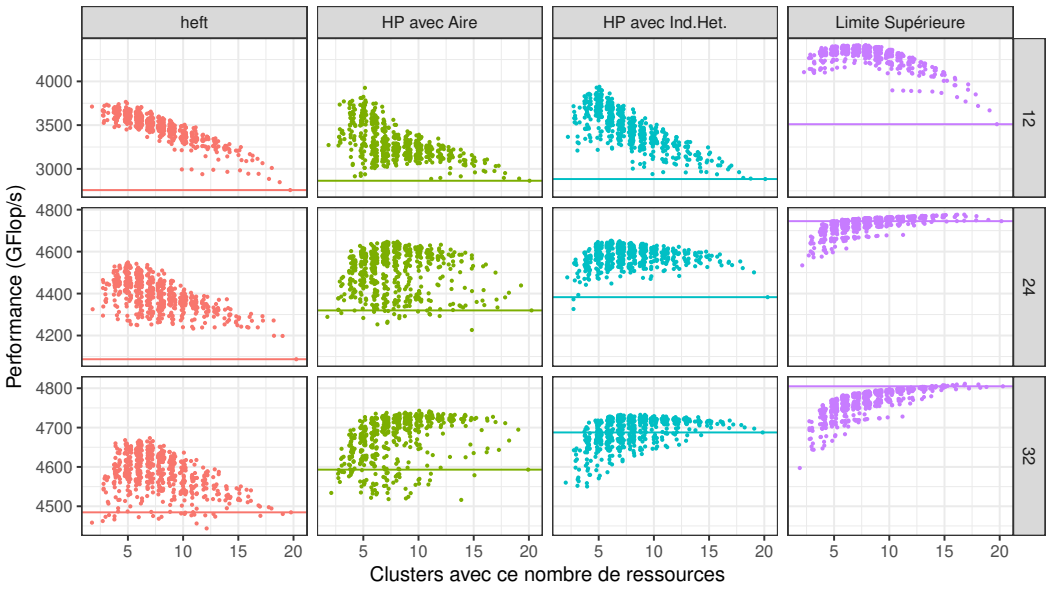


FIGURE 5.3 – Résultats de performance avec toutes les configurations pour la factorisation de Cholesky

Pour la factorisation de Cholesky dont les résultats se trouvent en Figure 5.3, on observe que les variantes HETEROPRIO obtiennent de meilleurs performances que Heft pour toutes les tailles de matrices considérées. L'algorithme Heft a besoin d'un faible nombre de groupes de cœurs pour obtenir de bonnes performances montrant que cet algorithme ne gère pas bien l'hétérogénéïté de la machine, même avec des matrices de grande taille. Par opposition, les variantes de HETEROPRIO sont capables de bien utiliser l'hétérogénéïté des configurations et de la machine et obtenir de meilleures performances, sauf pour de petites tailles de matrice où les performances sont comparables à Heft. Pour le cas des petites tailles de matrices il est intéressant de noter que la performance de la Limite Supérieure s'affaiblie lorsque l'on a beaucoup de groupes de cœurs, ce qui indique que la perte de performance observée pour toutes les versions s'explique par la nature du graphe de tâches : la performance dans ce cas est limitée par le chemin critique du graphe et faire usage de groupes de cœurs est nécessaire pour obtenir de bonnes performances. En général cependant, la Limite Supérieure ne permet pas de prédire quelles configurations peuvent obtenir de meilleurs performances pour les algorithmes d'ordonnancement.

On observe aussi que la performance de `HP avec Index Het.` est plus stable que la performance de `HP avec Aire` et qu'il y a un nombre important de configurations pour lesquelles `HP avec Aire` obtient de moins bonnes performances que `HP avec Index Het.` ; cependant leurs meilleurs configu-

rations sont comparables. Cette meilleure performance obtenue par `HP avec Index Het.` peut s'expliquer par le fait que le score utilisé par `HP avec Aire` est basé sur une vue globale du graphe de tâches sans dépendances pour calculer la répartition des tâches. Cette répartition serait parfaite si les tâches étaient indépendantes, cependant la répartition idéale change au cours de l'exécution du fait des dépendances. De plus, pour chaque ressource, cette répartition génèrent souvent de fausses égalités entre les types de tâches. Par exemple, deux types de tâches qui ne sont pas adaptés à une ressource obtiendraient toutes les deux le score de 0 puisque 0% de ces tâches devraient être exécutées sur ces ressources. L'ordonnanceur `HP avec Aire` traite ainsi les deux types de tâches de façon indifférente, alors qu'une d'entre elle peut malgré tout être beaucoup plus inefficace que l'autre ce qui implique une moins bonne performance que `HP avec Index Het.` dans certains cas.

On observe aussi que créer des groupes de cœurs n'est parfois pas bénéfique et certaines configurations obtiennent de moins bonnes performances que la référence où les cœurs sont utilisés de façon indépendante, ce qui indique que la performance dépend non seulement des tâches critiques mais aussi de l'efficacité des tâches. Cependant pour tous les algorithmes d'ordonnancement considérés on observe un nombre significatif de configurations, y compris pour `HP avec Aire` et `HP avec Index Het.` dont les performances sont meilleures que la référence.

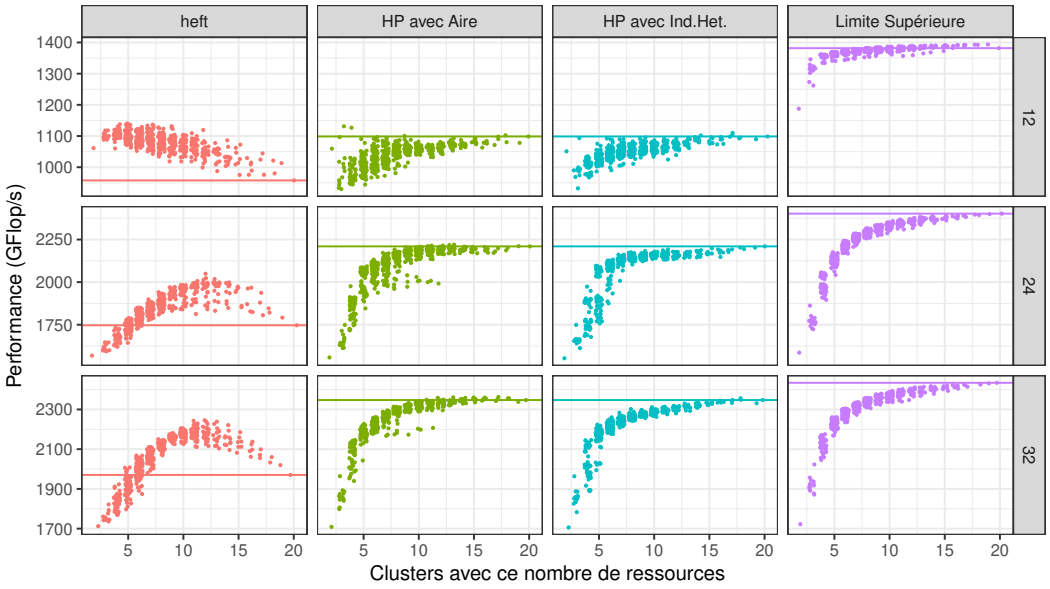


FIGURE 5.4 – Résultats de performance avec toutes les configurations pour la factorisation QR

Des observations similaires peuvent être réalisées pour la factorisation QR

dont les résultats se trouvent dans la Figure 5.4. Une différence notable est le comportement de tous les algorithmes d'ordonnancement (même la limite supérieure) où l'on observe une perte de performance notable lorsque le nombre de groupes de cœurs est faible. Ce comportement s'explique par le fait que certains des noyaux composant la factorisation QR ne peuvent pas être parallélisés aussi efficacement que les noyaux de la factorisation de Cholesky dans leur version actuelle. On observe cependant que Heft a un comportement légèrement différent, en effet pour une petite taille de matrice utiliser peu de groupes de cœurs de grande taille fourni les meilleures performances, de plus même lorsque la taille de la matrice est grande, Heft préfère un nombre intermédiaire de groupes de cœurs, notamment 12 et 14 groupes de cœurs plutôt que l'utilisation des 20 cœurs de façon indépendante. Pour `HP avec Aire` et `HP avec Index Het.` contrairement à Cholesky, très peu de configurations fournissent de meilleurs performances que la version de référence.

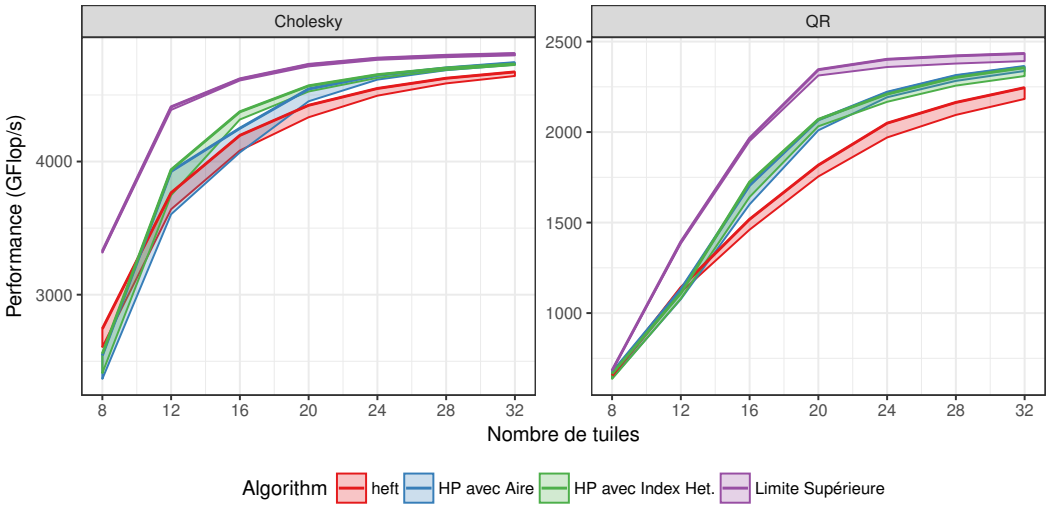


FIGURE 5.5 – Performance des 10% meilleures configurations pour les deux opérations

La Figure 5.5 présente une autre vue des mêmes résultats où seulement les 10% meilleures configurations pour chaque taille de matrice et chaque algorithme sont représentées. Cette figure montre les performances obtenues sur ces configurations avec un ruban pour chaque algorithme où le plus haut point représente la meilleure configuration et le plus bas la pire des 10% choisies. Ces résultats la performance typique que l'on peut obtenir avec chaque algorithme si la configuration peut être adaptée à l'algorithme. Pour Cholesky, on observe clairement que l'écart entre Heft et `HP avec Aire` et `HP avec Index Het.` est plus grand pour les matrices de tailles moyennes alors que pour QR l'écart est toujours présent pour les matrices de tailles larges. Les variantes

`HP avec Aire` et `HP avec Index Het.` de HeteroPrio ont un comportement similaire pour le meilleur cas, sauf pour la factorisation de Cholesky avec des matrices de tailles moyennes où `HP avec Index Het.` est meilleur que `HP avec Aire`. Enfin, les résultats obtenus par HeteroPrio sont proches de la Limite Supérieure dans tous les cas considérés.

Pour résumer, les variantes de HeteroPrio obtiennent dans ces simulations de meilleures performances que Heft dans la plupart des cas et l'utilisation du score `Indice H.` est préférable vu la stabilité observée de cette version. C'est donc cette version que l'on utilise dans les résultats suivants. Ces résultats mettent aussi en avant les avantages de l'utilisation de tâches parallèles : Heft obtient systématiquement de meilleures performances avec certaines configurations en utilisant des groupes de cœurs, et HeteroPrio a le même comportement avec la factorisation de Cholesky. Lorsque l'on considère la factorisation QR, la scalabilité limité de plusieurs noyaux rend l'utilisation de groupes de cœurs très peu bénéfique pour les variantes de HeteroPrio.

Analyses de l'utilisation réelle de l'ordonnancement de HeteroPrio calculé hors-ligne

Les résultats suivants sont obtenus à partir d'exécutions réelles de HeteroPrio en utilisant le support d'exécution StarPU. Les configurations retenues sont celles pour lesquelles HeteroPrio obtient les meilleures performances en simulation. Pour une configuration donnée, on construit les groupes de cœurs en se concentrant sur la localité des ressources comme présenté en Section 3.2.4 et on s'assure que ceux-ci se traversent pas les frontières NUMA présentent sur la machine. On obtient à partir d'une exécution hors-ligne de HeteroPrio une allocation des tâches sur les ressources et un ordre d'exécution des tâches sur chaque ressources que l'on utilise pour l'exécution réelle. Pour ce faire, on créé un simple ordonnanceur en-ligne dont le principe est de respecter l'ordonnancement créé par HeteroPrio.

La Figure 5.6 montre une trace d'exécution lorsque exécute l'ordonnancement HeteroPrio calculé hors ligne sans aucune modification. On remarque que l'ordonnancement calculé par HeteroPrio est plutôt bon et permet de bien occuper la machine (on obtient environ 3800 GFlop/s), cependant un peu d'inactivité est apparu. Ainsi, il est nécessaire de rendre l'ordonnancement calculé par HeteroPrio plus souple pour pouvoir s'adapter aux changements sur la machine.

Le problème principal qui explique la perte de performances observé dans la figure précédente s'explique par le fait que les communications sont négligées dans la simulation. Deux fonctionnalités supplémentaires sont ainsi implantées afin d'adapter dynamiquement l'ordonnancement aux variations environnementales de la machine. En premier lieu, dès qu'une ressource sur CPU manque de travail (car aucune tâches n'a encore été assignée à sa file de tâche

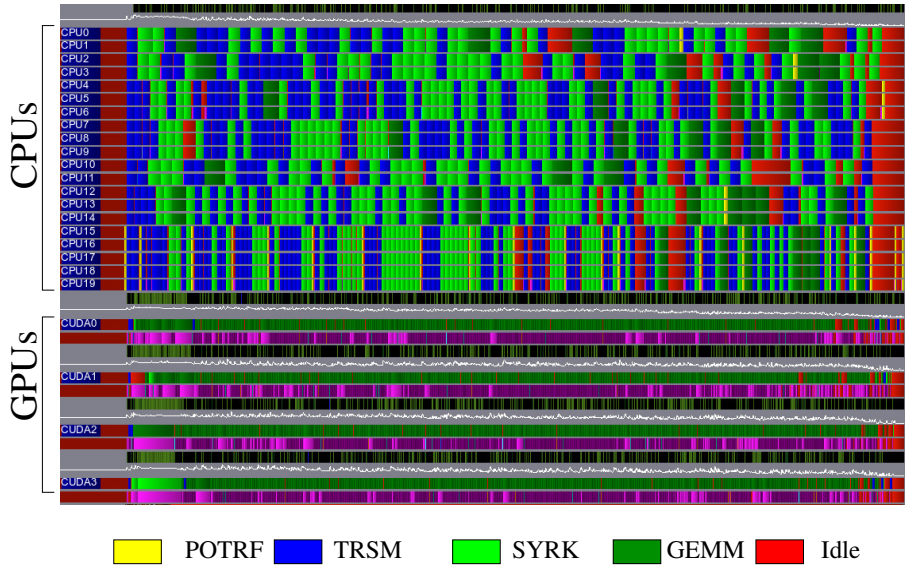


FIGURE 5.6 – Trace d'exécution de la factorisation de Cholesky pour une matrice de taille 24 × 24 tuiles avec l'ordonnancement de HETEROPRIO. L'abscisse représente le temps, l'ordonnée représente les ressources avec les GPUs en bas.

prêtes) on lui autorise de voler une tâche préférablement à un groupe de cœurs ayant le même nombre de ressources ; pour ce faire on calcule une distance entre les ressources selon leur nombre de ressources que l'on prend en compte pour le vol de travail. En second lieu, toutes les tâches attribuées sur GPU sont mises dans une file de tâches fusionnée pour tous les GPUs, depuis laquelle les tâches sont assignées, dans l'ordre, au GPU qui la fini le plus rapidement. Ceci permet de fortement réduire le nombre et le volume total de communications entre les GPUs qui apparaît car HETEROPRIO ne prend pas en compte les communications.

La Figure 5.7 montre les performances réelles obtenues pour une matrice de taille 24 × 24 tuiles avec les deux fonctionnalités présentées précédemment implantées. Ceci montre que la plupart des tâches DGEMM sont exécutées sur GPU et les communications sont majoritairement recouvertes par du calcul. On remarque cependant que les tâches exécutées sur les CPUs (particulièrement les tâches DPOTRFDTRSM), un petit temps d'inactivité est introduit à cause des transferts (en violet ici), ce qui cumulé créé un déséquilibre de charge et de l'inactivité sur les GPUs à la fin du calcul. Pour prendre en compte ce nouveau problème surcoût induit par une non-prévision de ces communications, on hausse le temps d'exécution des tâches sur CPUs pour le calcul hors-ligne de l'ordonnancement de HETEROPRIO. On a observé empiriquement qu'une augmentation de 15% du temps des tâches permet d'obtenir le meilleur équi-

libre de charge entre CPUs et GPUs pour toutes les tailles de matrices et configurations de groupes de cœurs.

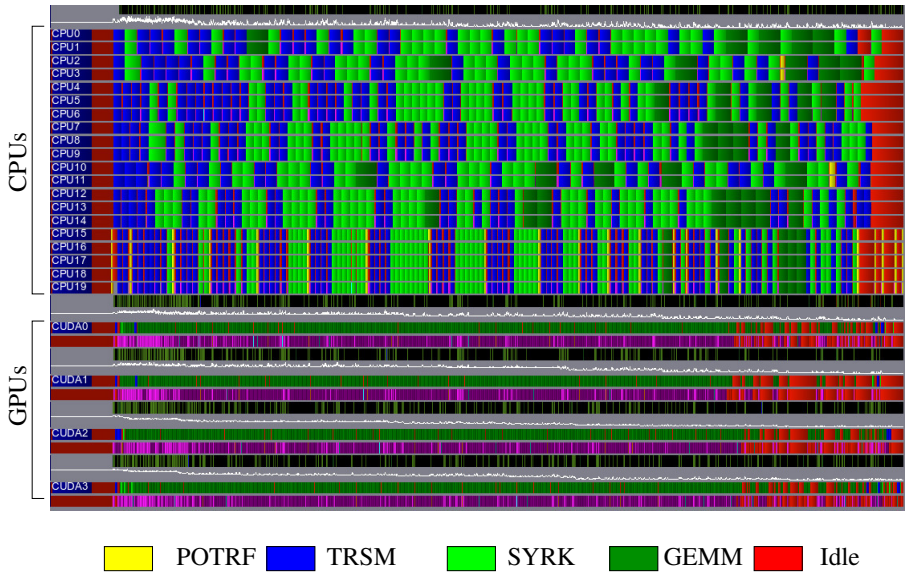


FIGURE 5.7 – Trace d'exécution de la factorisation de Cholesky pour une matrice de taille 24 × 24 tuiles avec l'ordonnancement de HETEROPRIO avec deux améliorations à l'ordonnancement afin de s'adapter au coût des communications.

La Figure 5.8 représente une trace d'exécution de l'ordonnancement HETEROPRIO corrigé avec la prise en compte des 15% de temps supplémentaire sur CPU. On observe que l'équilibre de charge est fortement amélioré : les GPUs et les cœurs des CPUs sont utilisés jusqu'à la fin de l'exécution. Dans la suite pour la comparaison de performances réelles, on garde ces réglages pour HETEROPRIO.

Comparaison des performances réelles de HETEROPRIO et HEFT

On compare les performances de la factorisation de Cholesky à travers de vraies exécutions pour différentes tailles de matrices des algorithmes d'ordonnancement HEFT et HETEROPRIO avec MAGMA [7], une bibliothèque de référence d'algèbre linéaire dense. Pour rappel, l'exécution HETEROPRIO réelle (hp-best dans la Figure 5.9) provient de l'exécution de l'ordonnancement obtenu en mode simulation sur la meilleure configuration de groupes de cœurs avec les deux assouplissements et la correction de 15% décrites précédemment. Pour HEFT (heft-best dans la Figure 5.9) on utilise la configuration de groupes de cœurs qui obtient la meilleure performance en simulation avec cette stratégie. On exécute aussi HEFT avec la meilleure configuration obtenue par l'ordon-

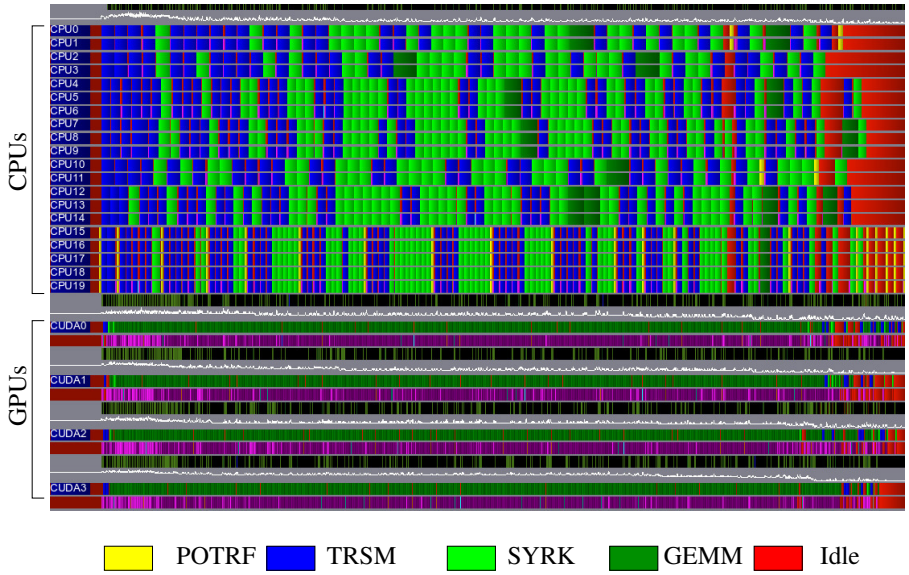


FIGURE 5.8 – Trace d'exécution de la factorisation de Cholesky avec un meilleur équilibre de charge entre CPUs et GPUs pour une matrice de taille 24×24 tuiles avec l'ordonnancement de HETEROPRIO.

nanceur HETEROPRIO, que l'on nomme heft (hp-best config.). On représente aussi la performance obtenue par HEFT lorsque l'on n'utilise pas de groupes de cœurs (autrement dit les 20 cœurs sont utilisés de façon indépendante) nommée heft-wc comme performance de référence ainsi que la performance obtenue par `MAGMA`. Pour chaque résultat, on montre la moyenne de 10 exécutions ainsi que l'écart type. Les meilleures configurations varient selon la taille de la matrice. On observe par exemple pour une matrice de taille 24×24 tuiles avec HETEROPRIO, la meilleure configuration est $(2, 3, 5) - (2, 2, 3, 3)$ et avec HEFT $(4, 6) - (3, 3, 4)$. On remarque que la granularité des ressources pour HEFT est plus grande que celle de HETEROPRIO et l'on a deux ressources de plus. Pour HEFT avec une taille de matrice de 32×32 tuiles, la meilleure configuration est $(2, 3, 5) - (2, 2, 2, 4)$ et pour HETEROPRIO $(1, 2, 3, 4) - (1, 1, 2, 2, 4)$. On remarque donc que dans les deux cas plus de ressources sont créées par rapport à leur meilleure configuration avec une matrice de taille 24×24 tuiles.

Pour une matrice de taille 12×12 tuiles, on observe que heft-wc et MAGMA obtiennent une performance similaire. Lorsque l'on utilise des groupes de cœurs, la performance de heft-best augmente de 58% par rapport à heft-wc. Ceci est attendu car la quantité de parallélisme généré par cette taille de matrice n'est pas suffisante pour remplir les 20 CPUs pour heft-wc et résulte en de mauvaises performances. hp-best obtient une meilleure performance de 6% par rapport à heft (hp-best config.) avec la même configuration. heft-best obtient des performances légèrement meilleures que heft (hp-best config.) ce qui peut être

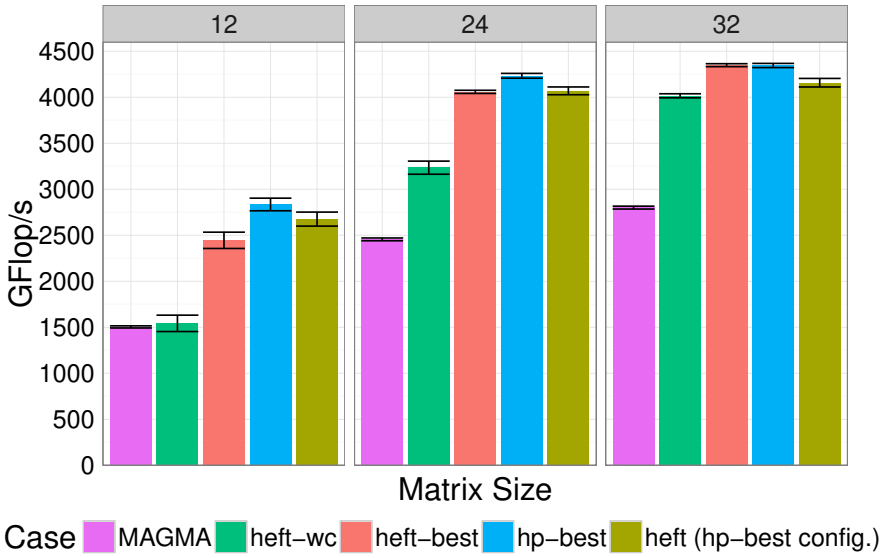


FIGURE 5.9 – Résultats de performance pour les politiques d'ordonnancement HEFT et HETEROPRIO avec certaines configurations de groupes de cœurs.

expliqué par les communications. En effet, lorsque l'on a un faible nombre de tâches comme c'est le cas ici, il est possible que l'on manque de tâches pour complètement recouvrir les communications et ainsi la configuration identifiée par la simulation comme étant la meilleure peut, sur une vraie exécution, souffrir de surcoûts à cause de transferts non recouverts par du calcul.

Pour des tailles de matrice plus large, on observe que l'écart de performance entre heft-wc et hp-best se réduit à 31% (pour 24×24) et 8% (pour 32×32). De plus, la simulation est plus précise et donc la performance de heft-best surpasse heft (hp-best config.). hp-best obtient une amélioration de performance de 4.5% par rapport à heft-best pour le cas 24×24. Cependant, *hp − best* n'obtient pas d'améliorations significative pour le cas 32×32 à cause de l'évolution de l'allocation des tâches lorsque l'on augmente la taille de la matrice. En effet, pour des tailles de matrices plus grandes, l'exécution est dominée par l'exécution de tâches DGEMM (presque indépendantes), ce qui rend le problème d'ordonnancement plutôt facile aussi bien pour hp-best que heft-best qui obtiennent alors une performance similaire. Ces résultats sont cohérents avec la Figure 5.5 qui montre que la différence entre HETEROPRIO et HEFT est bien plus faible pour le cas 32×32 comparé aux matrices de tailles plus faibles (et ils sont en plus tous les deux très proches de la borne supérieure).

5.1.4 Discussion

On a réalisé plusieurs extensions à l'algorithme d'ordonnancement HeteroPrio dans le cas où l'on utilise plus de deux types de ressources, ce qui apparaît dès que l'on utilise des tâches parallèles. On a observé que l'utilisation de ce type d'ordonnanceurs est possible lorsque l'on utilise des tâches parallèles grâce au score "indice d'hétérogénéïté". On a aussi réalisé une étude exhaustive de toutes les configurations possibles de groupes de cœurs sur deux applications, les factorisations de Cholesky et de QR. On a aussi réalisé des comparaisons de performance réelles entre les meilleurs configurations obtenues par Heft et HeteroPrio. L'intérêt d'utiliser des groupes de cœurs hétérogènes avec l'utilisation de tâches parallèles rigides a aussi été démontré expérimentalement.

Cette étude de performances permet d'observer que l'utilisation de tâches parallèles permet toujours d'obtenir de meilleurs performances sur ces deux applications pour l'algorithme d'ordonnancement Heft et pour HeteroPrio avec la factorisation de Cholesky. L'utilisation de tâches parallèles permet donc d'améliorer la qualité des ordonnancement en réduisant l'hétérogénéïté entre les CPUs et GPUs dans la machine. On observe aussi qu'une recherche exhaustive en simulation permet de trouver des configurations de groupes de cœurs non triviales, *i.e.* qui ne sont pas simplement 2×10 qui obtiennent de meilleures performances que celle-ci. Les résultats en simulation manquent cependant de précision car le coût des communications n'est pas pris en compte. Malgré cela, notamment pour les matrices de grandes tailles intermédiaire à grande les résultats observés par la simulation sont confirmés.

Plusieurs points d'améliorations peuvent être identifiés. Une technique intéressante mais non présentée ici pour implanter cet ordonnanceur HeteroPrio amélioré en pratique est d'utiliser un système de spéculation grâce à l'utilisation de simulation d'exécution sur les cœurs inactifs de la machine avant de prendre une décision pouvant générer ce type d'erreur, *i.e.* d'exécuter une tâche avec un gros facteur d'accélération sur CPU alors que l'on risque de manquer de tâches. Une alternative à une implantation réelle de HeteroPrio est de permettre à StarPU de fournir, de façon générique, à un outil externe le graphe de tâche considéré ainsi que les informations nécessaire pour que cet outil calcul une ordonnancement et que celui-ci soit respecté en exécution réelle. Il serait aussi intéressant d'explorer la faisabilité d'identification des meilleures configurations de groupes de cœurs grâce à une heuristique, notamment selon les modèles de performance des tâches et la largeur du graphe de tâches soumis. Enfin, dans le cadre d'un ordonnancement de tâches parallèles capable de changer dynamiquement les configurations, il serait intéressant d'observer si des configurations exotiques apparaissent naturellement et obtiennent d'aussi bonnes performances.

5.2 Une plateforme de reconfiguration et de contrôle dynamique de ressources pour l'ordonnancement de tâches parallèles

Dans la suite de ce chapitre, on propose une plateforme de reconfiguration et de contrôle dynamique des ressources afin de faciliter la création d'ordonnanceurs de tâches parallèles. Afin d'évaluer ces interfaces, deux implantations d'ordonnanceurs différents sont proposées en section 5.3.

5.2.1 Introduction et besoins principaux

L'objet de cette plateforme est de fournir un ensemble d'abstractions et de mécanismes simplifiant la création d'ordonnanceurs capables d'adapter à la volée la machine à une exécution, c'est à dire d'adapter régulièrement la configuration des groupes de cœurs au flot de tâches parallèles à traiter et d'assigner les tâches à ces groupes de cœurs. Il s'agit aussi de donner au programmeur expert un moyen de piloter efficacement les changements de configuration, et ce directement depuis l'application, afin que celui-ci puisse expérimenter différentes stratégies en transmettant des indications plus ou moins contraignantes à de tels ordonnanceurs.

De plus le cachier des charges d'une telle pateforme doit tenir compte des contraintes techniques imposées par StarPU (contraintes de faisabilité et de maintenabilité), de celles nécessaires à la composition de supports d'exécution (mécanismes d'endormissement et réveil de ressources) ainsi que celles issues de la théorie de l'ordonnancement des tâches moldables. Pour ce dernier point, et d'après l'état de l'art (*cf.* 2.1.3), les ordonnanceurs de tâches moldables fonctionnent le plus souvent en deux phases : la première phase, dîte d'allotement, attribue la quantité nécessaire de ressources, la seconde réalise l'allocation sur les ressources. Il est donc intéressant d'introduire des mécanismes facilitant l'implémentation de ce type d'ordonnanceur, comme la création d'un sac pertinent de tâches prêtes.

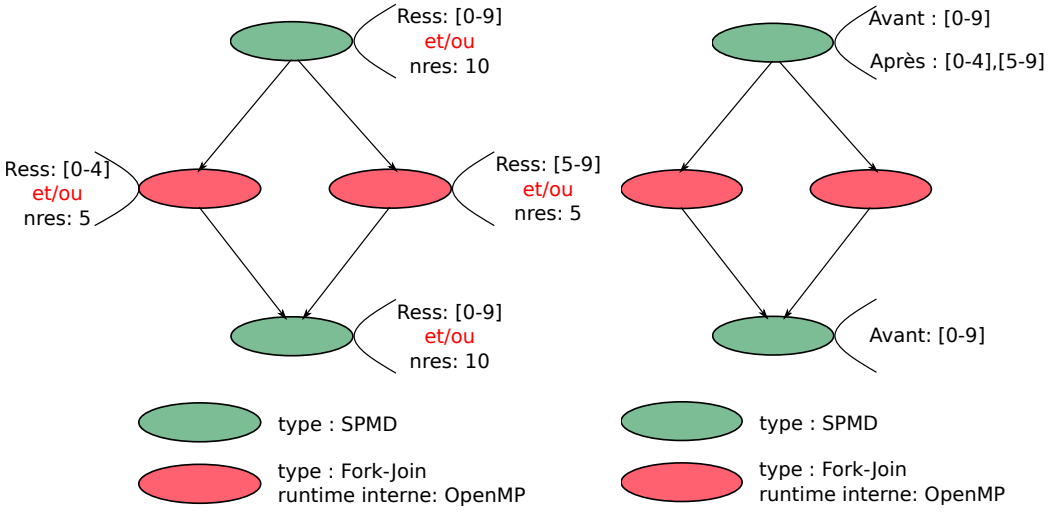
Il est également nécessaire d'obtenir des modèles de performances tenant compte du nombre de ressources attribuées à un noyau de calcul. Afin de faciliter la recherche de performance et de simplifier l'étape de calibration, qui pourrait être très coûteuse sur un KNL avec 72 cœurs, il n'est pas acceptable d'essayer toutes les combinaisons de nombre de cœurs avant de commencer l'ordonnancement de tâches parallèles.

Enfin les aspects génie logiciel incite à limiter au mieux l'impact des modifications à apporter à StarPU. L'implantation doit idéalement se cantonner au niveau des ordonnanceurs puisqu'il s'agit d'introduire de nouveaux mécanismes liés à l'ordonnancement. En particulier, il n'est notamment pas question de modifier le concept même de ressources afin d'utiliser des tâches parallèles,

il suffit de manipuler une nouvelle *vision* des ressources de calcul.

5.2.2 Modèle de programmation

Le modèle de programmation pour piloter la plateforme d'exécution de tâches parallèles proposées se base sur une annotation des tâches avec l'ajout d'informations soit par un utilisateur lors de la soumission des tâches, soit par le programmeur d'ordonnanceurs. La Figure 5.10 montre les deux types d'annotations proposées, chacune avec un comportement différent. Dans les deux cas, les codelets des tâches, dont une tâche n'est qu'une instance particulière, possèdent plusieurs informations dont la plus essentielle est le type de parallélisme, soit séquentiel, soit SPMD, soit Fork-Join. Il est possible de transmettre d'autres informations sur les granularités de ressources minimales et maximales souhaitées pour ce type de tâches afin de faciliter le travail du modèle de performance. Enfin, il est aussi nécessaire de transmettre la fonction d'interaction avec le support d'exécution interne. Plusieurs exemples sont fournis, notamment pour réaliser une interaction avec OpenMP.



(a) Association de nombre ou liste de ressources aux tâches.

(b) Partition de la machine liées à certaines tâches.

FIGURE 5.10 – Présentation de quelques points d'interface pour contrôler les ressources.

Dans la Figure 5.10a l'annotation se fait par tâche et consiste à spécifier une liste de ressources attribuée. Un ordonnanceur peut ainsi choisir comme dans cet exemple s'il est sur une machine à 10 ressources, d'attribuer les 10 ressources à cette tâche, puis d'attribuer aux deux tâches suivantes les ressources respectives $[0-4]$ et $[5-9]$. Il est aussi possible de fournir une information intermédiaire qui concerne uniquement la quantité de ressources souhaitées et

non pas une liste de ressources, dans un tel cas l'ordonnanceur doit se charger de respecter ces demandes et d'attribuer, par la suite, une liste de ressources aux tâches respectant les demandes reçues. Cette demande de ressource peut être réalisé soit pas une première passe d'ordonnanceur qui fonctionne en deux temps comme les ordonnanceurs de l'état de l'art, ou bien par un utilisateur expert qui souhaite faciliter le travail de l'ordonnanceur grâce à des connaissances algorithmiques.

La Figure 5.10b montre une seconde façon plus basique d'interagir avec la plateforme d'ordonnancement de tâches parallèles. Ce mode repose aussi sur une annotation de tâches mais elle ne se réalise pas avec le même objectif. Ici, l'annotation décrit une partition de la machine. En prenant le même exemple que la Figure 5.10a, on créé un unique groupe de cœurs sur les ressources $[0-9]$ avant d'exécuter la première tâche, et une fois cette tâche finie on se charge de partitionner la machine en créant deux groupes de cœurs des ressources $[0-4]$ et $[5-9]$. Contrairement à précédemment, ces partitions ne sont pas directement attribuées à la tâche en question, ce sont juste des ordres de partition de la machine. Les ordonnanceurs peuvent alors simplement utiliser ces partitions directement pour ordonnancer les tâches au dessus de celles-ci. Ces partitions peuvent être créées par un utilisateur qui souhaite adapter à certains points de son algorithme la configuration de la machine en suivant son chemin critique par exemple. Si l'on considère une factorisation de Cholesky il peut être avantageux de modifier à chaque tâche DPOTRF les groupes de cœurs de la machine selon le parallélisme libéré par ce DPOTRF en question. Lorsque l'utilisateur utilise ce mode de gestion de ressources, il est toujours possible d'annoter en plus les tâches de façon précise mais ce mode d'utilisation peut être complexe à réaliser correctement donc il est recommandé d'utiliser soit le mode de la Figure 5.10a soit celui de la Figure 5.10b sans les mélanger.

Choix sur la forme de l'implantation dans StarPU

L'implantation de cette plateforme de tâches parallèles doit se réaliser principalement au niveau des ordonnanceurs et des tâches. Il ne faut pas modifier le modèle de ressource d'origine, mais pouvoir implanter les modifications nécessaires simplement et sans modifier le cœur des supports d'exécution, notamment leur gestion des ressources. Un second objectif est que le contrôle des ressources et la création des groupes de cœurs se réalisent à base de l'annotation des tâches par liste des ressources et que tout soit automatique et complètement transparent à partir de cette information.

Pour ce faire, il est proposé ici de réaliser un petit module d'ordonnancement qui se branche principalement au niveau des ressources en s'inspirant des ordonnanceurs modulaires (*cf.* 2.2.2). Bien que le travail a été principalement réalisé pour les ordonnanceurs modulaires, il est tout à fait possible de

l'utiliser dans le cas d'ordonnanceurs classiques comme on observera dans la Section 5.2.3.

Un dernier choix d'implantation est de ne pas utiliser les contextes d'ordonnancement. Plusieurs points justifient ce choix. En premier, les contextes servent à isoler des ordonnanceurs différents sur différentes ressources, alors qu'ici on souhaite proposer un outil aux ordonnanceurs pour le contrôle des ressources. De plus, on peut avoir autant de contextes que l'on souhaite simultanément alors qu'ici la vision est plus simple : une ressource ne peut appartenir qu'à un seul groupe de cœurs en même temps. Aussi, les contextes s'inscrivent au cœur de StarPU et s'il faut les adapter à nouveau pour cette utilisation cela implique de nouveaux changements de fond dans StarPU, d'autant plus que on ne s'intéresse pas réellement à modifier les ressources directement mais uniquement une vue spécifique aux groupes de cœurs. Enfin, un dernier problème est que l'on peut vouloir combiner l'utilisation de contextes traditionnels avec l'utilisation de groupes de cœurs pour isoler deux flux d'applications parallèles différents et ceci ne peut pas se faire en réutilisant les contextes pour représenter les groupes de cœurs.

Utilisation de tâches de contrôle des ressources

L'idée principale centrale dans cette implantation de la plateforme d'ordonnancement de tâches parallèles est l'utilisation de tâches pour le contrôle de l'état des ressources. En effet, le concept de tâches est l'idée centrale dans StarPU, et celui-ci a été utilisé à plusieurs reprises notamment pour réaliser du contrôle sur la forme ou la représentation des données dans le graphe de tâches. Ce concept est bien approprié pour le contrôle de ressources de calcul, avec quelques changements nécessaires. La Figure 5.11 (*resp.* 5.12) illustre comment les tâches de contrôle sont utilisées dans le cas de tâches parallèles Fork-Join (*resp.* SPMD). Afin de contrôler l'état des ressources, des tâches compagnons pour faire dormir des ressources et en réveiller d'autres sont ajoutées dans la file de tâche des workers StarPU en même temps que la tâche de calcul.

Dans la Figure 5.11 l'ajout de ces tâches de contrôle de ressources de calcul est illustré sur un exemple précis. Dans cet exemple, on pousse sur les workers (0 à 3) deux tâches parallèles (représentées en haut à gauche). La première tâche, rouge, souhaite s'exécuter sur les ressources $[0, 1]$ puis après un certain temps la tâche violette arrive et souhaite s'exécuter sur la ressource $[0]$. Dans la situation de départ, tous les workers sont indépendants et réveillés, ce qui est la configuration standard de StarPU. Afin de respecter les demandes des tâches parallèles et de s'assurer qu'au moment d'exécuter les tâches parallèles les ressources seront dans les bonnes configurations successives il suffit d'ajouter deux tâches, la verte et la grise. La tâche de sommeil (la grise), sert à endormir la ressource 1 sur un sémaphore avant d'exécuter la tâche rouge. De fait, avant d'exécuter la tâche rouge la ressource 0 qui est choisie comme maître pour cette

tâche Fork-Join attend d'être notifié (grâce à un sémaphore) que la ressource 1 est correctement endormie. Par convention, la ressource maître est toujours la ressource avec le plus petit indice. Ensuite, avant de placer la tâche violette dans la file de tâches de la ressource de calcul 0, une nouvelle tâche compagnon, de réveil, (la verte), est ajoutée dans la file de tâches de cette même ressource. Cette tâche sert à réveiller la ressource 1 qui est toujours endormie sur son sémaphore. En effet, on se rappelle que jusqu'à présent les ressource $[0, 1]$ forment un groupe et la ressource 1 est sous le contrôle de la ressource 0, ainsi afin de rétablir la machine dans un état cohérent la ressource 0 doit s'assurer de réveiller la ressource 1 avant de casser le groupe de cœurs.

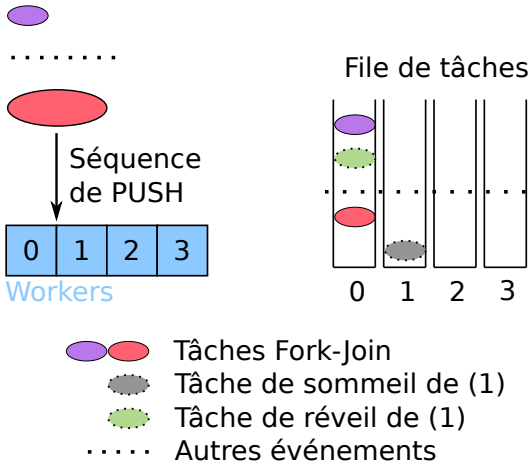


FIGURE 5.11 – Ajout de tâches compagnons afin de contrôler les ressources en mode Fork-Join.

Ce fonctionnement permet d'adapter de façon entièrement dynamique et avec le minimum de synchronisation requises par les demandes en ressources de calcul des tâches parallèles l'état des ressources et les groupes de cœurs. Puisque ces demandes sont réalisées par les ordonnanceurs, il est nécessaire d'utiliser ces tâches de contrôle comme des tâches compagnons de façon systématique lorsque l'on pousse les tâches sur les ressources puisque à ce moment là, les tâches sont vues comme des tâches indépendantes et il n'est pas possible d'ajouter de dépendances explicites entre les tâches pour séquentialiser les tâches parallèles dont les demandes de ressources sont en conflit.

La Figure 5.12 montre l'utilisation de tâches compagnons ajoutées dans les files de tâches des ressources de calcul lorsque l'on utilise des tâches parallèles de type SPMD, où toutes les ressources doivent être réveillées. Ici, les ressources sont dans la configuration $[0-1], [2], [3]$, le groupe de cœurs $[0-1]$ est de type Fork-Join donc la ressource 1 est endormie. Dans cette configuration, le module d'adaptation des tâches parallèles reconnaît l'existence de ce groupe et poste une tâche de réveil sur la ressource 0 pour réveiller la res-

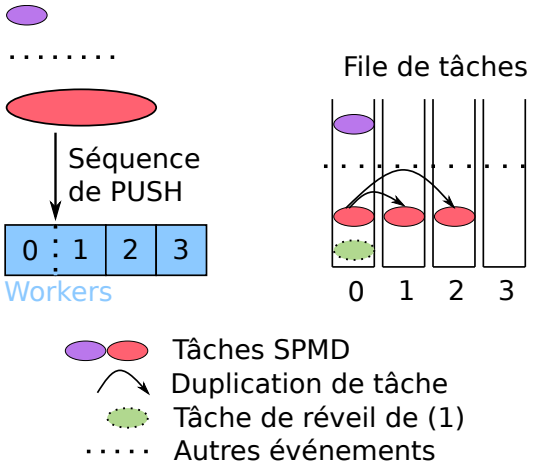


FIGURE 5.12 – Ajout de tâches compagnons afin de contrôler les ressources en mode SPMD.

source 1 avant l'exécution de la tâche SPMD rouge. De plus, la tâche rouge est automatiquement dupliquée avec un système d'alias et se retrouve poussée sur chacune des ressources $[0-2]$ de façon transparente pendant l'adaptation de l'état des ressources. Pour s'assurer que toutes les images des tâches sont exécutées par toutes les ressources de façon synchronisée, les barrières et les numéros de rangs présentés en Section 3.2.2 sont utilisés. Par la suite, quand la tâche violette doit être exécutée, aucun changement d'état des ressources n'est nécessaire puisqu'elles restent réveillées : il suffit de noter correctement les nouvelles configurations de groupes de cœurs.

5.2.3 Conception du contrôle des ressources à l'ordonnancement

De façon interne, pour réaliser ce contrôle de ressources plusieurs informations sont nécessaires. Les listes de ressources sont représentées par des tableaux de bits ("bitmaps") et fournies par les tâches parallèles. Il est aussi ajouté aux tâches parallèles lors de l'adaptation une information sur le nombre notification de sommeil de ressources que la ressource maître (dans le cas Fork-Join) doit recevoir avant l'exécution de cette tâche. Un second type d'information sert de mémoire de l'état des groupes grâce un pointeur de groupe de cœurs attribué à chaque ressources. Cette structure, qui représente un groupe de cœurs, sont les méta-données de la dernière tâche qui sera exécutée par un certain groupe de ressources. Cette structure sert à reconnaître le dernier état cohérent des ressources et lorsqu'une nouvelle tâche est poussée, procéder à l'adaptation des groupes de ressources avec succès. Pour cela, cette structure, qui représente un groupe de cœurs, contient un tableau de bits et un pointeur

vers le dernier codelet exécuté, principalement afin de savoir si l'on est un mode Fork-Join ou SPMD.

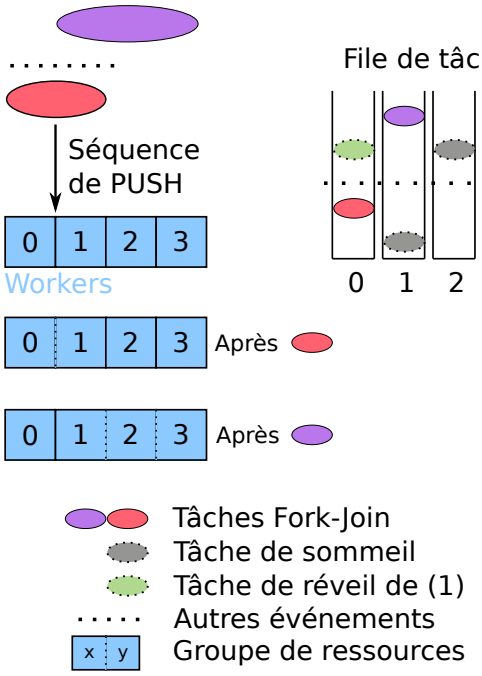


FIGURE 5.13 – Définition de groupe de ressources à partir des requêtes de changement de configurations liés aux tâches parallèles.

La Figure 5.13 illustre l'adaptation des ressources successivement et la création des groupes à travers un exemple. Ici, Une première tâche rouge de type Fork-Join est poussée sur les ressources $[0-1]$ et on considère les ressources dans leur état d'origine isolées. Après cette avoir ajouté cette tâche à la file de tâche du worker 0 et adapté l'état des ressources, les ressources 0 et 1 font parti d'un nouveau groupe de ressources $[0-1]$ de type Fork-Join. En se basant sur cette information, lorsque la tâche violette est poussée sur les ressources $[1-3]$, il est possible de reconnaître aisément que les ressources $[0-1]$ sont dans un groupe partagé et que 1 sera endormie à l'exécution de la tâche violette, ainsi il est nécessaire de la réveiller puisque c'est la ressource maître du futur groupe $[1-3]$. Une fois fait, il suffit de poster les tâches de sommeil pour les ressources 2 et 3. La ressource 0 forme alors un groupe de cœurs seule. Après ces modifications, si une nouvelle tâche parallèle est poussée il est aisé de procéder aux adaptations nécessaires car on considère alors la machine dans l'état $[0], [1-3]$ grâce aux méta-données des dernières tâches à exécuter et les groupes de cœurs.

La Figure 5.14 illustre avec le même exemple l'identification des ressources qu'il faut verrouiller pour procéder aux adaptations d'état des ressources, ce qui est un problème majeur de cette technique. En effet si l'on considère la

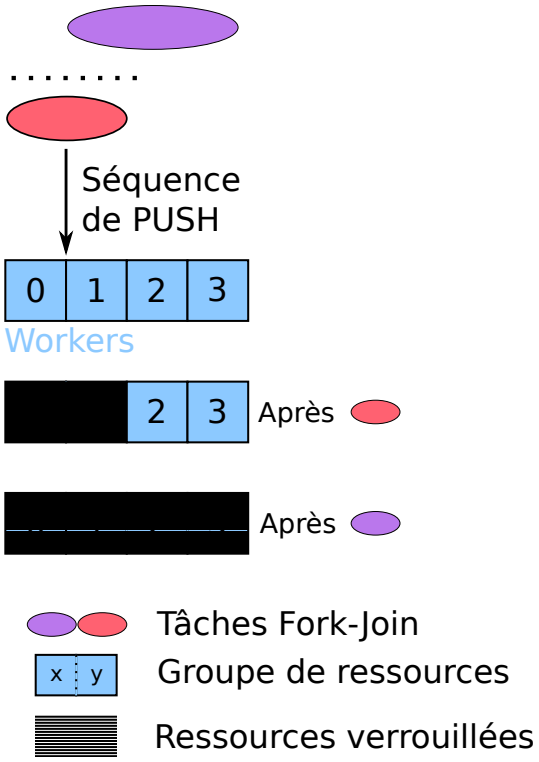
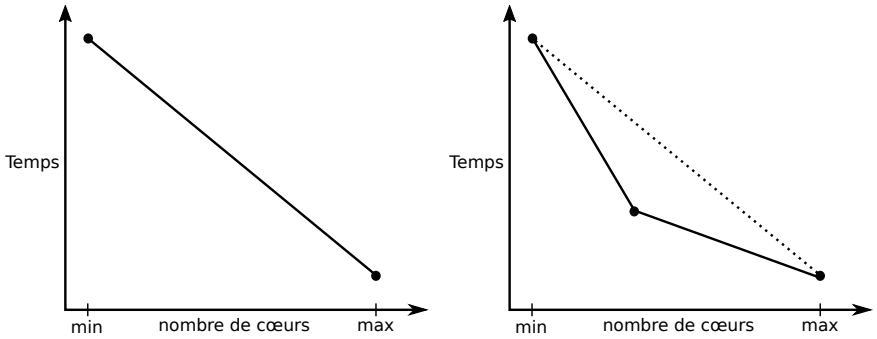


FIGURE 5.14 – Verrouillage des groupes de ressources pour la cohérence des changements de configurations.

tâche rouge, puisque les ressources sont isolées, il suffit de verrouiller les ressources 0 et 1 pour créer le groupe de cœurs $[0, 1]$ et endormir la ressource 1. Ensuite, quand la tâche violette est poussée, puisque la machine est dans une configuration $[0-1], [2], [3]$ et que la tâche violette émet une requête pour les ressources $[1-3]$, il est nécessaire de verrouiller les ressources 2 et 3 mais aussi les ressources du groupe $[0-1]$ car 0 doit réveiller 1. Afin de s'assurer de la cohérence de la machine lorsque l'on procède à une modification de l'état des ressources, il est nécessaire de verrouiller tous les groupes de cœurs existants dont au moins une ressource est impliquée dans la création du nouveau groupe de cœurs. Pour éviter les interblocages, un mutex est utilisé pour sérialiser les requêtes et identifier grâce à quelques opérations sur les tableaux de bits toutes les ressources à verrouiller.

5.2.4 Estimations de la performance des tâches parallèles

Un problème mis en avant précédemment concerne l'estimation de la durée d'une tâche parallèle lorsque l'on ne souhaite pas calibrer les modèles de performances pour chaque tâche avec toutes les possibilités de ressources avant de pouvoir commencer à l'ordonnancer. La Figure 5.15 illustre une proposition de



(a) Avec deux points. (b) Avec trois points.

FIGURE 5.15 – Exemple fictif d'un modèle simple de droites entre deux points pour estimer les performances des noyaux avec des configurations non calibrées.

modèle simple pour régler ce problème : un modèle linéaire par morceau. Afin d'utiliser ce modèle, il est nécessaire de connaître la performance des noyaux parallèles avec le nombre minimum et le nombre maximum de ressources disponibles. Ces granularités de ressources peuvent être spécifiées par l'utilisateur dans les structures de codelet des tâches. Le modèle choisi consiste alors à simplement calculer la pente entre les deux points et utiliser cette performance comme modèle. Ceci est montré dans la Figure 5.15a. Ce modèle peut fournir une estimation inexacte la première fois, cependant le modèle gagne significativement en qualité grâce à l'ajout de nouveaux points comme montré dans la Figure 5.15b. Dans l'exemple de la Figure 5.15, un nouveau point a été calibré et est disponible pour l'estimation de performance. Lorsque l'on souhaite alors connaître le temps d'exécution d'une tâche sur un nombre de ressources calibrées, on considère la pente sur laquelle ce point se trouve pour fournir la performance estimée. Afin d'utiliser ce modèle, des fonctions sont fournies non seulement pour obtenir la performance avec un certain nombre de cœurs mais aussi le nombre de cœurs nécessaires pour obtenir une certaine performance.

Les avantages de l'utilisation de ce type de modèle est qu'il est simple à mettre en œuvre et s'adapte à la présence de nouveaux points. Par la suite, un modèle de régression linéaire multiparamétrique (prenant en compte la taille des noyaux et le nombre de processus légers) pourrait être utilisé.

5.3 Ordonnanceurs dynamiques de tâches moldables

Dans cette section, deux ordonnanceurs sont proposés en utilisant la plateforme de création d'ordonnanceurs et d'exécution de tâches parallèles proposé dans la section précédente. Le premier ordonnanceur est un ordonnanceur dit

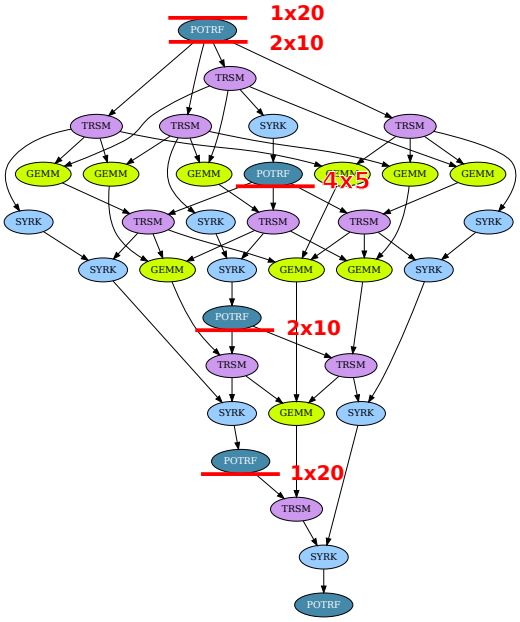
"naïf" afin de présenter une première approche de l'ordonnancement pratique de tâches parallèles notamment en se reposant sur plusieurs connaissances sur la factorisation de Cholesky, pour laquelle cet ordonnanceur est créé mais qui pourrait être adapté à d'autres applications à base de tâches. Le second ordonnanceur est une adaptation directe d'un algorithme proposé dans l'article [25] et déjà présenté succinctement en Section 2.1.3.

5.3.1 Conception d'un ordonnanceur dit «naïf»

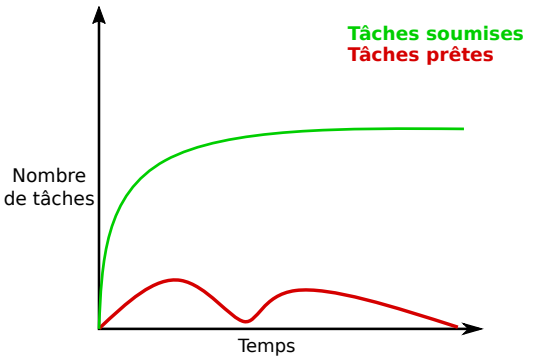
L'objectif de ce premier ordonnanceur est de réaliser une évolution de la taille des ressources de façon dynamique au cours du temps selon les tâches d'un chemin critique identifié par l'utilisateur. Le résultat attendu est montré dans la Figure 5.16a. Dans cette factorisation de Cholesky ont été ajoutés des points de changements de la configuration globale de la machine à chaque tâche DPOTRF. Il est pertinent de s'adapter à la largeur du graphe de tâches : dès que celui-ci dispose de trop peu de tâches en parallèles il est intéressant d'avoir une granularité de ressources de calcul importante, alors que lorsque le parallélisme augmente dans le graphe de tâches on peut vouloir réduire la granularité des ressources et éventuellement obtenir des ressources de calcul indépendantes. Dans cet exemple, un utilisateur averti qui utilise la deuxième interface présentée dans la Figure 5.10b peut créer un unique groupe de cœurs avec toutes les ressources de la machine pour l'exécution de la première tâche DPOTRF puis créer deux ressources pour les tâches suivantes. Par la suite, il est intéressant de réduire fortement la granularité des groupes de cœurs car la largeur du graphe de tâches s'accroît fortement, puis il est intéressant quelques DPOTRF plus loin de retrouver des granularités de ressources plus grosses telles que 2×10 et 1×20.

Une autre façon d'obtenir un résultat similaire en se basant sur les ordonnanceurs modulaires est d'utiliser un module d'ordonnancement programmé pour adapter la granularité des ressources de toute la machine à chaque tâche prête soumise d'un certain type, dans le cas de la factorisation de Cholesky le DPOTRF par exemple. Une fois la granularité des ressources adapté, un simple ordonnancement de type HEFT (MCT, Minimum Completion Time) peut être utilisé.

La tâche critique de choix de la granularité des ressources se fait selon un système de définition de tendances à l'augmentation ou la réduction de la granularité des ressources de calcul à partir des nombres de tâches prêtes et soumises. Des exemples fictifs de courbe d'évolution de ces deux valeurs sont représentés dans la Figure 5.16b. On considère notamment les nombres de tâches prêtes $n_p(i)$ entre les appels successifs $i-1$ et i à la tâche DPOTRF et $n_s(i)$ le nombre total de tâches soumises par l'application à StarPU à l'étape i. Plusieurs quantités peuvent être définies à partir de ces deux valeurs pour déterminer une tendance sur l'évolution des ressources, en considérant que l'on



(a) Exemple d'évolution dynamique de la taille de groupes de cœurs selon les DPOTRF d'une factorisation de Cholesky



(b) Exemples de courbes d'évolution des nombre de tâches prêtes et soumises au cours de tumps.

FIGURE 5.16 – Réalisation d'un ordonnanceur naïf d'adaptation de granularité de groupes de cœurs basé sur les tendances des tâches prêtes et soumises

commence et termine avec une ressource unique pour les CPUs. Si $n_p(i-1) << n_p(i)$, alors il est intéressant d'augmenter la granularité des ressources de calcul et inversement dans le cas contraire, lorsque ces changements sont suffisamment significatifs. De plus, dès lors que $n_s(i) - \sum_{0 \leq k \leq i} n_p(k) \simeq 0$ ou que $n_p(i) \simeq 0$ on considère que l'on atteint un état critique et qu'il est intéressant de n'avoir qu'un seul groupe de cœurs sur la machine.

5.3.2 Adaptation et implantation de l'ordonnanceur [25]

L'ordonnanceur implanté dans cette partie est une adaptation d'un ordonnanceur proposé dans l'article *Scheduling Independent Moldable Tasks on Multi-Cores with GPUs* de R. Bleuse *et al* [25]. Cet ordonnanceur est créé spécialement dans le cas d'une machine de type hétérogène, composée de m CPUs identiques et k GPUs identiques. De plus, l'ordonnanceur s'intéresse à l'ordonnancement de tâches moldables indépendantes. Ainsi, le problème visé est la minimisation du makespan de l'ordonnancement, défini comme le maximum du makespan sur les CPUs et du makespan sur les GPUs. Ce problème est aussi noté $(P_m, P_k|mold|C_{max})$. Deux ordonnancements sont proposés par les auteurs à partir d'un même principe : un algorithme de ρ-approximation qui, en

prenant une supposition λ le temps d'exécution d'un groupe de tâches comme entrée, retourne soit une solution à au maximum $\rho\lambda$ de l'objectif, soit une réponse correcte que l'ordonnancement en λ est impossible. De plus, les deux algorithmes fonctionnent en deux temps : d'abord un nombre de ressources est attribué aux tâches parallèles pour leur évolution, ensuite ces tâches sont ordonnancées sur le nombre de ressources demandées. Le premier ordonnancement proposé par les auteurs est un algorithme $\frac{3}{2}-$approché basé sur un algorithme d'optimisation linéaire à nombre entier qui ne sera pas implanté. Le second algorithme est un algorithme $2-$approché avec une complexité polynomiale en temps que l'on se propose d'implanter. Avec une supposition sur le temps d'exécution des tâches λ en entrée l'algorithme fonctionne de la façon suivante :

1. Allocation des tâches avec un temps moins de λ seulement sur un type d'architecture
2. Tri des tâches de façon décroissante selon le ratio de calcul $\frac{w_{j,\gamma(j,\lambda)}}{p_j}$. Pour une tâche T_j, p_j désigne son temps de calcul sur GPU, $\gamma(j,\lambda)$ le nombre de ressources nécessaires pour que la tâche T_j s'exécute en temps λ et $w_{j,\gamma(j,\lambda)}$ représente l'aire totale de calcul occupée par la tâche T_j sur les CPUs.
3. Allocation des tâches sur GPUs jusqu'à ce que chaque GPU possède une charge de plus de λ.
4. Ordonnancement des tâches rigides (avec $\gamma(j,\lambda)$ ressources) sur les CPUs avec un algorithme, par exemple de liste, $2-$approché.

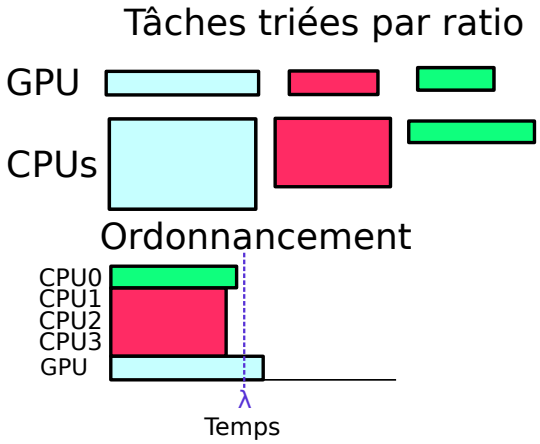


FIGURE 5.17 – Exemple d'utilisation de l'ordonnanceur proposé par [25]

Un exemple simple d'exécution de l'algorithme en temps polynomial de [25] est présentée dans la Figure 5.17. Dans cet exemple, trois tâches doivent être

ordonnancées sur une plateforme composée de 4 CPUs et 1 GPU. Une supposition de temps d'exécution λ est fournie en entrée de l'algorithme. Les tâches sont d'abord triées selon le ratio d'aire sur CPU pour atteindre λ et la performance sur GPU de la tâche. Dans cette figure, les aires des tâches triées sont représentées. Une fois fait, le GPU prend les tâches pour lesquels il a le meilleur ratio (*i.e.* les premières), ici la première tâche suffit à atteindre λ. Ensuite, les CPUs prennent les autres tâches jusqu'à atteindre λ. Dans ce cas, toutes les tâches peuvent être ordonnancés en temps moins de 2λ et l'ordonnancement est donc valide. Si ce n'était pas le cas, l'algorithme retournerait qu'avec cette entrée λ, il n'est pas possible d'obtenir un ordonnancement valide.

Une implantation de cet algorithme basée sur les ordonnanceurs modulaires (*cf.* 2.2.2) est proposée dans la Figure 5.18. L'idée générale de l'implantation et de l'algorithme est de regrouper ensemble, autant que souhaitable, les tâches prêtes (*cf.* Figure 2.2 pour un rappel sur le concept de tâches prêtes) afin de calculer un ordonnancement pour ces tâches selon l'algorithme exposé précédemment. Pendant que ces tâches en question s'exécutent, il est bénéfique d'appliquer à un nouveau lot de tâches cet algorithme afin d'en calculer l'ordonnancement. Autrement dit, la motivation de cette implantation est de réaliser un pipeline d'ordonnancement et de recouvrir l'ordonnancement avec du calcul. Pour cette implantation, plusieurs modules sont identifiés, et chacun est présenté plus en détail :

1. La calibration des tâches parallèles (SPMD ou Fork-Join)
2. La collecte de tâches et création de "sacs" de tâches
3. La prédiction itérative basée sur une dichotomie sur le temps de calcul estimé pour ces tâches λ.
4. La répartition des tâches CPU/GPU selon les étapes 2 et 3 de l'algorithme, et attribution du nombre de ressources pour les tâches CPUs.
5. Utilisation d'un nouvel algorithme simple nommé "PMCT" (Parallel Minimum Completion Time) pour les tâches parallèles, et "MCT" pour les GPUs.

Calibration de tâches parallèles

Pour calibrer les tâches parallèles, un module d'ordonnancement est utilisé. Le module vérifie si la tâche dispose au minimum d'un modèle de performance pour le cas où l'on utilise le minimum de ressources possibles (soit 1 soit spécifié par l'utilisateur), et de même pour le nombre maximum de ressources. Tant qu'une de ces informations sont manquantes pour un noyau, le nombre de ressources de la tâche est forcé au nombre de ressources manquant et la tâche est poussée sur un module PMCT. Ensuite, pour analyser la performance des tâches parallèles, les fonctions présentées en Section 5.2.4 sont utilisées.

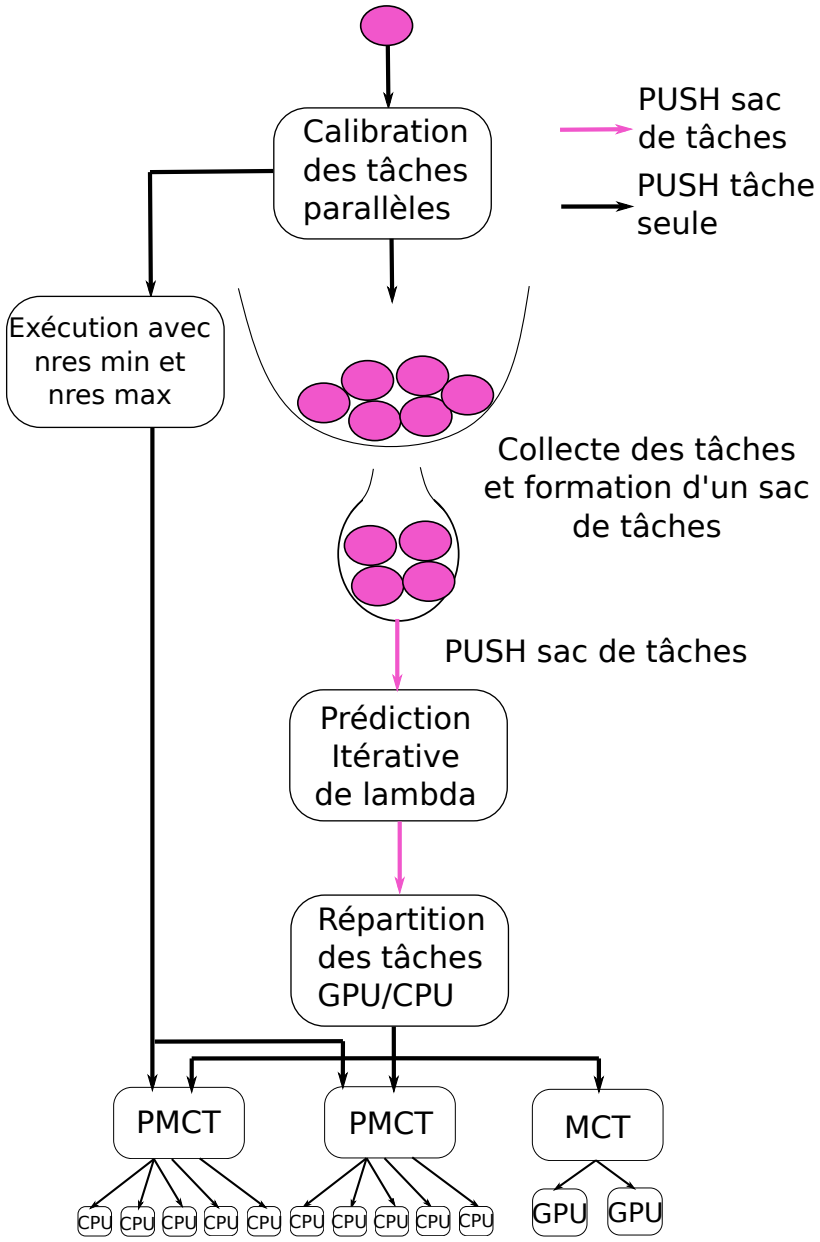


FIGURE 5.18 – Schéma de l'implantation à base de modules d'ordonnancements de l'ordonnanceur proposé par [25]

Collecte de tâches

Afin de regrouper les tâches, il est nécessaire d'avoir une vision du futur. En effet, lorsque l'on exécute un certain nombre de tâches, on se doit d'identifier le nombre de tâches qui deviendront prêtes après l'exécution de celles-ci. Avec ces informations à disposition, il est possible de décider du nombre de tâches réel

qui deviendront prêtes et disposer d'une limite minimale de tâches à attendre pour former un sac de tâches.

Pour obtenir ces informations une fonction interne de StarPU a été implantée dont l'objectif est d'analyser les structures internes de StarPU de gestion de dépendances afin de compter les tâches qui seront libérées après l'exécution d'une tâche. Pour obtenir une précision optimale, il faut que toutes les tâches soient soumises avant l'utilisation de ce genre de fonction (sinon les tâches et ces dépendances ne sont pas encore connues de StarPU). Afin de ne pas ajouter cette contrainte, un mode est ajouté pour prendre en charge les tâches inattendues, car elles n'étaient pas dans la liste des successeurs au moment de l'analyse des dépendances des tâches exécutées. Cette fonction est déclenchée grâce à un "hook" situé à la fin de l'exécution de la tâche qui retourne à l'ordonnanceur le nombre de tâches dont les dépendances viennent d'être satisfaites.

Au final, deux modes de fonctionnement de ce module sont distingués, 1) le mode critique et 2) le régime permanent. Le mode critique survient si le nombre de tâches prêtes $|T_P| << |R|$ où R est l'ensemble des ressources, ou bien si $|T_p| = 0$ à cause des tâches non comptées. Le régime permanent quand à lui survient dans le cas opposé. Dans le mode critique, ce module s'assure de pousser pour l'ordonnancement les tâches prêtes dès qu'elles sont disponibles, alors qu'on mode régime permanent lorsqu'un seuil, à déterminer expérimentalement, est atteint un groupe de tâche de la taille de ce seuil est créé ce qui permet effectivement la mise en place d'un "pipeline" d'ordonnancement.

Prédiction itérative du temps de calcul λ

La prédiction λ qui est une entrée de l'algorithme de [25] se calcule en plusieurs temps. Tout d'abord, une estimation est réalisée à partir du nombre de tâches à pousser et leurs performances respectives. Soit $t \subset T$ un ensemble représentant toutes les tâches d'un même type et pour une tâche $j \in t$ $\underline{p_j}$ est la performance de cette tâche sur un GPU et $\overline{p_j(1)}$ la performance de la tâche parallèles j en utilisant 1 CPU de la machine. Alors, on défini pour chacun de ces types deux sous-ensembles t_{cpu} et t_{gpu} dont les nombres de tâches sont régis par de simples proportions :

$$|t_{gpu}| = \left\lceil \frac{\overline{p_j}}{\underline{p_j}} \times |t| \right\rceil$$

$$|t_{cpu}| = \left\lfloor \frac{(1 - \overline{p_j}}{\underline{p_j}}) \times |t| \right\rfloor$$

En se basant sur ces proportions, on défini une première estimation λ l'entrée de l'algorithme, en considérant que tous les workers sont disponibles,

comme le meilleur temps prix par tâche moyenné par le nombre de ressources de la machine, constituée de k GPUs et m CPUs :

$$\lambda = max(\frac{1}{m} \sum_{\forall t \subset T, \forall j \in t_{cpu}} \overline{p_j(1)}, \frac{1}{k} \sum_{\forall t \subset T, \forall j \in t_{gpu}} \underline{p_j})$$

Cette estimation initiale sera toujours optimiste par rapport à la réalité. En utilisant cette estimation, un temps d'exécution prédit est retourné grâce à une simulation de l'exécution de l'algorithme en utilisant tous les modules d'ordonnancement et de ressources dans un mode dédié à cet effet. Cette simulation retourne le temps, en considérant l'occupation actuelle des ressources, que l'algorithme d'allocation utilisant l'entrée λ aurait obtenu. En utilisant ce résultat, plusieurs itérations sont réalisées en cherchant à augmenter le temps de l'estimation initiale (qui est trop optimiste) tant que la prédiction retournée est améliorée ou que cette quantité est trop éloignée de l'entrée λ.

Répartition CPU/GPU des tâches

Cette étape est une implantation des étapes 2 et 3 de l'algorithme de [25]. Pour commencer, un ratio de performance est calculé et assigné à chaque tâche, ce ratio est entre : 1) le nombre de CPUs tel que le temps de la tâche est proche de λ et 2) le temps de la tâche sur GPU. Une fois fait, les tâches sont triées selon leur ratio d'accélération par ordre décroissant. Ensuite, les GPUs piochent des tâches jusqu'à ce que leur nouvelle occupation soit supérieure à λ. C'est ensuite le tour des CPUs et ceci se répète jusqu'à épuisement des tâches à l'intérieur du sac actuel.

L'algorithme "Parallel-MCT" implanté

L'algorithme "Parallel-MCT" implémenté est un algorithme simple dont l'objectif est de minimiser le temps de complétion de l'ordonnancement en prenant en compte le temps de fin des ressources au cours du temps (FT). Pour rappel, chaque tâche j poussée ici a un nombre de ressources attribuées par l'étape précédente qui est le temps d'exécution nécessaire pour s'exécuter en temps λ, $\gamma(j, \lambda)$. Alors l'algorithme implanté est :

Algorithme 1 : Algorithme "Parallel-MCT"

```
1 tab = trier les CPUs tq avec c, d ∈ R_cpu FT(c) < FT(d) ⟹ c < d
2 for j ∈ T_cpu do
3 |   m=γ(j, λ)
4 |   assigner à j les ressources tab[0], ..., tab[m − 1]
5 end
```

Autrement dit, le principe de l'algorithme est de trier les ressources selon leur charges respectives de la plus faible à la plus grande, et assigner la tâche

parallèle aux $\gamma(j, \lambda)$ ressources avec les temps de complétion les plus faibles. Ceci permet de minimiser le temps d'attente entre les ressources en choisissant celles qui se terminent le plus tôt possible. Cette solution est moins précise qu'un algorithme de type "back-filling", mais l'avantage de cet algorithme est sa simplicité et relative rapidité de calcul.

Enfin, une propriété intéressante des modules d'ordonnancement est que l'adaptation à l'architecture de la machine peut être réalisée de la façon suivante : il suffit de créer autant de modules "PMCT" que la machine possède de "sockets" et de connecter les bons workers à ces modules. Ensuite, il est nécessaire d'équilibrer la charge des composants PMCT lorsque de nouvelles tâches CPU sont poussées.

Discussion

L'implantation de cet algorithme nécessite beaucoup de nouveaux concepts dans StarPU, tels que l'analyse de dépendances et la création de sacs de tâches. Plusieurs points de l'algorithme peuvent le rendre inefficace tel que la recherche itérative du temps de calcul du sac de tâches λ, un point sur lequel il faut faire des compromis et accepter une solution correcte trouvée rapidement plutôt qu'une solution parfaite trouvée lentement.

Cependant, c'est la première fois dans le support d'exécution StarPU que l'ordonnancement de tâches est réalisé groupe par groupe plutôt que tâche par tâche ce qui peut fournir des propriétés importantes. Le fait de regrouper des tâches et de prendre des décisions sur un groupe de tâches peut permettre une meilleure précision d'ordonnancement. De plus, si le calcul de l'ordonnancement est recouvert avec succès avec du calcul il est alors possible de passer un peu plus de temps à calculer un ordonnancement, car la machine entière sera remplie pour un certain temps une fois cette tâche terminée. Tous ces points doivent être étudiés de façon concrète avec plusieurs expériences.

5.4 Synthèse

Dans ce chapitre, l'algorithme d'ordonnancement HeteroPrio a été adapté au cas où plus de deux types de ressources existent sur la machine grâce au score d'accélération, cas qui survient lors de l'utilisation de tâches parallèles. Une étude exhaustive en simulation explore toutes les configurations de groupes de cœurs y compris plusieurs configurations hétérogènes. Ceci permet de comparer les performances des ordonnanceurs dans le cadre de l'exécution de tâches moldables, mais avec une configuration de machine statique. Dans la majorité des cas pour la factorisation de Cholesky l'utilisation de tâches parallèles améliore la qualité des deux ordonnanceurs Heft et HeteroPrio. Pour QR l'utilisation de tâches parallèles est bénéfique principalement pour l'ordonnanceur Heft car plusieurs noyaux ne passent pas à l'échelle. Ces résultats ont

été étudiés en exécution réelle et un système d'exécution d'ordonnancement de HeteroPrio calculé hors-ligne est mis en œuvre pour cela. Les résultats montrent des gains de performance significatifs lors de l'utilisation de configurations de groupes de cœurs hétérogènes pour l'exécution de tâches parallèles moldables par rapport aux tâches séquentielles et rigides.

Ensuite, une plateforme d'implantation d'ordonnanceurs de tâches parallèles moldables est proposée. L'objectif de cette plateforme est de fournir des outils et interfaces afin de permettre la reconfiguration dynamique des groupes de cœurs pour l'exécution de tâches parallèles. L'approche mise en œuvre s'inspire du fonctionnement des ordonnanceurs existants qui reposent sur l'utilisation de deux phases, une phase d'allotement pour décider du nombre de ressources attribuées à une tâche et une phase d'allocation de la tâche sur les ressources de calcul. Pour permettre la définition de ces deux phases, un système d'annotation de tâches est proposé. De plus, un système de tâches compagnons est mis en place afin de contrôler à l'ordonnancement l'état des ressources de façon automatique. Deux ordonnanceurs de tâches moldables sont proposés. Un premier ordonnanceur est basé sur un partitionnement de la machine le long du chemin critique selon la quantité de parallélisme disponible. Un second ordonnanceur est basé sur une contribution récente [25] proposant un ordonnanceur de tâches moldables pour machine hétérogène en utilisant une approximation duale et l'estimation de date limite d'exécution d'un groupe de tâches.

Chapitre 6

Conclusion

La complexification des architectures parallèles, devenues hiérarchiques et hétérogènes, est à l'origine du succès grandissant des supports d'exécution dynamiques. De plus en plus d'applications reposent en effet – directement ou indirectement au travers de langages tels que OpenMP – sur une distribution dynamique des tâches de calcul sur les unités de calcul (coeurs ou accélérateurs) et sur les transferts automatiques de données pour maintenir la cohérence mémoire.

Bien que les progrès récents dans le domaine des supports d'exécution permettent aujourd'hui d'atteindre des performances de tout premier plan, ces performances s'obtiennent au prix d'une délicate phase de calibrage permettant de déterminer la granularité des calculs qui fournira le meilleur compromis entre le rendement des coeurs traditionnels et celui des puissants accélérateurs de calcul. Mais l'adoption d'une granularité uniforme pour l'intégralité de l'application n'est pas optimal, comme le montre les expériences de la Section 3.3 et notamment de la Figure 3.14, remettant en cause le découpage des données à l'exécution : il est possible d'aller plus vite en adaptant la granularité des calculs ! Ce phénomène représente la limite actuelle la plus importante des supports d'exécution, au point que même l'efficacité théorique des supports d'exécution sur certains problèmes calculées à base de bornes [13] repose sur l'utilisation de granularité unique, alors que celle-ci devrait être adaptative...

Cette thèse apporte une contribution originale au problème de l'adaptation dynamique de granularité, en proposant de reconfigurer virtuellement l'architecture sous-jacente pour s'adapter aux tâches à exécuter. Regrouper les ressources pour résoudre le problème de granularité pourvoie plusieurs avantages tels que homogénéiser les ressources de calcul hétérogènes comme les GPUs et CPUs, améliorer la localité des données et l'utilisation des ressources ainsi que réutiliser des implantations de bibliothèques hautes performances existantes telles que la MKL parallèle en algèbre linéaire.

Pour cela, trois contributions principales sont proposées. D'abord, dans le Chapitre 3 le modèle de tâches parallèles est étudié et implanté dans StarPU.

Ce modèle repose sur la composition de bibliothèques de calcul et de supports d'exécution (notamment StarPU et OpenMP) pour réaliser une imbrication du parallélisme. Une évaluation expérimentale est conduite grâce à l'utilisation de tâches parallèles rigides sur l'Intel KNL et une plateforme hétérogène équipée de 4 GPUs mettant en lumière la supériorité de l'approche sur les bibliothèques existantes pour la factorisation de Cholesky.

Ensuite, dans le Chapitre 4 une étude de cas applicatifs justifiant l'utilisation de tâches parallèles est menée. Pour l'application de mécanique des fluides FLUSEPA développée par Airbus DS, l'utilisation de tâches parallèles permet une augmentation des performances et de palier au manque de parallélisme du graphe de tâches généré. De plus, une implantation préliminaire de l'algorithme $PA = LU$ démontre l'intérêt d'utiliser une tâche parallèle pour la tâche de factorisation de la matrice afin de masquer au support d'exécution les nombreuses synchronisations nécessaires réalisées à l'intérieur de la tâche dans ce cas. En effet, l'utilisation du modèle de tâches pures implique de nombreuses tâches et réductions pour chaque colonne de la matrice. Bien que préliminaire, cette implantation permet l'obtention de performances compétitives avec les implantations existantes, notamment MAGMA dans le cadre d'une machine hétérogène.

Enfin, le Chapitre 5 propose une étude d'ordonnancement de tâches parallèles dans le cas moldable. Il est montré grâce à une évaluation théorique et pratique que l'utilisation de tâches parallèles moldables permet d'obtenir de meilleures performances pour la factorisation de Cholesky et améliorer la qualité des ordonnanceurs HEFT et HETEROPRIO. De plus, une plateforme dédiée à l'implantation d'ordonnanceurs de tâches parallèles moldables à l'intérieur de StarPU est proposée afin de permettre aux ordonnanceurs de contrôler dynamiquement la granularité des ressources.

Perspectives

Tout d'abord nos travaux offrent la possibilité d'implémenter des ordonnanceurs de tâches moldables et il est maintenant possible de concilier études théoriques et pratiques en la matière. La Section 5.3.2 propose une adaptation d'un algorithme d'ordonnancement récent proposé par [25] qui reste à évaluer. D'autres travaux théoriques récents proposent des ordonnanceurs de tâches moldables pour machine hétérogène [34, 109]. Il serait donc intéressant de mettre en œuvre ces ordonnanceurs afin d'étudier leur comportement et de les peaufiner en collaboration étroite avec les membres de la communauté de l'ordonnancement. De plus, ce travail ouvre aussi quelques perspectives que nous présentons sous les quatre points suivants : appliquer nos travaux à de nouveaux cas, améliorer la composition de bibliothèques et des supports d'exécution, faire évoluer le modèle de tâches vers un modèle hiérarchique, et, enfin,

utiliser des tâches parallèles en mode distribué.

Étude de cas applicatifs L'étude présentée dans cette thèse se concentre sur un code de mécanique des fluides, FLUSEPA et sur deux algorithmes d'algèbre linéaire dense, la factorisation de Cholesky et la factorisation $PA = LU$. Néanmoins la technique proposée ici peut s'appliquer à tout programme codé à l'aide de StarPU ou tout autre support d'exécution équivalent : il s'agit de fournir une version multithreadée de chaque noyau de calcul que l'on souhaite voir s'exécuter de façon parallèle. On peut alors espérer obtenir de meilleures performances lorsque la perte d'efficacité des noyaux de calcul parallélisés est compensée par une exploitation plus équilibrée de toutes les ressources la machine (GPUs, cœurs, caches et bande passante mémoire). Nous pensons aussi que cette technique peut être particulièrement bénéfique aux applications présentant un parallélisme hiérarchique tels le solveur d'algèbre linéaire creux `qr_mumps` ou les codes exploitant la méthode *Adaptive Mesh Refinement*. En effet, dans ces cas, les tâches parallèles permettent non seulement d'ajuster le grain du calcul en agissant sur le curseur parallélisme interne / parallélisme externe, mais aussi d'équilibrer la charge à la volée car ces applications présentent des tâches aux quantités de calcul très variées. Une autre classe d'applications qui mérite d'être étudiée est celle où les synchronisations sont nombreuses et critiques à l'image des *stencils*. Tout comme la factorisation $PA = LU$, l'expression de ces algorithmes en tâches pures peut être inadapté car chaque synchronisation implique la terminaison et la création de plusieurs tâches alors que l'utilisation de tâches parallèles permet de masquer celles-ci à l'intérieur d'un noyau de calcul.

Composition de bibliothèques et supports d'exécution L'implantation de tâches parallèles proposée permet la composition de bibliothèques et supports d'exécution pour imbriquer les parallélismes. Afin d'améliorer les performances de ce mode d'utilisation, plusieurs questions importantes restent en suspend. Comment améliorer le partage d'informations entre les couches applicatives et leurs supports d'exécution respectifs pour co-organiser l'ordonnancement du calcul global de l'application ? En effet, le support d'exécution externe qui gère le parallélisme de tâches possède des informations sur la structure du calcul que le support d'exécution interne ne possède pas, mais dont il pourrait bénéficier. De plus, comment se passer de l'utilisation en boîte noire, présentée dans cette thèse, quand nécessaire ? Par exemple pour optimiser la réutilisation des données et des formats internes entre différents appels consécutifs à une même bibliothèque et à son support d'exécution.

Un problème trop souvent oublié, qui ressort dans les expériences de la Section 3.3 concerne la mesure de performances dans le cadre d'une machine parallèle en environnement bruité, où les unités de calcul sont en concurrence pour l'accès aux ressources. Plusieurs types de modèles de performances existent,

soit non bruité soit bruité, et dans le cas bruité la performance observée peut aussi dépendre du type de travail réalisé en parallèle ainsi que de la charge actuelle de la machine. L'utilisation de tâches parallèles permet d'alléger ce problème, cependant la mise en œuvre de modèles capables de prendre en compte la position des ressources agrégées reste à étudier. En effet, les architectures actuelles étant fortement hiérarchisées, les performances ne sont de façon générale pas équivalentes selon la position dans la machine des ressources agrégées pour l'exécution d'une tâche parallèle. Par exemple, les noyaux d'algèbre linéaire de type BLAS1 peuvent bénéficier de l'utilisation de cœurs éloignés afin d'augmenter la bande passante agrégée, alors que les noyaux de type BLAS3 ont le comportement inverse du fait de la réutilisation des données. Enfin, un dernier problème important concerne la prise de mesure pour les accélérateurs et notamment les GPUs dans le cadre de l'utilisation de la technologie "multistream". Cette technique permet de considérer les GPUs comme capables de plusieurs calculs simultanés, ce qui permet de baisser la granularité des calculs sur une machine hétérogène. Cependant, le manque de contrôle sur l'ordonnancement des calculs dans la carte et l'utilisation interne des ressources rend les performances fortement imprévisibles. Par la suite, il peut être intéressant de considérer l'utilisation des tâches parallèles présentées dans cette thèse sur un GPU qui permettrait de programmer l'exécution de noyaux sur des sous parties définies de celui-ci.

Les tâches hiérarchiques, une nouvelle évolution du modèle de tâches
Les tâches hiérarchiques présentées en Section 2.3.3 sont une nouvelle technique qui permet la mise en place d'un modèle à finalité équivalente à celle des tâches parallèles, mais plus détaillé et avec plus de capacités. En effet, ce modèle permet lors de l'exécution d'une de ces tâches parallèles (hiérarchiques), tel des supertâches, aussi nommées "bulles" de soumettre de nouvelles tâches, plus fines, construisant une hiérarchie de tâches. Puisque l'intérieur des tâches parallèles est connu du support d'exécution capable d'exécuter ces tâches hiérarchiques, il est alors possible de reconnaître les différentes dépendances fines entre les tâches hiérarchiques lorsqu'elles existent. Ce modèle propose aussi une nouvelle évolution du modèle de taches, car celles-ci sont désormais pseudo-malléables et non plus moldables : une même tâche hiérarchique peut s'exécuter sur un nombre variable de ressources au cours de son exécution.

Les tâches hiérarchiques possèdent d'autre propriétés d'intérêt. Il est par exemple possible de contrôler la soumission de façon plus naturelle et plus complète que l'interruption du flot de soumission des tâches présenté dans la Section 4.2. En effet, à la soumission d'une tâche hiérarchique la soumission des tâches internes peut être conditionnée par l'état des données considérées, créant une soumission de graphes de tâches complètement dynamiques. Cependant plusieurs points rendent cette technique complexe à mettre en œuvre et à utiliser. En premier lieu, l'ordonnancement se retrouve encore plus complexifié

que dans le cas de tâches moldables car plusieurs niveaux de tâches existent et chaque niveau doit être ordonnancé séparément. De plus, le modèle de performance dans le cadre d'exécution de tâches malléable est complexe à mettre en œuvre, ce qui justifiera peut être de fixer les ressources pour l'exécution d'une tâche hiérarchique afin de retrouver un modèle de tâches moldables en pratique. Enfin, puisque ces approches permettent d'obtenir les mêmes propriétés que les tâches parallèles, à savoir l'adaptation de la granularité des calculs aux ressources, il serait très intéressant de comparer en détail les deux approches et leurs comportements respectifs.

Les tâches parallèles sur plateforme distribuée La thèse de Marc Sergent [103] met en avant pour le passage à l'échelle y compris à l'Exascale d'un support d'exécution tel que StarPU le besoin de réduire le nombre de tâches parcourues dans le graphe par chaque nœud de machine distribuée ainsi que le nombre de communications MPI réalisées. Une première solution est d'utiliser une couche de passage de message tel que NewMadeleine-MadMPI [1] qui permet l'agrégation des communications afin d'en augmenter le grain et d'en limiter le nombre. Une autre solution est de déléguer en partie le problème à la compilation dans le cadre d'une répartition statique du calcul. Enfin, pour régler ce problème les tâches parallèles peuvent aussi être utilisées. En effet comme mis en avant dans le cadre de cette thèse, les tâches parallèles permettent de grossir la granularité des tâches sans perte de performances ce qui implique une génération de moins de communications de plus grande taille.

Enfin, le dernier point concerne la généralisation de ce travail à n'importe quelle échelle logicielle et notamment au distribué : dans le cadre de cette thèse l'imbrication du parallélisme a été réalisée à l'intérieur d'un nœud de machine. Cette question se pose d'autant plus que des algorithmes la factorisation $PA = LU$ présentée requiert l'utilisation de tâches parallèles afin de masquer au support d'exécution les nombreuses synchronisations et réductions nécessaires qui complexifieraient le graphe de tâches. Il est fort probable que pour la version distribuée de cet algorithme un mécanisme similaire soit nécessaire, auquel cas non seulement la composition de bibliothèques de calcul et de support d'exécution doit être menée, mais aussi la composition de bibliothèques de communications.

1. http://pm2.gforge.inria.fr/newmadeleine/doc/

Bibliographie

[1] David Bau III / Lloyd N. Trefethen. *Numerical linear algebra*. Society for Industrial et Applied Mathematics, 1996. ISBN : 978-0-89871-361-9 (cf. p. 75, 76).

[2] J. Dongarra A. Petitet R. C. Whaley et A. Cleary. *High Performance Linpack*. http://www.netlib.org/benchmark/hpl (cf. p. 96).

[3] E. Agullo et al. "A Hybridization Methodology for High-Performance Linear Algebra Software for GPUs". In : *GPU Computing Gems, Jade Edition* 2 (2011), p. 473-484 (cf. p. 12, 44).

[4] E. Agullo et al. "Fully Empirical Autotuned QR Factorization For Multicore Architectures". In : *CoRR* abs/1102.5328 (2011) (cf. p. 12).

[5] E. Agullo et al. "LU factorization for accelerator-based systems". In : *AICCSA*. 2011, p. 217-224 (cf. p. 12).

[6] E. Agullo et al. "Multifrontal QR Factorization for Multicore Architectures over Runtime Systems". In : *Euro-Par 2013 Parallel Processing*. 2013, p. 521-532 (cf. p. 12).

[7] E. Agullo et al. "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects". In : *Journal of Physics: Conference Series* 180.1 (2009) (cf. p. 11, 32, 44, 113).

[8] E. Agullo et al. "QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators". In : *Proceedings of 25th International Parallel and Distributed Processing Symposium (IPDPS'11)*. 2011, p. 932-943 (cf. p. 12).

[9] Emmanuel Agullo et al. "Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model". In : *IEEE Transactions on Parallel and Distributed Systems* (2017), to appear (cf. p. 67).

[10] Emmanuel Agullo et al. "Are Static Schedules so Bad ? A Case Study on Cholesky Factorization". In : *Proceedings of IPDPS'16*. 2016 (cf. p. 19, 57, 58, 100, 101).

[11] Emmanuel Agullo et al. “Bridging the gap between OpenMP and task-based runtime systems for the fast multipole method”. In : *IEEE Transactions on Parallel and Distributed Systems* (avr. 2017), p. 14 (cf. p. 8).

[12] Emmanuel Agullo et al. “Bridging the Gap between Performance and Bounds of Cholesky Factorization on Heterogeneous Platforms”. In : *Heterogeneity in Computing Workshop 2015*. Hyderabad, India, mai 2015 (cf. p. 102).

[13] Emmanuel Agullo et al. “Bridging the Gap between Performance and Bounds of Cholesky Factorization on Heterogeneous Platforms”. In : *Heterogeneity in Computing Workshop 2015*. 2015 (cf. p. 135).

[14] Emmanuel Agullo et al. “Implementing Multifrontal Sparse Solvers for Multicore Architectures with Sequential Task Flow Runtime Systems”. In : *ACM Trans. Math. Softw.* 43.2 (août 2016) (cf. p. 46).

[15] Emmanuel Agullo et al. *Poster: Matrices over Runtime Systems at Exascale*. Nov. 2012 (cf. p. 24, 32, 105).

[16] Emmanuel Agullo et al. “Task-based FMM for Heterogeneous Architectures”. In : *Concurr. Comput. : Pract. Exper.* 28.9 (juin 2016), p. 2608-2629. ISSN : 1532-0626 (cf. p. 19, 100).

[17] Emmanuel Agullo et al. “Task-Based FMM for Multicore Architectures”. In : *SIAM J. Scientific Computing* 36.1 (2014) (cf. p. 12).

[18] E. Anderson et al. “LAPACK: A Portable Linear Algebra Library for High-performance Computers”. In : *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*. Supercomputing ’90. New York, New York, USA : IEEE Computer Society Press, 1990, p. 2-11. ISBN : 0-89791-412-0 (cf. p. 10).

[19] The OpenMP ARB. *The OpenMP® API specification for parallel programming*. http://openmp.org/. 2012 (cf. p. 6, 7).

[20] C. Augonnet et al. “StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures”. In : *Concurr. Comput. : Pract. Exper.* 23 (2 fév. 2011), p. 187-198 (cf. p. 7).

[21] C’edric Augonnet. “Scheduling Tasks over Multicore machines enhanced with Accelerators: a Runtime System’s Perspective”. PhD Thesis. Universit’e Bordeaux 1, 2011 (cf. p. 29).

[22] E. Ayguadé et al. “An Extension of the StarSs Programming Model for Platforms with Multiple GPUs”. In : *Euro-Par 2009*. Delft, The Netherlands, 2009, p. 851-862. ISBN : 978-3-642-03868-6 (cf. p. 7).

[23] Michael Bauer et al. “Legion: expressing locality and independence with logical regions”. In : *SC Conference on High Performance Computing Networking, Storage and Analysis, SC ’12, Salt Lake City, UT, USA - November 11 - 15, 2012*. 2012, p. 66 (cf. p. 7).

[24] L. S. Blackford et al. “ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance”. In : *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*. 1996, p. 5-5 (cf. p. 10).

[25] Raphaël Bleuse et al. “Scheduling Independent Moldable Tasks on Multi-Cores with GPUs”. In : *IEEE Trans. Parallel Distrib. Syst.* 28.9 (2017), p. 2689-2702 (cf. p. 14, 30, 126-128, 130-132, 134, 136).

[26] G. Bosilca et al. “Dense Linear Algebra on Distributed Heterogeneous Hardware with a Symbolic DAG Approach”. In : *Scalable Computing and Communications: Theory and Practice* (2013) (cf. p. 12, 32).

[27] George Bosilca et al. “Dense Linear Algebra on Distributed Heterogeneous Hardware with a Symbolic DAG Approach”. In : *Scalable Computing and Communications: Theory and Practice* (2013) (cf. p. 7, 44).

[28] George Bosilca et al. “PaRSEC: Exploiting Heterogeneity to Enhance Scalability”. In : *Computing in Science and Engineering* 15.6 (2013), p. 36-45 (cf. p. 7).

[29] F. Broquedis et al. “hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications”. In : *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. PDP ’10. Washington, DC, USA : IEEE Computer Society, 2010, p. 180-186. isbn : 978-0-7695-3939-3 (cf. p. 11, 43).

[30] A. Buttari. *Fine-grained multithreading for the multifrontal QR factorization of sparse matrices*. To appear in SIAM SISC and APO technical report number RT-APO-11-6. 2013 (cf. p. 11).

[31] A. Buttari et al. “A class of parallel tiled linear algebra algorithms for multicore architectures”. In : *Parallel Comput.* 35 (1 2009), p. 38-53. issn : 0167-8191 (cf. p. 11).

[32] A. Buttari et al. “A class of parallel tiled linear algebra algorithms for multicore architectures”. In : *Par. Comp.* 35.1 (2009), p. 38-53. issn : 0167-8191 (cf. p. 75).

[33] A. Buttari et al. “The impact of multicore on math software”. In : *Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing*. PARA’06. Umeå, Sweden : Springer-Verlag, 2007, p. 1-10. isbn : 3-540-75754-6, 978-3-540-75754-2 (cf. p. 11).

[34] Louis-Claude Canon, Loris Marchal et Frédéric Vivien. "Low-Cost Approximation Algorithms for Scheduling Independent Tasks on Hybrid Platforms". In : *Euro-Par 2017: Parallel Processing: 23rd International Conference on Parallel and Distributed Computing, Santiago de Compostela, Spain, August 28 – September 1, 2017, Proceedings*. Sous la dir. de Francisco F. Rivera, Tomás F. Pena et José C. Cabaleiro. Cham : Springer International Publishing, 2017, p. 232-244 (cf. p. 14, 136).

[35] E. Chan et al. "SuperMatrix: a multithreaded runtime scheduling system for algorithms-by-blocks". In : *PPOPP*. 2008, p. 123-132 (cf. p. 8).

[36] P. Choudhury, P.P. Chakrabarti et R. Kumar. "Online Scheduling of Dynamic Task Graphs with Communication and Contention for Multiprocessors". In : *Parallel and Distributed Systems, IEEE Transactions on* 23.1 (2012), p. 126 -133. issn : 1045-9219 (cf. p. 14).

[37] Jérôme Clet-Ortega. "Exploitation efficace des architectures parallèes de type grappes de NUMA à l'aide de modèles hybrides de programmation". Thèse de doct. 351 cours de la Libération — 33405 TALENCE cedex : Université Bordeaux 1, avr. 2012 (cf. p. 9).

[38] E. F. Codd. "Multiprogram Scheduling: Parts 1 and 2. Introduction and Theory". In : *Commun. ACM* 3.6 (juin 1960), p. 347-350. issn : 0001-0782 (cf. p. 13).

[39] E. F. Codd. "Multiprogram Scheduling: Parts 3 and 4. Scheduling Algorithm and External Constraints". In : *Commun. ACM* 3.7 (juil. 1960), p. 413-418. issn : 0001-0782 (cf. p. 13).

[40] Intel Corporation. *MKL Reference Manual*. http://software.intel.com/en-us/articles/intel-mkl (cf. p. 10, 32).

[41] Nvidia Corporation. *cuBLAS Library User Guide*. http://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf (cf. p. 12).

[42] Jean Marie Couteyen Carpaye. "Contributions to the parallelization and the scalability of the FLUSEPA code". Theses. Université de Bordeaux, sept. 2016 (cf. p. 69, 70, 72).

[43] James Demmel et al. "Communication-optimal Parallel and Sequential QR and LU Factorizations". In : *SIAM Journal on Scientific Computing* 34.1 (2012), A206-A239 (cf. p. 76).

[44] F. Desprez et F. Suter. "A Bi-criteria Algorithm for Scheduling Parallel Task Graphs on Clusters". In : *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*. 2010, p. 243 -252 (cf. p. 14).

[45] S. Donfack, A. K. Gupta et L. Grigori. "Adapting communication-avoiding LU and QR factorizations to multicore architectures". In : *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. T. 00. Avr. 2010, p. 1-10 (cf. p. 76).

[46] Simplice Donfack et al. "A Survey of Recent Developments in Parallel Implementations of Gaussian Elimination". In : *Concurrency and Computation: Practice and Experience* 27.5 (2014), p. 1292-1309 (cf. p. 75, 76).

[47] J. Dongarra et al. *LINPACK Users' Guide*. Society for Industrial et Applied Mathematics, 1979. eprint : http://epubs.siam.org/doi/pdf/10.1137/1.9781611971811 (cf. p. 10).

[48] Jack Dongarra et al. "Achieving Numerical Accuracy and High Performance Using Recursive Tile Lu Factorization With Partial Pivoting". In : *Concurrency and Computation: Practice and Experience* 26.7 (2013), p. 1408-1431 (cf. p. 78, 79).

[49] Maciej Drozdowski. *Scheduling for Parallel Processing*. 1st. Springer Publishing Company, Incorporated, 2009. isbn : 1848823096, 9781848823099 (cf. p. 13).

[50] P.-F. Dutot, G. Mounié et D. Trystram. "Scheduling Parallel Tasks: Approximation Algorithms". In : *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Sous la dir. de Joseph T. Leung. chapter 26. CRC Press, 2004, p. 26-1 -26-24 (cf. p. 13, 14).

[51] Jr. E. G. Coffman et al. "Performance Bounds for Level-Oriented Two-Dimensional Packing Algorithms". In : *SIAM Journal on Computing* 9.4 (1980), p. 808-826 (cf. p. 13).

[52] M. Faverge et P. Ramet. "Dynamic Scheduling for sparse direct Solver on NUMA architectures". In : *Proceedings of PARA'2008*. Trondheim, Norway, mai 2008 (cf. p. 11).

[53] Mathieu Faverge. "Ordonnancement hybride statique-dynamique en algèbre linéaire creuse pour de grands clusters de machines NUMA et multi-coeurs". Thèse de doctorat dirigée par Namyst, Raymond et Roman, Jean Informatique Bordeaux 1 2009. Thèse de doct. 2009 (cf. p. 11).

[54] Dror G. Feitelson et al. "Theory and Practice in Parallel Job Scheduling". In : *Proceedings of the Job Scheduling Strategies for Parallel Processing*. IPPS '97. London, UK, UK : Springer-Verlag, 1997, p. 1-34. isbn : 3-540-63574-2 (cf. p. 13).

[55] M. Frigo, C.E. Leiserson et K.H. Randall. "The implementation of the Cilk-5 multithreaded language". In : *SIGPLAN Not.* 33.5 (1998), p. 212-223. issn : 0362-1340 (cf. p. 6).

[56] M. Garland et al. “Parallel Computing Experiences with CUDA”. In : *IEEE Micro* 28.4 (2008), p. 13-27 (cf. p. 12).

[57] R. L. Graham. “Bounds for Certain Multiprocessing Anomalies”. In : *Bell System Technical Journal* 45.9 (1966), p. 1563-1581. issn : 1538-7305 (cf. p. 13).

[58] Laura Grigori, James W. Demmel et Hua Xiang. “Communication Avoiding Gaussian Elimination”. In : *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. SC ’08. Austin, Texas : IEEE Press, 2008, 29:1-29:12. isbn : 978-1-4244-2835-9 (cf. p. 76).

[59] A. Gupta. “A Shared- and distributed-memory parallel general sparse direct solver”. In : *Appl. Algebra Eng. Commun. Comput.* 18.3 (2007), p. 263-277 (cf. p. 11).

[60] A. Gupta, S. Koric et T. George. “Sparse matrix factorization on massively parallel computers”. In : *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. SC ’09. Portland, Oregon : ACM, 2009, 1:1-1:12. isbn : 978-1-60558-744-8 (cf. p. 11).

[61] Fred G. Gustavson. “New Generalized Data Structures for Matrices Lead to a Variety of High Performance Dense Linear Algebra Algorithms”. In : *Applied Parallel Computing. State of the Art in Scientific Computing*. Sous la dir. de Jack Dongarra, Kaj Madsen et Jerzy Waśniewski. Berlin, Heidelberg : Springer Berlin Heidelberg, 2006, p. 11-20. isbn : 978-3-540-33498-9 (cf. p. 11).

[62] Yuxiong H., Wen-Jing H. et C.E. Leiserson. “Provably Efficient Online Nonclairvoyant Adaptive Scheduling”. In : *Parallel and Distributed Systems, IEEE Transactions on* 19.9 (2008), p. 1263 -1279. issn : 1045-9219 (cf. p. 14).

[63] A. Haidar et al. “LU, QR, and Cholesky Factorizations: Programming Model, Performance Analysis and Optimization Techniques for the Intel Knights Landing Xeon Phi”. In : *IEEE High Performance Extreme Computing Conference (HPEC’16)*. 2016, to appear (cf. p. 53).

[64] Harshvardhan et al. “The STAPL Parallel Graph Library”. In : *Languages and Compilers for Parallel Computing*. Sous la dir. d’Hironori Kasahara et Keiji Kimura. Berlin, Heidelberg : Springer Berlin Heidelberg, 2013, p. 46-60. isbn : 978-3-642-37658-0 (cf. p. 9).

[65] T. D. R. Hartley, E. Saule et Ü. V. Çatalyürek. “Improving performance of adaptive component-based dataflow middleware”. In : *Parallel Computing* 38.6-7 (2012), p. 289-309 (cf. p. 7).

[66] P. Hénon, P. Ramet et J. Roman. "PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems". In : *Parallel Computing* 28.2 (jan. 2002), p. 301-321 (cf. p. 11).

[67] Everton Hermann et al. "Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations". In : *Euro-Par 2010 - Parallel Processing*. T. 6272. 2010, p. 235-246. isbn : 978-3-642-15290-0 (cf. p. 7).

[68] J. D. Hogg, J. K. Reid et J. A. Scott. "Design of a Multicore Sparse Cholesky Factorization Using DAGs". In : *SIAM J. Scientific Computing* 32.6 (2010), p. 3627-3649 (cf. p. 11).

[69] Andra-Ecaterina Hugo. "Composability of parallel codes on heterogeneous architectures". Theses. Université de Bordeaux, déc. 2014 (cf. p. 9, 15, 21, 25, 29, 35).

[70] Sascha Hunold. "One Step Toward Bridging the Gap Between Theory and Practice in Moldable Task Scheduling With Precedence Constraints". In : *Concurrency and Computation: Practice and Experience* 27.4 (2015), p. 1010-1026. issn : 1532-0634 (cf. p. 14).

[71] F. D. Igual et al. "The FLAME approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations". In : *J. Parallel Distrib. Comput.* 72.9 (2012), p. 1134-1143 (cf. p. 12).

[72] Emmanuel Jeannot. "Allocation de graphes de tâches paramétrés et génération de code". ftp://ftp.ens-Lyon.fr/pub/LIP/Rapports/PhD/PhD1999/PhD1999-08.ps.Z. Thèse de doct. École Normale Supérieure de Lyon, France, oct. 1999 (cf. p. 7).

[73] Thierry Joffrain, Enrique S. Quintana-Ortí et Robert A. van de Geijn. "Rapid Development of High-Performance Out-of-Core Solvers". In : *Applied Parallel Computing. State of the Art in Scientific Computing: 7th International Workshop, PARA 2004, Lyngby, Denmark, June 20-23, 2004. Revised Selected Papers*. Sous la dir. de Jack Dongarra, Kaj Madsen et Jerzy Waśniewski. Berlin, Heidelberg : Springer Berlin Heidelberg, 2006, p. 413-422. isbn : 978-3-540-33498-9 (cf. p. 75).

[74] L. V. Kalé et S. Krishnan. "CHARM++: A Portable Concurrent Object Oriented System Based On C++". In : *OOPSLA*. 1993, p. 91-108 (cf. p. 6).

[75] James E. Kelley Jr et Morgan R. Walker. "Critical-path Planning and Scheduling". In : *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*. IRE-AIEE-ACM '59 (Eastern). Boston, Massachusetts : ACM, 1959, p. 160-173 (cf. p. 13).

[76] Kyungjoo Kim, Victor Eijkhout et Robert A. van de Geijn. *Dense Matrix Computation on a Heterogenous Architecture: A Block Synchronous Approach*. Rapp. tech. TR-12-04. Texas Advanced Computing Center, The University of Texas at Austin, 2012 (cf. p. 26).

[77] Suraj Kumar. "Scheduling of Dense Linear Algebra Kernels on Heterogeneous Resources". Thèse de doctorat dirigée par Beaumont, Olivier et Thibault, Samuel Informatique Bordeaux 2017. Thèse de doct. 2017 (cf. p. 101).

[78] D. M. Kunzman et L. V. Kalé. "Programming heterogeneous clusters with accelerators using object-based programming". In : *Scientific Programming* 19.1 (2011), p. 47-62 (cf. p. 7).

[79] J. Kurzak et J. Dongarra. "Fully Dynamic Scheduler for Numerical Computing on Multicore Processors". In : *LAPACK working note* lawn220 (2009) (cf. p. 8).

[80] Yu-Kwong Kwok et I. Ahmad. "Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors". In : *IEEE Transactions on Parallel and Distributed Systems* 7.5 (1996), p. 506-521. issn : 1045-9219 (cf. p. 13).

[81] X. Lacoste et al. *Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes*. Rapport de recherche RR-8446. INRIA, jan. 2014, p. 25 (cf. p. 12).

[82] C. L. Lawson et al. "Basic Linear Algebra Subprograms for Fortran Usage". In : *ACM Trans. Math. Softw.* 5.3 (sept. 1979), p. 308-323. issn : 0098-3500 (cf. p. 10).

[83] X. S. Li. "Evaluation of Sparse LU Factorization and Triangular Solution on Multicore Platforms". In : *VECPAR*. 2008, p. 287-300 (cf. p. 11, 14).

[84] Andrea Lodi, Silvano Martello et Michele Monaci. "Two-dimensional packing problems: A survey". In : *European Journal of Operational Research* 141.2 (2002), p. 241 -252. issn : 0377-2217 (cf. p. 13).

[85] H. Ltaief et al. "A Scalable High Performant Cholesky Factorization for Multicore with GPU Accelerators". In : *VECPAR*. 2010, p. 93-101 (cf. p. 12).

[86] Hatem Ltaief et Rio Yokota. "Data-driven execution of fast multipole methods". In : *Concurrency and Computation: Practice and Experience* 26.11 (2014), p. 1935-1946 (cf. p. 12).

[87] C.-K. Luk, S. Hong et H. Kim. "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping". In : *MICRO*. 2009, p. 45-55 (cf. p. 6).

[88] R. Nath, S. Tomov et J. Dongarra. "Accelerating GPU Kernels for Dense Linear Algebra". In : *VECPAR*. 2010, p. 83-92 (cf. p. 12).

[89] R. Nath, S. Tomov et J. Dongarra. "An Improved MAGMA GEMM For Fermi Graphics Processing Units". In : *IJHPCA* 24.4 (2010), p. 511-515 (cf. p. 12).

[90] Tan Nguyen et al. "Perilla: Metadata-Based Optimizations of an Asynchronous Runtime for Adaptive Mesh Refinement". In : *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*. Nov. 2016, nil (cf. p. 9).

[91] OpenACC-standard.org. *The OpenACC® Application Programming Interface*. 2017 (cf. p. 7).

[92] Matthieu Ospici et al. "SGPU-2: a runtime system for using large applications on clusters of hybrid nodes". In : *Proceedings of the WHMC2011 Workshop on Hybrid Multi-core Computing*. Bangalore, India, déc. 2011, p. 1-8 (cf. p. 7).

[93] Heidi Pan, Benjamin Hindman et Krste Asanović. "Composing parallel software efficiently with lithe". In : *SIGPLAN Not.* 45 (6 2010), p. 376-387. issn : 0362-1340 (cf. p. 9).

[94] Sankaralingam Panneerselvam et Michael Swift. "Rinnegan: Efficient Resource Use in Heterogeneous Architectures". In : *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. PACT '16. Haifa, Israel : ACM, 2016, p. 373-386. isbn : 978-1-4503-4121-9 (cf. p. 10).

[95] INTERTWinE Project. *Best Practice Guide for Writing OpenMP/OmpSs/StarPU + Multithreaded Libraries Interoperable Programs*. 2017 (cf. p. 9).

[96] G. Quintana-Ortí et al. "Programming matrix algorithms-by-blocks for thread-level parallelism". In : *ACM Trans. Math. Softw.* 36.3 (2009) (cf. p. 11).

[97] G. Quintana-Ortí et al. "Solving dense linear systems on platforms with multiple hardware accelerators". In : *PPOPP*. 2009, p. 121-130 (cf. p. 12).

[98] A. Radulescu et A.J.C. van Gemund. "A low-cost approach towards mixed task and data parallel scheduling". In : *Parallel Processing, International Conference on, 2001.* 2001, p. 69 -76 (cf. p. 14).

[99] Lawrence Rauchwerger, Francisco Arzu et Koji Ouchi. "Standard Templates Adaptive Parallel Library (STAPL)". In : *Languages, Compilers, and Run-Time Systems for Scalable Computers*. Sous la dir. de David R. O'Hallaron. Berlin, Heidelberg : Springer Berlin Heidelberg, 1998, p. 402-409. isbn : 978-3-540-49530-7 (cf. p. 9).

[100] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007 (cf. p. 6).

[101] Jérôme Richard. "Conception of a software component model with task scheduling for many-core based parallel architecture, application to the Gysela5D code". Theses. Université de Lyon, déc. 2017 (cf. p. 10).

[102] O. Schenk et K. Gärtner. "Solving unsymmetric sparse systems of linear equations with PARDISO". In : *Future Generation Comp. Syst.* 20.3 (2004), p. 475-487 (cf. p. 11).

[103] Marc Sergent. "Scalability of a task-based runtime system for dense linear algebra applications". Theses. Université de Bordeaux, déc. 2016 (cf. p. 20, 139).

[104] D.B. Shmoys, J. Wein et D.P. Williamson. "Scheduling parallel machines on-line". In : *SIAM Journal on Computing* 24.6 (1995), p. 1313-1331 (cf. p. 14).

[105] G. C. Sih et E. A. Lee. "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures". In : *IEEE Transactions on Parallel and Distributed Systems* 4.2 (1993), p. 175-187. issn : 1045-9219 (cf. p. 13).

[106] Marc Snir et al. *MPI: The Complete Reference*. Cambridge, MA, USA : MIT Press, 1995. isbn : 0262691841 (cf. p. 10).

[107] F. Song, A. YarKhan et J. Dongarra. "Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems". In : *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC'09*. 2009 (cf. p. 8).

[108] Fengguang Song, Stanimire Tomov et Jack Dongarra. "Enabling and Scaling Matrix Computations on Heterogeneous Multi-core and multi-GPU Systems". In : *Proceedings of ICS'12*. San Servolo Island, Venice, Italy, 2012, p. 365-376. isbn : 978-1-4503-1316-2 (cf. p. 26).

[109] Hongyang Sun et al. *Scheduling Parallel Tasks under Multiple Resources: List Scheduling vs. Pack Scheduling*. Research Report RR-9140. Inria Bordeaux Sud-Ouest, jan. 2018 (cf. p. 136).

[110] S. Tomov, J. Dongarra et M. Baboulin. "Towards dense linear algebra for hybrid GPU accelerated manycore systems". In : *Parallel Computing* 36.5-6 (2010), p. 232-240 (cf. p. 12).

[111] S. Tomov et al. "Dense linear algebra solvers for multicore with GPU accelerators". In : *IPDPS Workshops*. 2010, p. 1-8 (cf. p. 12).

[112] Stanimire Tomov, Jack Dongarra et Marc Baboulin. "Towards dense linear algebra for hybrid GPU accelerated manycore systems". In : *Parallel Computing* 36.5-6 (juin 2010), p. 232-240. issn : 0167-8191 (cf. p. 12).

[113] H. Topcuoglu, S. Hariri et Min-You Wu. “Performance-effective and low-complexity task scheduling for heterogeneous computing”. In : *IEEE Transactions on Parallel and Distributed Systems* 13.3 (2002), p. 260-274. ISSN : 1045-9219 (cf. p. 13).

[114] John Turek, Joel L. Wolf et Philip S. Yu. “Approximate Algorithms Scheduling Parallelizable Tasks”. In : *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA ’92. San Diego, California, USA : ACM, 1992, p. 323-332. ISBN : 0-89791-483-X (cf. p. 13).

[115] F. Van Zee et al. “Introducing: The Libflame Library for Dense Matrix Computations”. In : *Computing in Science Engineering* PP.99 (2009), p. 1. ISSN : 1521-9615 (cf. p. 11).

[116] V. Volkov et J. Demmel. “Benchmarking GPUs to tune dense linear algebra”. In : *SC08*. 2008, p. 31 (cf. p. 12).

[117] Min-You Wu et Daniel D. Gajski. “Hypertool: A Programming Aid For Message-Passing Systems”. In : *IEEE Trans. on Parallel and Distributed Systems* 1 (1990), p. 330-343 (cf. p. 13).

[118] W. Wu et al. “Hierarchical DAG scheduling for Hybrid Distributed Systems”. In : *29th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. Hyderabad, India, 2015 (cf. p. 26, 27, 65).

Publications personnelles

Conférences et Workshops avec actes

- Olivier Beaumont et al. "Scheduling of Linear Algebra Kernels on Multiple Heterogeneous Resources". In : *International Conference on High Performance Computing, Data, and Analytics (HiPC 2016)*. Proceedings of the IEEE International Conference on High Performance Computing (HiPC 2016). Hyderabad, India : IEEE, déc. 2016.
- T. Cojean et al. "Resource Aggregation for Task-Based Cholesky Factorization on Top of Heterogeneous Machines". In : *Euro-Par 2016: Parallel Processing Workshops: Euro-Par 2016 International Workshops, Grenoble, France, August 24-26, 2016, Revised Selected Papers*. Sous la dir. de Frédéric Desprez et al. Cham : Springer International Publishing, 2017, p. 56-68. isbn : 978-3-319-58943-5.

Conférences et Workshops sans actes

- Terry Cojean. *The StarPU Runtime System at Exascale ?* RESPA workshop at SC16. Salt Lake City, Utah, United States, nov. 2016.
- T. Cojean. *Exploiting Two-Level Parallelism by Aggregating Computing Resources in Task-Based Applications Over Accelerator-Based Machines*. SIAM Conference on Parallel Processing for Scientific Computing (SIAM PP 2016). Paris, France, avr. 2016.

Journal soumis en cours de révision

- Terry Cojean et al. "Resource aggregation for task-based Cholesky Factorization on top of modern architectures". Submitted for review to the Parallel Computing Special Issue for HCW and HeteroPar 16 workshops. Nov. 2016.