Etude et amélioration de l'exploitation des architectures NUMA à travers des supports exécutifs

Philippe Virouleau

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Philippe Virouleau

Thèse dirigée par **Fabrice Rastello**
coencadrée par **Thierry Gautier** et **François Broquedis**

préparée au sein du **Laboratoire d'Informatique de Grenoble,**
dans **l'Ecole Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Étude et amélioration de l'exploitation des architectures NUMA à travers des supports exécutifs

Thèse soutenue publiquement le **5 Juin 2018**,
devant le jury composé de :

Emmanuel Jeannot
Directeur de Recherche Inria, Président, Rapporteur
Julien Langou
Professeur, Rapporteur
Karine Heydemann
Maître de Conférences, Examinatrice
Fabrice Rastello
Directeur de Recherche Inria, Directeur de thèse
François Broquedis
Maître de Conférences, Co-Encadrant de thèse
Thierry Gautier
Chargé de Recherche Inria, Co-Encadrant de thèse

Étude et amélioration de l'exploitation des architectures NUMA à travers des supports exécutifs

Résumé

L'évolution du calcul haute performance est aujourd'hui dirigée par les besoins des applications de simulation numérique. Ces applications sont exécutées sur des supercalculateurs qui peuvent proposer plusieurs milliers de cœurs, et qui sont découpés en un très grand nombre de nœuds de calcul ayant eux un nombre de cœurs beaucoup plus faible. Chacun de ces nœuds de calcul repose sur une architecture à mémoire partagée, dont la mémoire est découpée en plusieurs blocs physiques différents : cela implique un temps d'accès dépendant à la fois de la donnée accédée ainsi que du processeur y accédant. On appelle ces architectures NUMA (pour *Non Uniform Memory Access*).

La manière actuelle de les exploiter tend vers l'utilisation d'un modèle de programmation à base de tâches, qui permet de traiter des programmes irréguliers au dela du simple parallélisme de boucle. L'exploitation efficace des machines NUMA est critique pour l'amélioration globale des performances des supercalculateurs. Cette thèse a été axée sur l'amélioration des techniques usuelles pour leur exploitation : elle propose une réponse au compromis qu'il faut faire entre localité des données et équilibrage de charge, qui sont deux points critiques dans l'ordonnancement d'applications. Les contributions de cette thèse peuvent se découper en deux parties : une partie dédiée à fournir au programmeur les moyens de comprendre, analyser, et mieux spécifier le comportement des parties critiques de son application, et une autre partie dédiée à différentes améliorations du support exécutif. Cette seconde partie a été évaluée sur différentes applications, ce qui a permis de montrer des gains de performances significatifs.

Mots-clés : NUMA, support exécutif, OpenMP, tâches, caractérisation, simulation

Financement

Ce travail a été réalisé dans le cadre du projet ELCI, un projet collaboratif Français financé par le FSN ("Fond pour la Société Numériqe"), qui associe des partenaires académiques et industriels pour concevoir et produire un environnement logiciel pour le calcul intensif.

Studying and improving the use of NUMA architectures through runtime systems

Abstract

Nowadays the evolution of High Performance Computing follows the needs of numerical simulations. These applications are executed on supercomputers which can offer several thousands of cores, split into a large number of computing nodes, which possess a relatively low number of cores. Each of these nodes consists of a shared memory architecture in which the memory is physically split into several distinct blocks: this implies that the memory access time depends both on which data is accessed, and on which core tries to access it. These architectures are named NUMA (for *Non Uniform Memory Access*).

The current way to exploit them tends to be through a tasks-based programming model, which can handle irregular applications beyond a simple loop-based parallelism. Efficient use of NUMA architectures is critical for the overall performance improvements of supercomputers. This thesis focuses on improving common techniques for the exploitation of these architectures: it proposes an answer to the tradeoff that has to be made between data locality and load balancing, that are two critical aspects of applications scheduling. Contributions of this thesis can be split into two parts: the first part is dedicated to providing the programmer with means to understand, analyze, and better characterize the behavior of their applications' critical parts, and the second part is dedicated to several improvements made to the runtime systems. This last part has been evaluated on various applications and has shown some significant performance gains.

Keywords: NUMA, runtime systems, OpenMP, tasks, characterization, simulation

« If we knew what it was we were doing, it would not be called research, would it? »

Albert Einstein

Remerciements

Je souhaite dans un premier temps remercier les deux personnes qui m'accompagnent depuis que j'ai démarré ma vie scientifique il y a 5 ans : Thierry Gautier et François Broquedis. Merci de m'avoir fait apprécier la recherche et de m'avoir trouvé un sujet sur mesure, de m'avoir fait confiance, et d'avoir su m'encourager. Malgré les changements d'équipes vous avez été assidus dans mon encadrement, aussi assidus que pour faire des enfants, puisqu'à vous deux vous en avez fait plus que le nombre d'années qu'il m'a fallu pour faire cette thèse ! Je ne me serais pas engagé dans cette thèse si cela avait été avec d'autres encadrants.

Je souhaite également remercier Fabrice Rastello de m'avoir accueilli dans l'équipe CORSE, et d'avoir pris la direction de ma thèse à ce moment là. Tu as indéniablement apporté plein d'idées à cette thèse, et plein de compétences que je n'aurais pas pu avoir dans un autre contexte.

J'adresse des chaleureux remerciements à Emmanuel Jeannot et Julien Langou pour avoir accepté de rapporter mon manuscrit, ainsi qu'à Karine Heydemann pour avoir accepté de prendre part à mon jury en tant qu'examinatrice.

Merci à tous mes collègues des équipes MESCAL, MOAIS, NANO-D, et CORSE d'avoir partagé ces années de thèse avec moi. En particulier à Annie et Imma d'avoir survécu aux démarches administratives ; à Mathias, Marie, et Pierrick mes co-bureaux de Montbonnot : malgré vos efforts pour m'avertir des «dangers» de la thèse je ne vous ai pas écoutés ;)

ευχαριστώ George pour m'avoir donné des cours de Grec en plus de tolérer mes nombreux goodies, **Fabian** & François pour avoir hébergé de nombreuses soirées très agréables, et bien sûr Raphaël pour m'avoir permi de ne pas être à la rue pendant les 6 derniers mois de ma thèse ! Je n'oublie pas non plus tous les autres avec qui on a pu partager de nombreuses bières et pizzas à Grenoble.

Parce que 3 ans de thèse c'est aussi 52 compétitions de speedcubing dans 12 pays, je voudrais remercier Zecho, Zippo, Lina, Rio, Po, l'homme le plus intelligent de France [TF1 2017], et tous les autres cubeurs et cubeuses en France ou à l'international que j'oublie : vous avez activement participé au maintien de ma bonne santé mentale ! Un énorme merci en particulier à celle et ceux qui ont relu mon manuscrit et ont permis de retirer l'aléatoire sur la présence des 's' (ou pas).

J'ai aussi de gros remerciements à faire aux personnes avec qui j'ai pu partager de nombreuses soirées dans la joie et la bonne humeur au Fam's (avec ou sans coinche) : Mazette, Noémie, Archnouff, Arnaud, Do et Tout Rouge, entre autres.

Merci Fred et Mouchoum d'avoir été là (avant de m'abandonner lâchement :p) et de m'avoir fait découvrir toutes les joyeusetés qu'il est possible de faire à partir de vermicelles arc-en-ciel.

Un énorme merci à Testi (pour tout en fait, tout lister ça serait bien trop long), on ne l'aura pas gagné ce tournoi mais les deux bonnes nouvelles apportées cette dernière année compensent largement !

Je tiens à remercier tout particulièrement ma famille ; d'une part pour avoir affronté les grèves de la SNCF pour venir me voir parler pendant 45 minutes d'un sujet incompréhensible, mais surtout pour m'avoir supporté dans mon parcours et avoir été présents quand il le fallait.

Enfin il y en a une qui a du passer ces quelques paragraphes à chercher son nom, mais je l'ai gardée pour la fin : merci Manou ! Merci d'avoir été là pendant ces 3 ans, merci de ne pas être (trop) désespérée quand je m'enfile un pot d'Häagen-Dazs devant la télé, quand je décide d'être productif entre 22h et 1h, ou encore quand je propose des menus bien trop gras pour être raisonnables. Je suis déjà un peu fou de base, je suis bien content d'avoir trouvé un peu plus de folie ailleurs pour stabiliser tout ça. J'ai vraiment bien fait d'arriver à l'arrache totale dans un pays dont je ne parle pas la langue et de squatter le taxi d'une inconnue sans dépenser un centime. Vive les conférences au Brésil ! ;)

Sommaire

« Elle sert à quoi ta thèse ? »

1 Introduction

L'évolution du calcul haute performance est aujourd'hui dirigée par les besoins croissant en calcul des applications de simulation numérique. Ces applications sont omniprésentes dans l'industrie, et concernent parfois même directement le grand public.

Par exemple des secteurs comme l'aéronautique, les applications militaires, ou encore le nucléaire ont besoin de simuler des phénomènes à grande échelle, se traduisant souvent par la résolution de systèmes linéaires à plusieurs millions d'inconnues. Les prévisions météorologiques à destination du grand public sont faites à l'aide d'applications simulant les interactions entre les différents éléments de l'atmosphère. Il en va de même pour tout ce qui se rapporte à l'étude de la propagation des ondes sismiques dans le sol, ou de la prévision de l'impact d'un séisme sur un bassin de population, où le problème se traduit également par la résolution de systèmes linéaires.

Toutes ces simulations sont au final exécutées sur des supercalculateurs, et il n'y a pas de limite, a priori, au nombre de ressources qu'elles peuvent utiliser : que ce soit pour améliorer la précision de la simulation, ou augmenter la taille de l'ensemble simulé, elles pourront toujours bénéficier d'un plus grand nombre de ressources. Avoir plus de ressources peut également permettre d'exécuter un plus grand nombre de simulations simultanément, voire de les coupler entre elles.

Si les supercalculateurs peuvent proposer plusieurs milliers de cœurs, ils sont en fait composés d'un grand nombre de nœuds de calcul avec un nombre de cœurs beaucoup plus faible. Ces machines peuvent proposer, en plus de processeurs traditionnels, des accélérateurs plus ou moins spécifiques comme des GPUs ou des FPGA,

formant une architecture dite *hétérogène*. La très grande majorité des nœuds de calcul intègrent plusieurs processeurs qui accèdent à une mémoire commune.

Contrairement aux processeurs du siècle dernier pour lesquels un changement de génération s'accompagnait d'une augmentation de leur fréquence de fonctionnement, l'évolution des processeurs contemporains se traduit aujourd'hui par la multiplication du nombres de cœurs de calcul qu'ils embarquent. Pour illustrer ce phénomène il suffit de regarder par exemple la gamme de produits proposés par Intel : la première génération de Pentium 4 - Willamette - lancée par Intel en 2000 était constituée d'un unique cœur cadencé à 1.5GHz. 6 ans plus tard, la dernière génération de Pentium 4 - Cedar Mill - était également consituée d'un seul cœur, mais cette fois cadencé à 3.6 GHz. 10 ans plus tard en 2016, les processeurs de la génération Skylake d'Intel i7 ne dépassent pas les 3.4GHz de fréquence, mais tous ont 4 cœurs physiques au lieu d'un seul.

Avec ce changement de design, les modalités d'accès à la mémoire ont été repensées pour éviter les goulots d'étranglement se formant lors des accès concurrents de plusieurs cœurs au même bus mémoire.

Pour éviter trop de contention sur le bus mémoire, la mémoire est divisée en plusieurs bancs physiques différents, avec chacun leur contrôleur. Sur chacun de ces bancs, les processeurs disposent d'un cache partagé commun en plus des caches privés à chaque processeur. La conséquence directe de ce changement est que le temps d'accès à la mémoire est devenu non uniforme : il dépend directement de quel processeur essaye d'accéder à quelle partie de la mémoire. On appelle ces architectures NUMA (pour *Non Uniform Memory Access*) et elles sont aujourd'hui la brique de base pour créer des supercalculateurs.

Plusieurs modèles de programmation permettent de cibler ce genre d'architectures. Les boucles parallèles et les tâches avec dépendances sont deux types de constructions très utilisées, et elles sont présentes dans la majorité des modèles de programmation.

Les boucles parallèles sont particulièrement adaptées aux applications régulières où les temps d'exécution de chaque itération est facilement prévisible. Les tâches avec dépendances permettent d'exprimer un parallélisme à grain fin, et sont particulièrement adaptées dans le cas d'applications dont certains calculs peuvent être imprévisibles et entrainer un déséquilibre de charge.

Cela peut être expliqué par la manière donc sont ordonnancées ces deux types d'applications par le support exécutif : dans le cas de boucle parallèle, les itérations à exécuter sont généralement découpées équitablement entre les cœurs de calcul. Dans le cas des tâches avec dépendances, l'une des techniques les plus communes est l'ordonnancement par vol de travail : l'application est exprimée comme un graphe de flot de données, chaque sous-partie - tâche - consommant et produisant des données. À chaque fois qu'un processeur devient inactif, il va récupérer — *voler* — une tâche disponible pour l'exécuter. Pour minimiser les périodes d'inactivités des processeurs, il faudrait donc qu'il y ait toujours des tâches disponibles à être exécuter. Pour que le vol de travail soit efficace, il faut donc pouvoir exprimer un maximum de parallélisme : plus il y a de parallélisme, plus il y a de tâches, mieux on est capable d'équilibrer la charge au cours de l'exécution.

Même dans le cas d'une application où le travail peut sembler régulier et où une parallélisation par boucles parallèles semblerait convenir, le côté NUMA des architec-

tures ajoute une variabilité dans le temps des accès mémoires, ce qui rend le temps d'exécution difficile à prévoir et peut entrainer du même coup un déséquilibre de charge. L'utilisation des tâches avec dépendances semblent donc tout à fait appropriée lorsque l'on cible des architectures NUMA.

OpenMP est le standard *de-facto* pour les architectures à mémoire partagée, qui a récemment évolué pour supporter les tâches avec dépendance en plus des boucles parallèles. C'est le modèle que nous avons utilisé comme support pour nos idées, sachant qu'elles pourraient être transposées dans les autres modèles de programmation puisqu'ils utilisent des concepts similaires. Les modèles de programmation existant présentent un manque lorsqu'il s'agit d'exploiter efficacement les machines NUMA. Le programmeur doit faire de gros efforts pour effectuer des optimisations spécifiques peu portables, par exemple via des bibliothèques externes pour contrôler précisément le placement des données. Les outils standards et non intrusifs permettent simplement de distribuer les pages de la mémoire sur les différentes parties physiques : ce n'est pas suffisant, cela permet de diviser et répartir les données et donc de diminuer la contention sur les bus mémoire, mais cela ne permet malheureusement pas aux données dépassant la taille d'une page de rester groupées proches les une des autres, ce qui est un point clé pour avoir une bonne localité des données. La localité des données est essentielle pour les performances, mais lorsqu'il faut équilibrer la charge et donc déplacer des tâches, elle est difficile (voire impossible) à conserver. La bonne exploitation des machines NUMA repose donc sur deux objectifs incompatibles : l'équilibrage de charge et la localité des données.

Cette thèse est axée sur l'amélioration des standards et techniques pour l'exploitation des machines NUMA, et cela passe par plusieurs étapes : tout d'abord fournir au programmeur les moyens de comprendre et analyser le comportement des parties critiques de son application. Ensuite lui permettre de fournir plus d'information au support exécutif, principalement en lui permettant d'exprimer une *affinité* entre ses tâches et les ressources de la machine. Et enfin proposer des techniques d'ordonnancement prenant en compte ces informations, dans le but d'améliorer efficacement les performances globales de l'application.

1.1 Objectifs

L'objectif principal de cette thèse était d'étudier les améliorations possibles de l'exploitation des architectures NUMA, à l'aide d'un modèle de programmation à base de tâches. Cela a été découpé en trois axes de travail.

Analyse du comportement d'applications sur machine NUMA

Avant de pouvoir penser aux améliorations, il faut commencer par analyser les différents points améliorables, tant du côté logiciel que matériel. Si l'on souhaitait cibler les modèles de programmation à base de tâches, il fallait néanmoins choisir l'un des modèles existants pour l'étude concrète, et ce choix s'est porté sur OpenMP.

Les tâches avec dépendances ont été ajoutées peu de temps avant le début de cette thèse dans OpenMP, avec la version 4.0. Face à l'absence de suite de benchmarks ciblant spécifiquement cette construction, nous avons commencé par développer une

suite de benchmarks, les KASTORS [Virouleau 2014]. Les applications présentes dans cette suite ont été adaptées depuis des applications existantes, afin d'utiliser les constructions dont nous avions besoin, et qui sont aujourd'hui utilisées par la communauté.

Dans le but de pouvoir étudier plus précisément le comportement de ces applications sur les architectures NUMA, nous avons écrit un outil, CarToN. Les applications reposent sur des tâches de calculs, dont les bonnes performances sont nécessaires pour la performance globale de l'application. Étudier ces tâches au cours de l'exécution de l'application peut être contraignant : instrumenter le support exécutif, regarder des traces d'exécution ; ce sont des options qui permettent de mettre en évidence un problème. Mais les raisons derrière une différence de performance entre deux tâches similaires ne sont pas forcément évidentes, et rejouer une tâche pour tenter d'isoler le ou les paramètres à la source de cette variation est encore plus dur.

L'objectif derrière cet outil est le suivant : à partir d'un code d'une tâche de calcul à étudier fournit par l'utilisateur et de ses paramètres en entrée, CarToN permet de rejouer ce code en changeant certains paramètres. Concrètement cela peut être par exemple le placement des données de la tâche, son jeu de données en entrée, ou encore le cœur sur lequel elle s'exécute. L'utilisateur peut alors analyser finement quels sont les impacts de chaque paramètre et comment l'architecture sous-jacente réagit.

Quelles améliorations pour l'utilisateur et le support exécutif ?

À partir des conclusions tirées du point précédent, le second objectif était de trouver, proposer, et évaluer une amélioration possible, tant pour l'utilisateur que pour le support exécutif.

Ces réflexions ont donné lieu à une contribution majeure, séparée en deux volets : le premier est axé sur la réponse au besoin de l'utilisateur, en proposant une clause `affinity` pour les tâches OpenMP [Virouleau 2016b]. Cette clause a pour but de permettre à l'utilisateur d'indiquer explicitement un lien fort entre une tâche et une ressource de la machine, que ce soit un cœur, un nœud, ou une donnée. Le second est axé sur l'extension du support exécutif [Virouleau 2016a], d'une part pour faciliter la distribution des données sur la machine, et d'autre part pour exploiter les informations disponibles sur les données manipulées par les tâches, dans le but d'améliorer la localité des données au cours de l'exécution.

Place des travaux dans l'évolution du matériel et du logiciel

Au cours de cette thèse les architectures NUMA ont évolué, on peut alors se demander dans quelle mesure l'évolution du matériel impacte les travaux de cette thèse. Le dernier objectif est donc d'analyser les travaux effectués - voir les compléter - afin de proposer des approches indépendantes du matériel, dans le but de faciliter le travail du programmeur et du développeur de support exécutif à l'avenir.

Parmi nos travaux, CarToN est indépendant de l'architecture. L'approche générale consiste à caractériser les sections de calculs vitales, à l'aide de scénarios d'expérience qui eux sont spécifiques à l'architecture, pour au final obtenir des données qui aideront à déterminer le comportement global de l'application. Pour estimer ce comportement global à partir des données fournies par CarToN, nous avons réalisé des travaux

préliminaires sur un simulateur de support exécutif fonctionnant par vol de travail. Le coût - en temps - de changer l'implémentation utilisée par un support exécutif réel est assez important, ces travaux préliminaires ont pour objectif de donner un premier aperçu de l'impact que pourrait avoir une modification de l'ordonnancement par le support exécutif. Cela permet ainsi, lors du "portage" d'une application ou d'un support exécutif sur une nouvelle architecture, d'estimer son comportement, et évaluer si des changements dans l'un des deux sont nécessaires.

1.2 Organisation du contenu du manuscrit

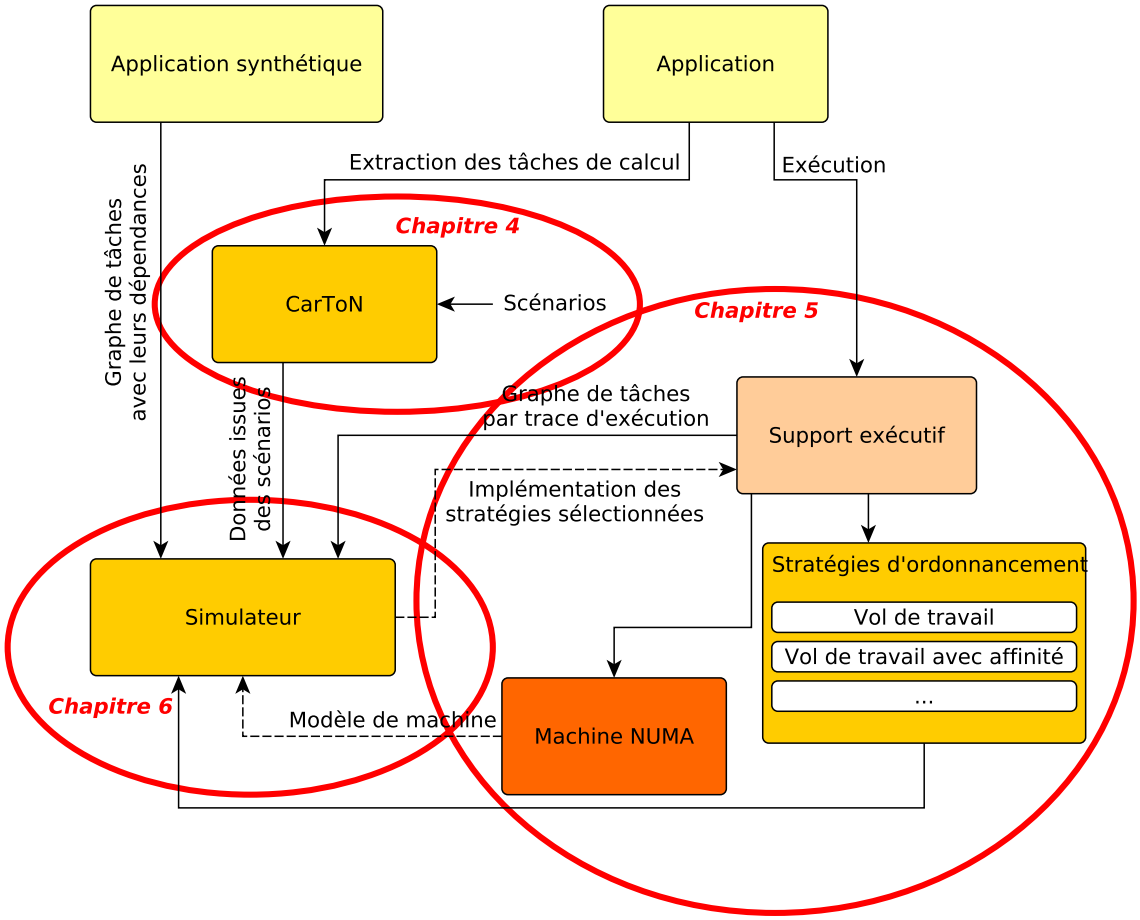

Figure 1.1: Vision schématique des différentes contributions de la thèse

Le manuscrit est découpé en deux grandes parties. La première partie traite des problématiques abordées par cette thèse, ainsi que des approches existantes sur les points techniques abordés. Dans cette partie, le chapitre 2 introduit les éléments de base nécessaires au déroulement de cette thèse : les architectures à mémoire partagée, les moyens existants de les programmer, et une description détaillée de certains outils et concepts techniques fondamentaux. Le chapitre 3 revient sur l'état de l'art des techniques utilisées ou étendues par nos travaux.

La seconde partie regroupe nos travaux sur l'étude des machines NUMA, et l'amélioration de leur utilisation à travers OpenMP. La figure 1.1 illustre les différentes contributions de cette thèse ainsi que leurs interactions. Le chapitre 4 introduit CarToN, un outil que nous avons créé pour étudier en détails le comportement des applications et de l'architecture sous jacente ; nous décrivons également l'orientation des travaux à la suite de nos observations. Les extensions du langage et du support exécutif sont motivées, décrites et évaluées dans le chapitre 5. Le chapitre 6 décrit un prototype de simulateur que nous avons écrit afin d'analyser et d'envisager de nouvelles stratégies d'ordonnancement, à partir notamment des données issues des expériences effectuées avec CarToN.

Enfin le chapitre 7 se concentre sur les perspectives et revient sur l'évolution du matériel et du logiciel pendant la thèse, et discute des pistes de recherche envisageables pour pousser plus loin nos idées, avant de conclure notre travail et ce manuscrit.

Partie I

Problématiques impliquées et approches existantes

« People who are more than casually interested in computers should have at least some idea of what the underlying hardware is like. Otherwise the programs they write will be pretty weird. »

Donald Knuth

2 Contexte

L'objectif de ce chapitre est de donner au lecteur les connaissances de base nécessaires pour l'appréciation du reste de la thèse. Elles peuvent se classer en trois catégories : la première concerne les architectures cibles pour nos travaux. Il y a une grande variété de matériels disponibles pour le calcul haute performance : la section 2.1 décrit en détails les architectures à mémoire partagée actuelles (NUMA), et la section 2.2 décrit leur gestion par le système d'exploitation. La seconde concerne les modèles de programmation à base de tâches. La section 2.3 décrit les techniques modernes utilisées pour cibler les architectures NUMA, et décrit en détails les concepts de base ainsi que les points clés pertinents à nos travaux. La section 2.4 regroupe des informations générales concernant les applications parallèles à base de tâches et le rôle des supports exécutifs dans leur exécution. Enfin la section 2.5 est dédiée à OpenMP, modèle de programmation qui a été utilisé pour l'application des travaux de la thèse.

2.1 Architectures à mémoire partagée

Les architectures à mémoire partagée ont subi des changements majeurs liés à l'évolution des processeurs. L'augmentation du nombre de cœurs par processeur a introduit des problèmes d'accès à la mémoire centrale : dans une architecture composée d'une unique mémoire, plusieurs cœurs cherchant à accéder à la mémoire vont rentrer en concurrence et introduire de la contention et donc des délais dans l'accès à la mémoire, ce qui pénalise fortement les performances.

Pour pallier ce problème, les constructeurs ont divisé la mémoire centrale en plusieurs parties physiquement distinctes, appelées *nœuds*. Le nœud NUMA est le composant de base pour une architecture NUMA. Chaque nœud est constitué d'une partie de la mémoire centrale, d'un contrôleur local d'accès à ce bloc mémoire, ainsi que d'un certain nombre de processeurs multicœurs, eux-mêmes composés de plusieurs cœurs de calcul. L'ensemble des nœuds de la machine sont ensuite reliés entre eux par un réseau d'interconnexion. La topologie de l'interconnexion ne permet généralement pas d'avoir des nœuds équidistants, ce qui introduit une hiérarchie mémoire.

Malgré le fait que les différentes parties de l'architecture soient physiquement séparées, le système d'exploitation voit l'ensemble comme une unique machine.

Nous décrivons dans un premier temps, dans la section 2.1.1, les caractéristiques techniques communes des processeurs multicœur consituant les nœuds NUMA, puis celles des systèmes d'interconnexion des nœuds dans la section 2.1.2. Elles ont une influence directe sur le comportement du nœud au sein de la machine.

2.1.1 À l'intérieur d'un processeur multicœur

Les sections suivantes se concentrent sur certains composants clés des performances d'un processeur multicœur. Les fréquences des cœurs des processeurs sont beaucoup plus élevées que celles des bus d'accès à la mémoire centrale. Cela implique a priori que le temps d'exécution du programme va être limité par le temps d'accès à la mémoire (phénomène introduit sous le nom de *Memory Wall* [Wulf 1995]). L'architecture des caches et leurs caractéristiques sont cruciales pour limiter l'impact de ce phénomène, nous les abordons en détail dans la section 2.1.1.1. La section 2.1.1.2

décrit le parallélisme que les processeurs peuvent exposer pour augmenter les performances, via des instructions ou composants matériels spécifiques.

2.1.1.1 Des caches pour accélérer l'accès aux données

Le cache est une mémoire pour laquelle le temps d'accès est bien meilleur que pour la mémoire centrale, mais dont la capacité est beaucoup plus restreinte. Il peut être spécifique à un cœur du processeur, ou partagé par plusieurs d'entre eux.

Fonctionnement Au niveau des accès, le fonctionnement d'un cache est légèrement différent de celui de la mémoire centrale : plutôt que de pouvoir adresser uniquement un octet, c'est généralement une partie fixe de la mémoire contenant cet octet qui est chargée. On appelle cette quantité une *ligne de cache*, et un exemple de taille standard pour une ligne de cache est 64 octets (8 nombres réels double précision).

Lorsqu'une instruction demande le chargement d'une valeur située en mémoire, le contrôleur du cache reçoit la requête du processeur, détermine la ligne de cache correspondante à partir de l'adresse demandée, et effectue l'une des deux opérations suivantes :

Cache hit : si la ligne correspondante est déjà présente dans le cache, la donnée est directement retournée au cœur.

Cache miss : si la ligne n'est pas présente, le contrôleur de cache va charger la valeur depuis la mémoire centrale ou un autre niveau de cache, et retourner la valeur au cœur.

Une fois qu'une ligne est chargée dans le cache, elle n'y reste pas de manière permanente, plusieurs raisons décrites ci-après peuvent entraîner son *éviction* du cache :

- Le maintien de la cohérence : lorsqu'un cœur modifie une ligne de cache dans son propre cache, cette même ligne est *invalidée* dans les caches des autres processeurs qui disposent d'une version de celle-ci. Si un processeur essaie de faire un accès sur une ligne invalidée, elle sera rechargée depuis la mémoire centrale ou un autre cache en possédant une copie valide, générant un *cache miss*.

- Le dépassement de la capacité du cache : il est assez commun que l'ensemble des données manipulées par le programme ne tienne pas dans le cache. Lorsqu'une requête est effectuée sur une ligne, qu'elle n'est pas présente dans le cache, et que le cache est plein, le contrôleur choisira une ligne à évincer du cache pour faire de la place à la nouvelle ligne. Le choix de la ligne à évincer est un sujet très étudié, et le prochain paragraphe revient sur les politiques d'éviction communément utilisées.

- Le conflit d'adresse : dans la quasi totalité des modèles de caches, certaines lignes correspondant à des adresses mémoires doivent être stockées dans le même emplacement du cache. Si elles sont requises alternativement pendant l'exécution du programme, elles se sortiront mutuellement du cache. Ce problème dépend directement de l'*associativité* du cache, qui est traitée dans le paragraphe suivant.

Associativité Le cache ne peut être suffisamment grand pour contenir toute la mémoire : le temps d'accès est lié à sa taille, et sa taille est généralement négligeable face à la quantité totale de mémoire disponible. Néanmoins chaque partie de la mémoire doit pouvoir être stockée dans une partie du cache, il faut donc pouvoir déterminer l'association entre l'adresse d'une ligne dans la mémoire centrale et son emplacement dans le cache. L'associativité du cache varie entre une association directe (*direct-mapped cache*), où chaque ligne de la mémoire est associée à exactement un emplacement dans le cache ; et une association complète (*fully associative cache*), où chaque ligne de la mémoire peut être associée à n'importe quel emplacement dans le cache.

Hill et al. [Hill 1989] illustrent l'impact de l'associativité (dans la table III de l'article) sur la proportion de *cache miss* et catégorise son origine (conflit d'adresse, défaut de capacité du cache, chargement normal de la donnée). Cela permet de dégager deux observations : d'une part qu'augmenter l'associativité permet de diminuer les *cache miss*. D'autre part que les défauts de cache ayant une forte associativité sont quasi exclusivement dus à la capacité du cache, alors que pour les caches à association directe les conflits d'adresse sont une part non négligeable des *cache miss*.

Niveaux de caches Il y a en général 3 niveaux de caches dans ces architectures, labellisés L1, L2, et L3. La figure 2.1 décrit la hiérarchie typique d'un processeur multicœurs que l'on peut trouver sur un nœud NUMA.

Le L1 est privé au cœur, il est découpé en deux parties : une spécifique aux données, et l'autre aux instructions. C'est le niveau de cache le plus proche du CPU mais aussi le plus petit : seulement 32 Ko dans cet exemple. Le cache L2 est lui aussi privé au cœur, mais propose une plus grande capacité (ici 256 Ko) au prix d'une latence plus importante. Enfin le cache L3, ou cache de dernier niveau - *Last Level Cache (LLC)*, est partagé par tous les cœurs du processeur. La latence pour y accéder est plus importante que pour accéder au L2, mais sa capacité est bien supérieure, atteignant en général plusieurs Mo, 20 dans cet exemple.

En général les développeurs d'applications pour le HPC accordent beaucoup d'attention à l'optimisation de leur application, pour que les parties de code séquentiel critiques utilisent des données qui puissent être contenues dans le L1/L2. De même, beaucoup d'optimisations au sein des compilateurs visent également cet objectif.

Lorsqu'il s'agit de cibler des architectures NUMA, on va tout particulièrement s'intéresser au L3 qui représente la mémoire la plus rapide accessible par tous les cœurs d'un même nœud, et donc faire attention à ce que les données qui sont partagées par plusieurs cœurs puissent être contenues dans ce cache.

Afin de donner un ordre d'idée des latences et bandes passantes sur les différents processeurs, le tableau 2.1 récapitule les chiffres en fonction de la génération du processeur. Nous avons fait ces mesures à l'aide de l'outil LMbench [McVoy 1996]. Pour la latence, les débits considérés sont ceux du benchmark `lat_mem_rd` avec un accès aléatoire à la mémoire. Pour la bande passante les temps considérés sont ceux du benchmark `bw_mem`, en utilisant un équivalent de `memcpy`. La majorité des processeurs dotés d'un cache dispose également d'un composant matériel - le *prefetcher* - dont le but est de charger en avance des données dans le cache en fonction des accès aux données déjà effectués. Pour la latence, tout effet potentiel de ce composant est annulé par l'utilisation de *pointer chasing* (l'adresse de la case suivante à charger est située dans la

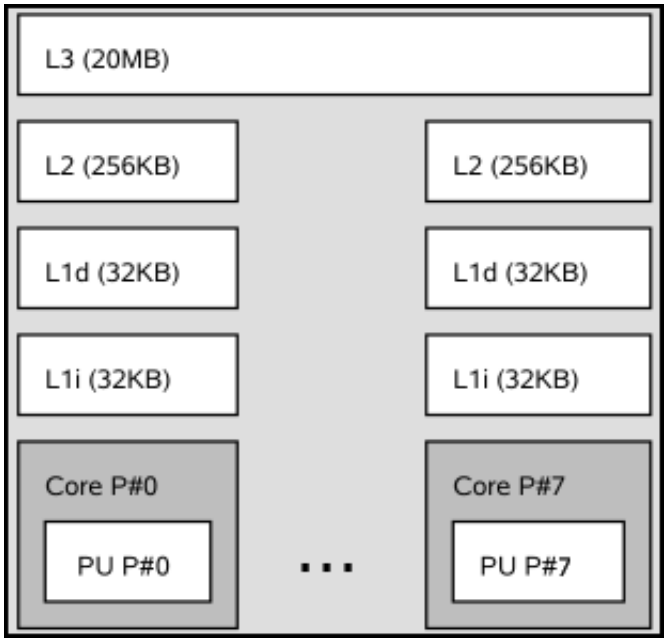

Figure 2.1: Schéma d'un processeur multicœurs

case courante), ce qui n'est pas le cas pour la bande passante.

Comme nous pouvons le voir sur ces chiffres, le coût d'accès à la mémoire principale est bien plus grand que l'accès des différents caches, et l'accès à la mémoire distante est tout simplement prohibitif. Il faut également noter une évolution intéressante au niveau des latences : bien que Broadwell soit une architecture plus récente que Sandy Bridge, les latences aux différents niveaux de caches sont plus élevées, sauf pour l'accès à la mémoire distante où elle a été diminuée de moitié. Les différentes caractéristiques de ces processeurs et des machines sur lesquelles ils sont intégrés seront étudiées en détails dans la section 4.2.

Politiques d'éviction Le choix de la ligne de cache à remplacer lorsque le cache est plein a un impact direct sur les performances, et les politiques de remplacement ont été très étudiées par les différents acteurs de la communauté académique et industrielle. Le choix de la politique dépend de multiple facteurs, tels que la taille du cache, son associativité, ou l'espace matériel disponible pour la réaliser. Al-Zoubi et al. [Al-Zoubi 2004] proposent une évaluation complète et détaillée de l'impact des politiques d'éviction en fonction de l'associativité du cache. Dans cette évaluation, des variations de la politique Pseudo-LRU semblent être les plus avantageuses pour les associativités étudiées. Le coût d'implémentation de la politique *Least Recently Used (LRU)* devient trop important lorsque l'associativité dépasse un certain seuil [Kędzierski 2010] ; Pseudo-LRU offre une alternative où la ligne choisie pour être évincée est une parmi celles utilisées il y a le plus longtemps.

Dans les processeurs Intel, des variantes de la politique Pseudo-LRU semble être les plus utilisées pour les caches de faibles associativités, et donc typiquement les caches de niveaux L1 et L2 [Abel 2014, Intel 2014]. Dans le cas spécifique du cache de dernier niveau (LLC), certains fabricants comme Intel utilisent des politiques adap-

Cache	Intel Sandy Bridge (2012, idchire)		Intel Broadwell (2016, brunch)	
	Latence	Bande passante	Latence	Bande passante
L1	1.6 ns	16 Go/s	1.8 ns	32 Go/s
L2	5.0 ns	14 Go/s	5.4 ns	17 Go/s
L3	20 ns	8 Go/s	24 ns	9 Go/s
RAM	90 ns	4 Go/s	140 ns	7 Go/s
RAM distante	500 ns	1.1 Go/s	230ns	4.2 Go/s

Table 2.1: Tableau synthétique des latences et bandes passantes en fonction du niveau de cache et du processeur

tatives présentées comme supérieures à Pseudo-LRU, prenant en compte la fréquence à laquelle sont utilisées certaines lignes de caches. *Dynamic Re-Referency Interval Prediction* en est un exemple, qui a été présenté parmi d'autres par Jaleel et al. [Jaleel 2010]. D'autres constructeurs tels que ARM font le choix de la simplicité d'implémentation, comme par exemple dans les processeurs ARM Cortex-R [ARM 2010], où le choix de la ligne à évincer est tout simplement aléatoire.

2.1.1.2 Du parallélisme au sein du processeur pour augmenter la puissance de calcul

En plus des caches qui nous intéresseront tout particulièrement dans la suite de ce manuscrit, les processeurs ont également d'autres composants matériels dont il est important d'avoir conscience, mais qui ont une place plus limitée dans le contexte de cette thèse : il s'agit des instructions vectorielles et de l'hyperthreading, décrit ci-après.

Instructions vectorielles Le concept de vectorisation est l'action d'appliquer une même instruction sur plusieurs données (ou un *vecteur* de données) nécessitant la même opération. Ce type d'instructions est appelé SIMD, pour *Single Instruction Multiple Data* [Flynn 1966]. La vectorisation permet au processeur d'optimiser la décomposition de l'instruction en micro opérations et donc l'utilisation du pipeline des différentes Unité Arithmétique et Logique (UAL) [Muller 1989].

Des extensions au jeu d'instructions x86 ont été créées afin de pouvoir opérer sur des éléments plus larges que 64 bits, et la plupart des architectures des processeurs récents utilisent des registres plus larges que le type le plus large en C (`long long`). La première d'entre elles, SSE (*Streaming SIMD Extensions*), a été introduite par Intel dès 1999 et a évolué régulièrement, agrandissant progressivement la taille des registres jusqu'à l'extension AVX-512, permettant d'effectuer des instructions sur des registres de 512 bits.

La plupart des processeurs actuels (depuis les Sandy Bridge d'Intel, et les Bulldozer d'AMD, en 2011) supportent au moins l'extension AVX avec des registres de 128 bits.

Hyperthreading Chaque cœur possède un certain nombre d'UALs qui lui sont privées. Lorsqu'il est en attente d'une donnée de la mémoire centrale, ces UALs ne sont pas utilisées, et des cycles CPU sont donc "perdus" à ne rien faire.

Afin de maximiser l'utilisation de ces ressources, certains processeurs Intel sont équipés de la technologie *hyperthreading*. Le concept est assez simple : avoir deux cœurs logiques (*hyperthreads*) associés à un seul cœur physique. De cette manière lorsqu'un thread est en attente sur une donnée (par exemple lors d'un chargement d'une donnée depuis la mémoire), le second peut éventuellement profiter des UALs disponibles.

Pour des tâches peu gourmandes en ressources ou utilisant beaucoup de données, cela peut effectivement se traduire par un gain de performance, mais dans le cadre du calcul haute performance il faut regarder le type d'applications utilisé pour savoir si on peut espérer un gain ou non. En particulier les caches L1 et L2 sont partagés par les deux hyperthreads, donc si le code séquentiel généré est optimisé pour les tailles de caches correspondant, exécuter le même type de code séquentiel sur deux hyperthreads peut entrainer du *cache trashing*. Au meilleur des cas l'hyperthreading améliorera les performances : Jeffers et al. [Jeffers 2016] rapportent par exemple entre 2.3 et 3.3 de speed-up pour 4 hyperthreads par cœur sur les derniers processeurs d'Intel, les Xeon Phi. Si l'application est très intensive en calcul et utilise au maximum les UALs, l'hyperthreading n'apportera pas grand chose, voire rien.

L'hyperthreading est généralement une option que l'on peut désactiver dans le BIOS de la machine, ou éviter en plaçant correctement les threads de son application.

2.1.2 Passage à l'échelle supérieure : interconnexion des processeurs

L'une des parties majeures d'une machine NUMA est le système d'interconnexion entre les différents nœuds. C'est cette partie qui détermine le coût d'accès à la mémoire située sur un nœud distant, et donc l'influence de l'aspect NUMA de la machine sur les performances d'une application. Dans la majorité des cas, ce système d'interconnexion est *cache-coherent*, c'est à dire que la cohérence de cache est assurée entre les différents nœuds par le matériel, et n'est pas la responsabilité du programmeur ou du support exécutif. L'impact sur les performances peut être lié à la fois au protocole de cohérence de cache utilisé, ainsi qu'à la topologie de l'interconnexion des nœuds.

2.1.2.1 Protocoles de cohérence de cache

Le nombre de caches utilisés dans une machine NUMA peut être important, et la même ligne de la mémoire peut être présente dans plusieurs caches en même temps. Il est important que l'état des différentes copies soit cohérent : par exemple si une ligne a été écrite et modifiée, les copies de cette ligne dans les autres caches doivent être mises à jour.

La plupart des protocoles se basent sur un ensemble d'états possibles pour une ligne de cache :

Modified (M) : la ligne a été modifiée dans le cache. Les données dans cette ligne ne sont donc pas cohérentes avec la mémoire principale. Quand la ligne est évincée, elle doit être écrite dans la mémoire principale.

Shared (S) : la ligne est «propre» (non modifiée), elle existe dans d'autres caches mais est en lecture seule dans le cache courant. Cette ligne peut être évincée sans autre action.

Invalid (I) : la ligne est soit absente du cache courant ou a été invalidée par un autre cache. Elle doit être récupérée depuis la mémoire principale (ou un autre cache qui en possède une copie valide).

Cet ensemble d'états de base forme le protocole *MSI*, qui a ensuite été étendu avec plusieurs autres états :

Exclusive (E) : la ligne est présente uniquement dans le cache courant et elle est «propre».

Owned (O) : la ligne est présente dans plusieurs caches dans un état valide, mais seul le cache courant peut y effectuer des modifications. Cet état permet de partager des lignes qui ont été modifiées sans passer par une ré-écriture dans la mémoire centrale : le cache qui possède la ligne est responsable de fournir une version à jour, et d'écrire la ligne dans la mémoire centrale lorsque que la ligne est évincée.

Forward (F) : cet état est similaire à l'état *Shared*, mais indique au cache courant qu'il est responsable de satisfaire une requête en lecture sur la ligne.

Intel Quick Path Interconnect [Ziakas 2010] (QPI) est le système d'interconnexion utilisé dans les machines d'expérimentation que nous avons utilisées, et qui sont décrites dans la section 4.2. Il implémente le protocole MESIF. Dû à l'introduction de l'état *F*, ce protocole est avantageux lorsque la latence de cache à cache est bien plus faible que la latence d'accès à la mémoire principale. AMD HyperTransport [Keltcher 2003] est le système d'interconnexion utilisé dans les machines à base de processeurs AMD, et implémente le protocole MOESI.

Pour implémenter ces protocoles, il existe deux types de mécanismes :

- À base de Snooping : chaque cache observe le trafic sur le bus mémoire pour des requêtes qui concerneraient les lignes de caches dont il possède une copie. Un composant dédié peut effectuer un premier filtre pour restreindre le trafic lié au mécanisme de snooping. Néanmoins ce type de mécanismes ne passe pas bien à l'échelle, et aurait du mal à obtenir de bonnes performances dans les architectures NUMA, où le nombre de caches est important.

- À base de répertoires : la donnée est placée dans un répertoire qui maintient la cohérence entre les caches. Les caches ne peuvent pas accéder directement à la mémoire principale mais doivent passer par le répertoire.

Le QPI utilise un mix des deux mécanismes : il utilise des *home agents* - comparables à des répertoires - qui sont les principales autorités pour fournir une version du cache depuis la mémoire principale. Il utilise ensuite du *snooping* pour satisfaire les requêtes des différents *caching agents* (les entités qui possèdent un cache, telles que les processeurs).

Le QPI peut fonctionner avec deux modes de *snooping* :

Source snooping : le *caching agent* envoie la requête à tout le système. Les autres *caching agents* peuvent répondre à la requête s'ils possèdent une version de la ligne de cache dans un état compatible. Le *home agent* responsable de la ligne mémoire doit fournir une copie propre de la ligne de cache si besoin, et résoudre les conflits s'ils apparaissent.

Home snooping : le *caching agent* envoie la requête au *home agent*, qui envoie ensuite une requête aux *caching agents* qui possèdent une copie de la ligne et peut commencer à charger la ligne depuis la mémoire. L'un des *caching agents* possédant la ligne et/ou le *home agent* peut ensuite envoyer la réponse.

Le mode *source snooping* a une latence plus faible que le mode *home snooping*, mais passe moins bien à l'échelle. C'est donc ce dernier qui est le plus adapté à un environnement NUMA, puisqu'il limite le nombre de requêtes envoyées sur le bus mémoire, et économise donc de la bande passante.

2.1.2.2 Topologies

La topologie du système d'interconnexion peut être très différente d'une machine à une autre, et de multiples exemples existent dans les machines commercialisées. L'objectif de l'interconnexion est de proposer à un maximum de nœuds d'être en lien les uns avec les autres, tout en minimisant la «distance» entre eux. Il existe des topologies plates, où chaque nœud est directement connecté aux autres, comme c'est le cas pour la machine *brunch* décrite en détails dans la section 4.2.2, mais également des topologies plus compliquées, où par exemple des couples de nœuds sont groupés entre eux et peuvent passer par un ou deux niveaux d'interconnexion, comme c'est le cas pour la machine *idchire*, décrite en détail dans la section 4.2.1.

Pour illustrer la complexité et l'impact que peut avoir la topologie, prenons l'exemple de la machine *idchire* dont une partie de la topologie est illustrée dans la figure 2.2. La figure montre 2 nœuds NUMA sur un même groupe, la machine possède en tout 12 groupes (et donc 24 nœuds) interconnectés.

Physiquement accéder aux nœuds 1, 2, et 10 depuis le nœud 0 n'est pas équivalent, et les données passent par des liaisons physiques différentes. Le rapport entre le coût d'accès à un nœud distant et le coût d'accès au nœud local s'appelle le *facteur NUMA*. Le nombre de rebonds - *hops* - à effectuer avant d'accéder à la mémoire demandée impacte directement la latence et la bande passante.

Une carte complète - *heatmap* - des distances est montrée dans le chapitre 4, Figure 4.2. Le tableau 2.2 donne un comparatif avec des chiffres indicatifs. Nous pouvons constater que sur *idchire* les bandes passantes distantes sont relativement similaires comparativement à la bande locale, qui est presque trois fois plus importante. Sur une architecture plus récente, *brunch*, la différence de bande passante est bien moindre mais toujours présente. Ces chiffres motivent fortement l'intérêt à porter à la localité des données.

Cette partie s'est concentrée sur les connaissances de base nécessaires pour comprendre l'interaction entre les différents composants des architectures NUMA. Le développeur d'application ne va généralement pas influer directement sur ces composants, en revanche il est capital d'avoir conscience des caractéristiques de cha-

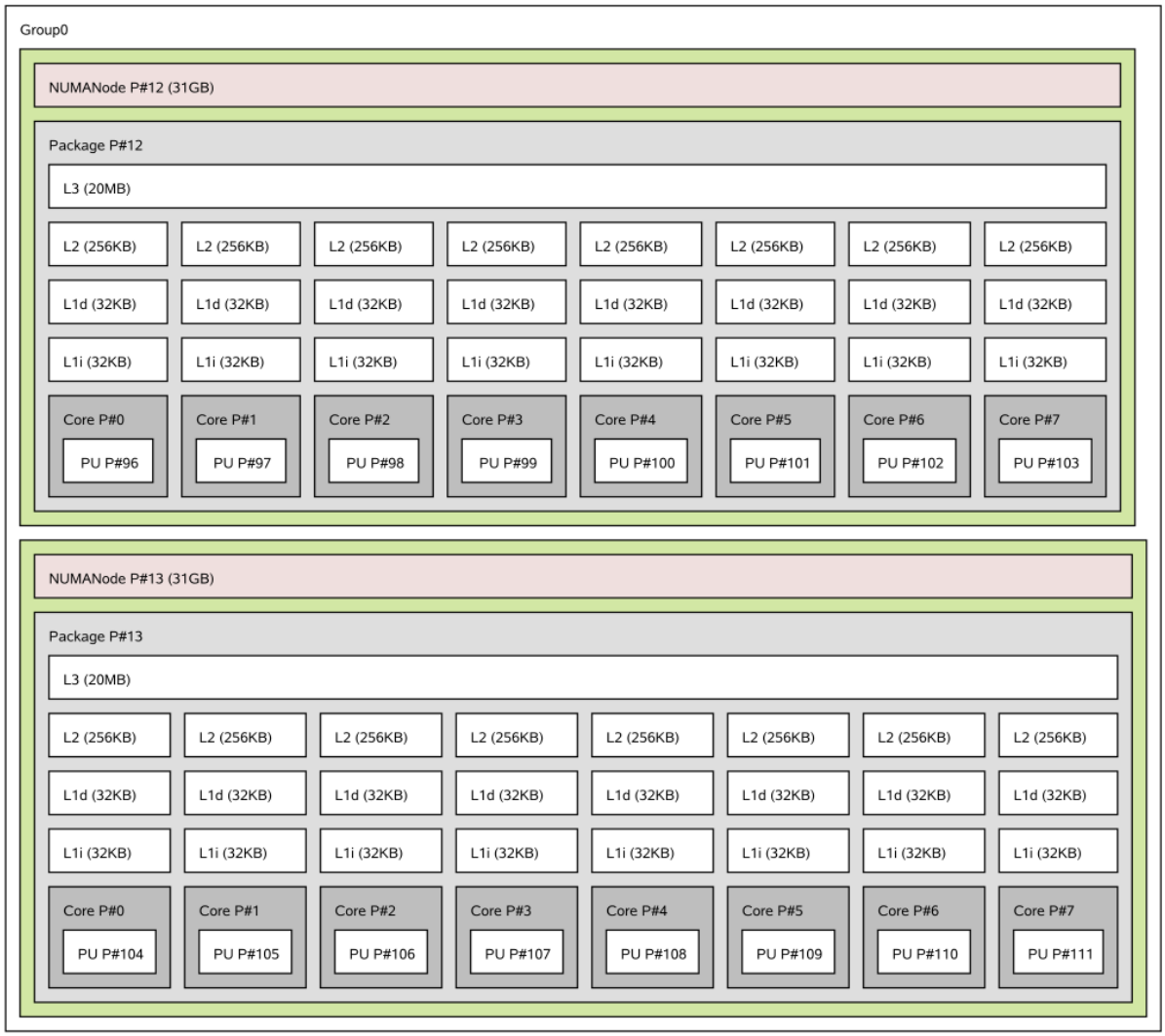

Figure 2.2: Schéma d'un groupe de deux nœuds sur *idchire*

cun d'entre eux pour pouvoir expliquer facilement tel ou tel comportement de l'application.

Le premier composant logiciel qui va s'intéresser à la gestion directe du matériel est le système d'exploitation. La section suivante revient sur les points relatifs à la gestion des architectures NUMA dans le système d'exploitation.

2.2 Exploitation des architectures NUMA par le système d'exploitation

Le support des machines NUMA dans Linux est arrivé dès 2003 [Dobson 2003]. Détailler les dispositifs spécifiques à l'exploitation des machines NUMA au niveau système ne peut pas se faire sans avoir connaissance de certains mécanismes déjà existants. La section suivante détaille donc la manière dont Linux gère la mémoire d'une machine «standard», puis nous détaillerons les spécificités liées au caractère

Destination	idchire	brunch
Local	3.1 Go/s	3.6 Go/s
Même groupe	1.27 Go/s	NA
1 hop	1.12 Go/s	3.2 Go/s
2 hops	1.0 Go/s	NA

Table 2.2: Bande passante (*memcpy* de 200 Mo) en fonction du nœud destinataire de la copie

NUMA de la machine, ainsi que certaines bibliothèques externes utiles pour les programmeurs.

2.2.1 Gestion de la mémoire

La compréhension de la gestion de l'allocation de la mémoire par le système d'exploitation est un point critique lorsque l'on traite les machines NUMA. Le système d'exploitation gère une mémoire *paginée*. Lors de l'allocation d'un tableau par exemple, si sa taille dépasse celle d'une *page*, l'espace mémoire alloué consistera donc de plusieurs pages. Lorsqu'un allocateur utilise une primitive d'allocation de pages telle que *mmap*, elles sont allouées selon une politique d'allocation physique des pages. La politique par défaut dans Linux est d'allouer physiquement la page lors du premier accès - *first-touch*, sur le nœud NUMA du cœur effectuant ce premier accès. Dans ces conditions, le contrôle de l'allocation physique n'est possible que lors de la première allocation des *pages*. Cette contrainte est limitée à la durée de vie du programme, mais il est aussi possible d'utiliser des librairies externes pour gérer plus finement l'allocation de données, comme décrit dans la section suivante.

Enfin, un dernier détail mérite de l'attention, il s'agit de la taille des *pages* manipulées par le système. En effet celle-ci est configurable, et deux options sont généralement possibles : soit des *pages* de 4 Kilo octets, soit des - *huge* - *pages* de 2 Mega octets.

2.2.2 Prise en compte des architectures NUMA et bibliothèques externes

Avec l'arrivée du support des machines NUMA dans le système d'exploitation Linux, plusieurs fonctionnalités fondamentales sont apparues et parmi elles on retrouve : la capacité d'obtenir des informations à propos de la machine, telles que le numéro de nœud d'un cœur donné, ou le numéro de nœud d'une adresse mémoire donnée. On trouve aussi la possibilité de contrôler le placement d'un thread sur un ensemble de cœurs physiques à l'aide d'un masque d'affinité, ce qui permet par exemple de garantir la proximité physique de deux threads vis à vis d'un nœud, ou tout simplement de garantir qu'un thread sera exécuté par un cœur précis.

Plusieurs outils se sont basés sur ces briques de base pour proposer des fonctionnalités plus avancées, comme par exemple :

numactl [Kleen 2004] : cet outil s'utilise avant l'utilisation d'un exécutable sur la ligne de commande. Il permet : de contrôler l'ensemble des cœurs (ou nœuds) sur lesquels les threads peuvent être placés au cours de l'exécution du programme ; de contrôler l'ensemble des nœuds NUMA sur lesquels peuvent être allouées des données ; et encore de modifier la politique d'allocation des *pages* du système, en permettant par exemple de distribuer les *pages* successives allouées de manière cyclique sur un ensemble de nœuds.

hwloc [Broquedis 2010b] : cet outil permet, entre autres, d'obtenir des informations précises sur le matériel et la topologie de la machine. Il expose les facteurs NUMA théoriques fournis par le matériel, et permet de visualiser les différents nœuds et groupes de nœuds. Il propose également des fonctions d'allocation plus fiables que `malloc`, garantissant l'allocation de nouvelles *pages*, et proposant différentes politiques d'allocation (incluant le *first-touch*).

MaMi [Broquedis 2009] : cet allocateur est implémenté au sein de ForestGOMP, décrit en détails dans la section 3.1.3. Il permet l'allocation et la migration de *pages*, soit explicitement via des appels dédiés, soit implicitement en marquant des pages à migrer au *next-touch* : elles seront alors migrées sur le nœud du prochain cœur qui y accède.

Dans la section précédente nous avons vu les détails concernant le matériel utilisé au cours de cette thèse, et dans cette section nous avons vu comment l'utiliser au niveau du système d'exploitation.

Le développeur d'applications scientifiques n'est généralement pas un spécialiste des fonctionnalités de base du système d'exploitation : il va plutôt être un expert de son application et des parties critiques qui la composent. Il faut donc une couche par dessus le système d'exploitation pour lui permettre d'exprimer de manière plus abstraite son application. Cette couche est le *modèle de programmation*. Pour cibler les architectures à mémoire partagée, les modèles de programmation à base de tâches offrent des propriétés très intéressantes : ils permettent d'exprimer du parallélisme à grain fin, et l'équilibrage de charge est effectué de manière dynamique par le système exécutif. Cela permet potentiellement de maximiser l'utilisation du nombre important de ressources disponibles sur une machine NUMA. L'expression d'un programme à base de tâches permet également d'isoler les parties critiques du programme, ce qui peut faciliter leur analyse.

2.3 Modèles de programmation à base de tâches

Il existe de nombreux modèles de programmation à base de tâches, et dans le cadre de cette thèse nous nous sommes restreints à appliquer nos idées dans OpenMP. Les concepts de base sont les mêmes que dans les autres modèles de programmation, et sont détaillés ci dessous.

2.3.1 L'unité de base : la tâche

Une tâche peut être vue comme la plus petite quantité de travail séquentiel exécutable sur un processeur. En pratique c'est une section de code bien définie du programme, et cela peut être une simple instruction, un bloc de code délimité, ou encore une fonction très complexe. La quantité de calcul idéale dans une tâche - la *granularité* - peut varier fortement en fonction de l'application et du support exécutif, ce point est abordé en détail dans la section 2.3.5

Une tâche est nécessairement accompagnée de données qu'elle manipule. De la même manière qu'une fonction utilise des paramètres, le bloc de code composant une tâche utilise des variables qui peuvent être soit locales (on parlera alors de données privées), soit partagées par d'autres parties du code (on parlera de données partagées).

2.3.2 Traitement d'une tâche : de la création à l'exécution

Si la notion de tâche peut paraître simple, elle s'accompagne d'un certain nombre de traitements plus ou moins automatiques (en fonction du modèle de programmation), mais qui dans tous les cas ont un coût. Les trois paragraphes suivants décrivent quelques points clés accompagnant l'utilisation des tâches, qui sont parfois cachés au programmeur.

2.3.2.1 Création

En fonction du modèle de programmation, la création d'une tâche peut être plus ou moins complexe pour le programmeur. L'exemple simple ci-dessous illustre la création d'une tâche en OpenMP :

```
1  void foo()
2  {
3    // ...
4    int a = 1;
5    #pragma omp task shared(a)
6    {
7      // calcul sur a
8    }
9    // ...
10 }
```

Certains modèles de programmation nécessitent que l'utilisateur écrive une fonction suivant un prototype particulier, qui sera le point d'entrée passé à une routine spécifique du support exécutif pour effectivement créer la tâche.

Dans le cas spécifique d'OpenMP, ce travail est effectué par le compilateur via deux transformations :

- l'*outlining* de la fonction, qui consiste à externaliser le code de la tâche et son contexte dans une fonction séparée.
- la substitution du pragma par un appel au support exécutif.

Cet ensemble d'appels au support exécutif n'est présent que dans le binaire, et s'appelle l'*ABI* (pour *Abstract Binary Interface*).

Au final le code correspondant qui est généré dans le binaire est plus complexe qu'il n'y parait, et en l'exprimant dans l'ABI de Clang il serait équivalent à un programme de ce type :

```
struct unamed_struct_1 {
  int *a;
}

// Outlining du bloc de code correspondant à la tâche
void omp_task_entry(void *args)
{
  struct unamed_struct_1 *shared_variables =
                (struct unamed_struct_1 *) args;
  int a = *(shared_variables->a);
  // calcul sur a
}

void foo()
{
  // ...
  int a = 1;
  // Substitution du pragma par une allocation
  // puis une création de la tâche
  struct unamed_struct_1 tmp;
  tmp.a = &a;
  kmp_task_t task_1 = __kmpc_omp_task_alloc(omp_task_entry, &tmp,
     /* ... */);

  __kmpc_omp_task(&task_1);
  // ...
}
```

Ce morceau de code est similaire à ce qu'il est nécessaire d'écrire dans certains modèle de programmation, comme on le verra dans la section 2.3.4.

2.3.2.2 Gestion

Une fois que le programmeur a défini sa tâche et l'a soumise au support exécutif, celui-ci doit créer et maintenir une structure de données représentant cette tâche. Celle-ci peut être plus ou moins grande en fonction des informations associées à la tâche.

Le support exécutif va utiliser ces informations au cours de l'exécution du programme pour déterminer quelles tâches sont prêtes pour l'exécution. En pratique cela signifie que ces structures de données vont être placées dans des conteneurs tels que des files ou des piles, et qu'un certain nombre d'opérations seront effectuées dessus (ajout, suppression, parcours).

En conséquence le coût du maintien des informations à propos d'une tâche entre sa création et son exécution dépend du support exécutif et des structures de données qu'il utilise. Ce point est illustré dans la section 2.3.5.

2.3.2.3 Exécution

Cette étape est l'un des points où la gestion par tâche dispose d'un gros avantage par rapport à la création individuelle de threads par le programmeur. Au début de l'exécution de l'application, un certain nombre de cœurs physiques sont utilisables par le support exécutif (cela peut être déduit implicitement par le support exécutif, ou spécifié explicitement par le programmeur). Lors de son initialisation, le support exécutif va **créer et attacher** un thread logique par cœur physique, virtualisant ainsi la gestion des cœurs physiques.

Ces threads vont se voir attribuer différents attributs, comme par exemple une structure de données contenant des tâches. Ils seront des «travailleurs» permanents pour le support exécutif, qui leur donnera des tâches à exécuter au fur et à mesure.

Les threads sont donc les mêmes tout au long de l'exécution de l'application, ce qui évite les coûts liés à la création ou à la destruction de threads.

2.3.3 Moyens de synchronisation

Lorsqu'on parle de programmation parallèle, il faut bien évidemment parler de synchronisation. Les différentes tâches définies par l'utilisateur vont être exécutées en parallèle sur la machine, mais dans beaucoup de cas certaines tâches doivent attendre la complétion d'une ou plusieurs tâches avant de pouvoir commencer à être exécutées.

Il y a deux grands types de synchronisations pour la programmation à base de tâches : la synchronisation explicite, et les dépendances de données.

2.3.3.1 Synchronisation globale

Le programmeur peut ajouter un point de synchronisation global dans le code. Pour les tâches OpenMP, ce moyen de synchronisation est le `taskwait`, et il est montré en exemple dans le listing 2.1. Lorsque le thread exécutant la tâche atteint le `taskwait`, il se bloque et attend que l'ensemble des tâches créées par la tâche courante soient terminées. Dans l'exemple le thread arrivant sur le `taskwait` attendra donc la complétion des tâches B et C, mais n'aura pas besoin d'attendre la complétion de la tâche A.

Listing 2.1: Synchronisation dans le thread courant (OpenMP)

```
1  void foo() {
2    #pragma omp task // Tâche 1
3    {
4      #pragma omp task // Tâche A
5      A();
6      #pragma omp task // Tâche 2
7      {
8        #pragma omp task // Tâche B
9        B();
10       #pragma omp task // Tâche C
11       C();
12       #pragma omp taskwait
13     }
14   }
15 }
```

Cette sémantique est particulière à OpenMP, dans d'autres modèles de programmation tels que Cilk ou Kaapi, le point de synchronisation imposerait l'attente de la complétion de toutes les tâches créées avant le point de synchronisation.

2.3.3.2 Dépendances de données

Le programmeur spécifie des dépendances de données, avec des modes, pour chacune des tâches. Dans l'exemple du listing 2.2, la tâche `write_A` possède une dépendance en *écriture* sur la variable `a`, et la tâche `read_A` possède une dépendance en *lecture*.

Étant donné qu'il y a une lecture et une écriture, le principe de cohérence séquentielle impose que les opérations soient ordonnées dans l'ordre où elles ont été créées (ici la tâche en lecture devrait avoir lieu **après** la tâche en écriture).

Si on regarde le reste du programme, la tâche `write_B` dispose d'une dépendance en écriture sur `b` et la tâche `read_AB` souhaite lire `a` et `b`.

Étant donné que du point de vue des dépendances les tâches `write_A` et `write_B` sont indépendantes, elles pourraient très bien être exécutées en même temps, et `write_B` pourrait terminer son exécution avant même que `read_A` commence la sienne.

En revanche `read_AB` devra forcément être exécutée après `write_A` et `write_B` puisqu'elle a une dépendance en lecture sur des données écrites par ces deux tâches, et qu'elle a été créée après dans l'ordre séquentiel.

Listing 2.2: Synchronisation via des dépendances (OpenMP)

```
1  void foo() {
2    int a;
3    int b;
4    #pragma omp task depend(out: a)
5    write_A(&a);
6    #pragma omp task depend(in: a)
7    read_A(&a);
8    #pragma omp task depend(out: b)
9    write_B(&b);
10   #pragma omp task depend(in: a, b)
11   read_AB(a, b);
12 }
```

Cet exemple de code se traduit en un graphe de tâches direct et acyclique (*DAG*) équivalent à la figure 2.3, sur lequel apparaissent également les différentes versions des variables.

Exprimer son code avec des dépendances de données plutôt qu'avec des synchronisations globales permet de définir des synchronisations plus fines entre les différentes tâche, que le support exécutif peut utiliser pour minimiser le temps d'inactivité des ressources. En effet, en utilisant des dépendances de données une tâche devient prête dès lors que ses données en lecture sont prêtes, ce qui, contrairement à une approche par synchronisation globale, est robuste à un déséquilibre de charge pour quelques raisons que ce soit.

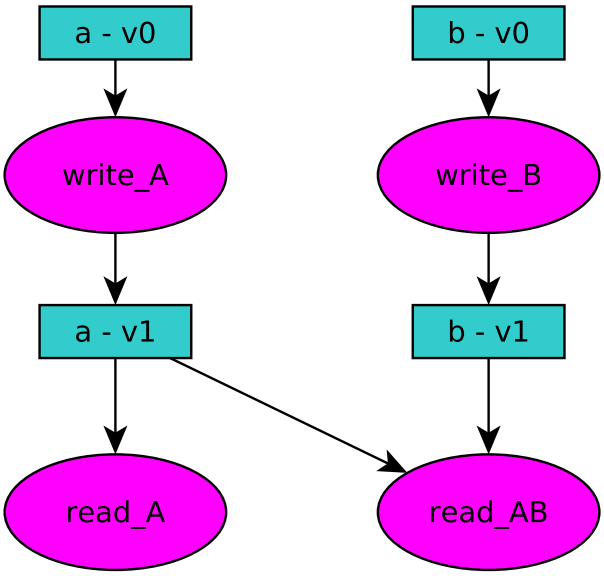

Figure 2.3: DAG équivalent au listing 2.2

2.3.4 Quelques exemples de modèles de programmation

Plusieurs modèles de programmation populaires proposent d'exprimer du parallélisme à base de tâches, parmi lesquels Cilk, OmpSs, et OpenMP, qui fonctionnent avec des approches légèrement différentes. Le but de cette section est de comparer l'expression d'une même application, la factorisation de Cholesky, dans ces différents modèles. Une comparaison plus large des supports exécutifs est effectuée dans la section 3.2.

2.3.4.1 Code séquentiel de base

La factorisation de Cholesky sera prise comme étude de cas et présentée en détails dans la section 4.3. Le code séquentiel utilisé comme base pour la parallélisation est présenté dans le listing 2.3.

2.3.4.2 Cilk

Cilk [Frigo 1998] est un modèle de programmation basé sur C. Il introduit principalement deux nouveaux mots clés : `spawn` et `sync`, pour, respectivement, exposer du parallélisme et introduire un point de synchronisation. Le mot clé `spawn` vient précéder un appel de fonction pour indiquer que la fonction peut s'exécuter en parallèle. Cela en fait donc un modèle de programmation à base de tâches. Cilk propose également une extension de la notation de tableau, ayant pour but de faciliter la vectorisation automatique par le compilateur. La version parallélisée en Cilk est présentée dans le listing 2.4.

2.3.4.3 OpenMP

OpenMP [OpenMP Architecture Review Board 2015] est un modèle de programmation supportant le C/C++ et Fortran. Le standard d'application de nos idées pour

Listing 2.3: Algorithme séquentiel

```
// Prototype des fonctions communes aux modèles de programmation
void dpotrf(double *A);
void dtrsm(double *A, double *B);
void dsyrk(double *A, double *B);
void dgemm(double *A, double *B, double *C);

// On suppose ici que l'expression A(i, j) retourne un pointeur
// vers le début du bloc i,j de la matrice A.
for (int k = 0; k < n_blocs; k++) {
  dpotrf(A(k, k));
  for (int m = k+1; m < n_blocs; m++)
    dtrsm(A(k, k), A(k, m));
  for (int m = k+1; m < n_blocs; m++) {
    for (int n = k+1; n < m; n++) {
      dgemm(A(k, n), A(k, m), A(n, m));
    }
    dsyrk(A(k, m), A(k, k));
  }
}
```

Listing 2.4: Cholesky exprimé en Cilk

```
for (int k = 0; k < n_blocs; k++) {
  spawn dpotrf(A(k, k));
  sync;
  for (int m = k+1; m < n_blocs; m++)
    spawn dtrsm(A(k, k), A(k, m));
  sync;
  for (int m = k+1; m < n_blocs; m++) {
    for (int n = k+1; n < m; n++) {
      spawn dgemm(A(k, n), A(k, m), A(n, m));
    }
    spawn dsyrk(A(k, m), A(k, k));
    sync;
  }
}
```

cette thèse étant OpenMP, une description détaillée des fonctionnalités et de ses spécificités est faite dans la section 2.5.

Contrairement à Cilk où la synchronisation est faite via un `sync` global, dans l'implémentation donnée dans le listing 2.5 ce sont les dépendances de données qui induisent l'ordre d'exécution.

Listing 2.5: Cholesky exprimé en OpenMP

```
for (int k = 0; k < n_blocs; k++) {
  #pragma omp task depend(inout:A(k,k))
  dpotrf(A(k, k));
  for (int m = k+1; m < n_blocs; m++) {
    #pragma omp task depend(in:A(k, k)) depend(inout:A(k, m))
    dtrsm(A(k, k), A(k, m));
  }
  for (int m = k+1; m < n_blocs; m++) {
    for (int n = k+1; n < m; n++) {
      #pragma omp task depend(in:A(k, n), A(k, m))
                       depend(inout:A(n, m))
      dgemm(A(k, n), A(k, m), A(n, m));
    }
    #pragma omp task depend(in:A(k, m)) depend(inout:A(k, k))
    dsyrk(A(k, m), A(k, k));
  }
}
```

2.3.4.4 OmpSs

OmpSs [Duran 2011] est un modèle de programmation proche d'OpenMP, mais où certaines *déclarations* de fonctions sont marquées comme des tâches avec leurs dépendances (et leurs tailles associées). Le listing 2.6 montre la factorisation de Cholesky exprimée en OmpSs.

2.3.4.5 Quelques autres modèles de programmation

En plus des modèles présentés ci-dessus et des modèles abordés en détails dans la section 3.2, il existe un certain nombre de modèles de programmation à base de tâches, trop éloignés de nos travaux pour en faire une description approfondie.

Threading Building Block (TBB) [Reinders 2007] est un modèle de programmation développé par Intel comme une bibliothèque C++. Les tâches sont soumises au support exécutif via des fonctions mises à disposition du programmeur, qui doit passer en paramètre un pointeur vers une fonction au prototype prédéfini.

Hpx [Kaiser 2014] est un modèle de programmation construit sur le modèle *Partitioned Global Address Space* [PGAS 2013]. Il s'agit également d'une API C++, où les primitives de création de tâches prennent en paramètre un pointeur de fonction et une liste d'arguments à lui passer.

X10 [Charles 2005] est un langage basé sur Java utilisant un modèle PGAS. Les opérations sur les tâches (création, synchronisation) se font de manière similaire à Cilk, à l'aide de mots clés ajoutés dans le langage. Le langage X10 supporte une synchronisation fine des tâches à travers la notion de *futures* : le résultat d'une tâche peut être encapsulé dans un objet *future*, sur lequel d'autres tâches peuvent se placer en attente.

Listing 2.6: Cholesky exprimé en OmpSs

```
// Taille de bloc
int bs = X;

#pragma omp task inout([bs][bs]A)
void dpotrf(double *A);
#pragma omp task in([bs][bs]A) inout([bs][bs]B)
void dtrsm(double *A, double *B);
#pragma omp task in([bs][bs]A) inout([bs][bs]B)
void dsyrk(double *A, double *B);
#pragma omp task in([bs][bs]A, [bs][bs]B) inout([bs][bs]C)
void dgemm(double *A, double *B, double *C);

for (int k = 0; k < n_blocs; k++) {
  dpotrf(A(k, k));

  for (int m = k+1; m < n_blocs; m++)
    dtrsm(A(k, k), A(k, m));

  for (int m = k+1; m < n_blocs; m++) {
    for (int n = k+1; n < m; n++) {
      dgemm(A(k, n), A(k, m), A(n, m));
    }
    dsyrk(A(k, m), A(k, k));
  }
}
```

2.3.5 Quantité de travail et granularité

Dans ce type de modèles de programmation, l'une des clés pour maximiser l'utilisation des ressources est de réduire l'overhead du support exécutif par rapport au calcul en trouvant le bon *grain* de tâche.

Il faut donc jouer sur le degré de parallélisme pour atteindre les meilleures performances : les tâches doivent être suffisamment petites pour proposer le maximum de parallélisme, mais pas trop pour ne pas surcharger le support exécutif, vis à vis des coûts décrits dans la section 2.3.2.

Ce grain optimal dépend de plusieurs facteurs : les structures de données utilisées par le support exécutif, le coût de création des tâches, et la quantité de travail mis à disposition par cœur via ce grain.

Cela peut être illustré via une application telle que la factorisation de Cholesky par bloc présentée dans la section précédente : à taille de matrice fixée le nombre de tâches créées dépend directement de la taille de bloc choisie. Plus la taille de bloc est petite, plus le nombre de blocs créés (et donc le nombre de tâches, et le parallélisme potentiel) est important.

La figure 2.4 illustre l'évolution des performances d'une factorisation de Cholesky d'une matrice de taille 8192, sur un nombre de cœurs fixé (64), en fonction de la taille

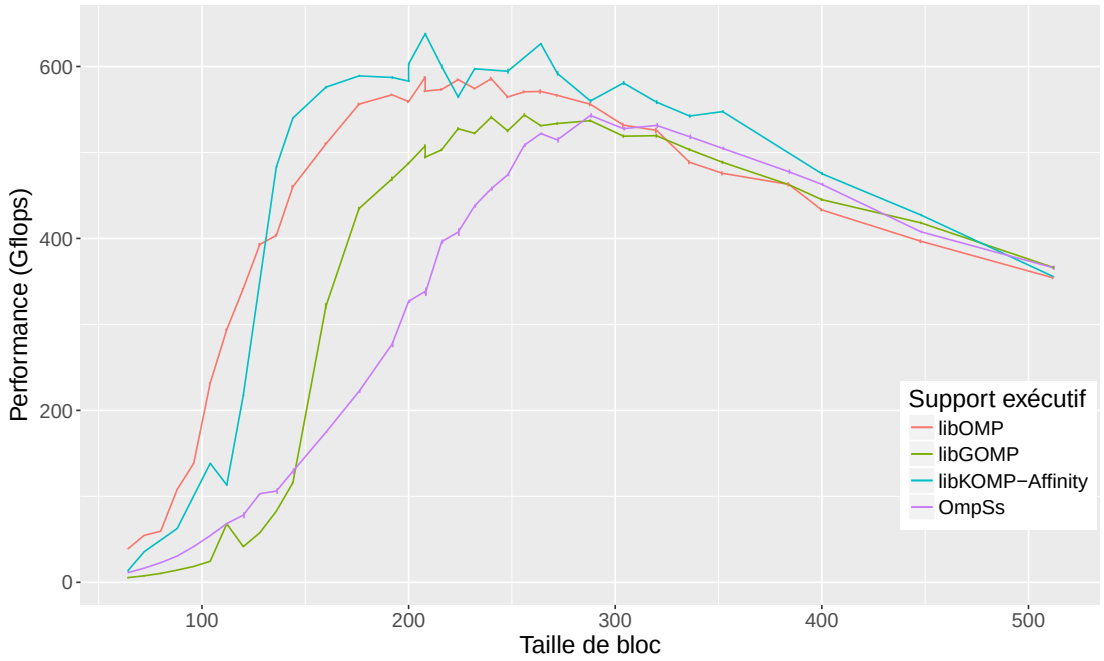

Figure 2.4: Performances de Cholesky pour une matrice de taille 8192 et 64 threads, en fonction de la taille de bloc

de bloc et du support exécutif. La machine utilisée pour cette expérience est *idchire*, qui sera décrite en détails dans la section 4.2.1. De même une description détaillée des supports exécutif sera faite dans la section 5.4.1.1 pour libKOMP, et dans la section 3.2 pour les autres.

Comme on peut le voir, les parties extrèmes de la courbe se comportent de manière similaires quel que soit le support exécutif : une taille de bloc trop faible génère beaucoup trop de tâches et les support exécutifs sont complètement surchargés. Une taille de bloc trop importante limite complètement le parallélisme et donc les performances.

Le grain adapté n'est pas nécessairement unique : en fonction du support exécutif on peut avoir un choix plus ou moins important. Cela est illustré sur la figure 2.4 : avec OmpSs le grain optimal se situe dans un intervalle de tailles de blocs assez restreint (entre environ 250 et 350), alors qu'avec clang ou libkomp une taille de bloc entre 150 et 350 permet d'obtenir des performances sensiblement équivalentes.

La courbe tracée avec libkomp inclut nos travaux sur l'affinité, et permet d'illustrer que même si l'affinité permet d'influer sur les performances maximales, elle n'a pas vraiment d'impact sur le grain optimal, comme cela est illustré sur la figure 2.5 (également réalisée sur la machine *idchire*), ou le maximum de performance est obtenu pour tous les supports exécutifs aux environs de 300.

Le choix du grain pour une tâche dépend entièrement de l'application, et reste à l'appréciation du programmeur.

Bien que tous les modèles de programmation à base de tâches avec dépendances aient leur spécificités, ils permettent tous de décrire l'application sous forme de graphe de tâches direct et acyclique (DAG).

L'étape suivante consiste à exécuter ce graphe sur la machine, et pour cela

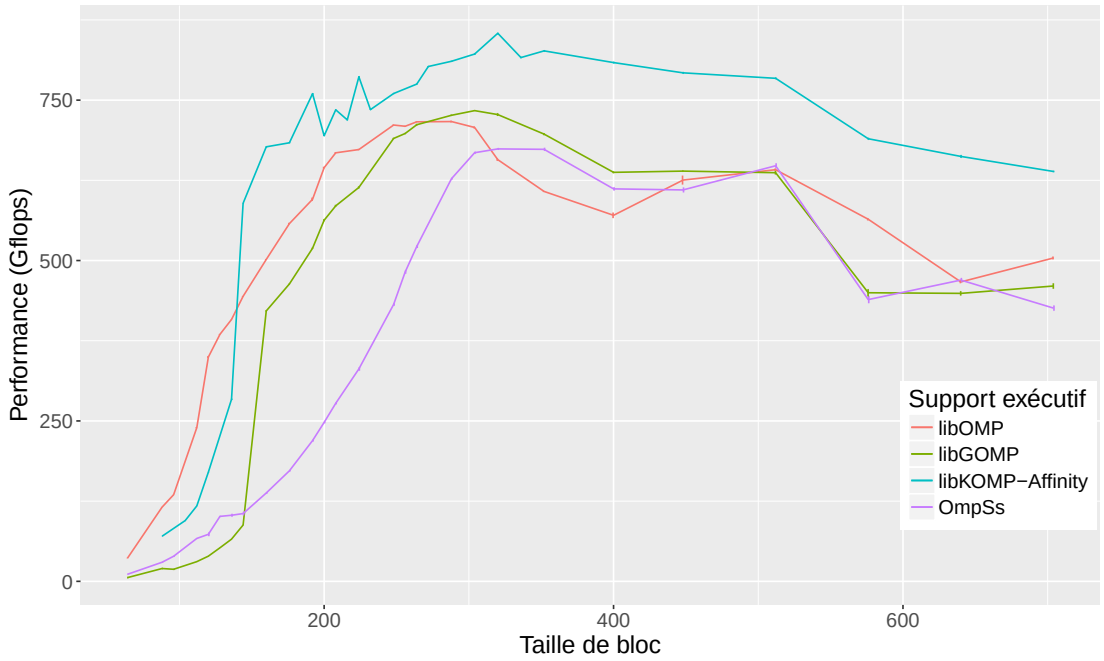

Figure 2.5: Performances de Cholesky pour une matrice de taille 163384 et 64 threads, en fonction de la taille de bloc

le support exécutif peut se reposer sur un ensemble important de techniques d'ordonnancement.

2.4 Techniques d'ordonnancement pour supports exécutifs

On peut distinguer deux grands types d'ordonnancement : les techniques de rééquilibrage de la charge de travail en cours d'exécution, dites *online*, et les techniques dites *offline* [Karp 1992], où les informations sur les tâches à exécuter sont connues à l'avance, et où un ordonnancement est calculé avant l'exécution du programme. L'objectif de nos travaux est l'amélioration d'un support exécutif d'un modèle de programmation du type d'OpenMP : nous pourrons certes avoir certaines informations avant même l'exécution du programme, mais l'ordonnancement se fera principalement à la volée et nous porterons donc un intérêt particulier aux techniques *online*. Les sections suivantes décrivent les différentes caractéristiques de ces techniques, ainsi que quelques exemples en rapport avec nos travaux. En particulier la section 2.4.2.2 décrit en détail le *vol de travail*, un concept largement utilisé dans les supports exécutifs auxquels nous nous sommes intéressés.

2.4.1 Ordonnancement *offline*

Les techniques d'ordonnancement *offline* supposent connues les informations à propos du matériel et des tâches à exécuter, telles que leur temps d'exécution en utilisant telle où telle ressource, ou encore les contraintes de précédence à respecter pour

l'exécution. Avant l'exécution un ordonnancement des tâches est généré à partir de ces informations, et il est stocké pour être ensuite rejoué à l'exécution.

Ce type de calcul d'ordonnancement a plusieurs avantages : d'une part l'exécution de l'application est totalement déterministe et reproductible ; ensuite cela entraine très peu de coût au niveau du support exécutif lors de l'exécution du programme, puisque les décisions ont déjà été prises ; et enfin cela permet a priori de minimiser le temps d'exécution, car un algorithme *offline* à une connaissance parfaite du programme et peut sélectionner la séquence de tâches qui est la plus avantageuse.

En revanche il y a plusieurs détails qui peuvent jouer contre l'utilisation de telles techniques [Locke 1992] : l'ordonnancement généré est très sensible à la justesse des informations utilisées. Si une ou plusieurs tâches viennent à durer plus longtemps que prévu, cela peut tout simplement rendre l'ordonnancement très inefficace. Pour cette raison beaucoup de supports exécutifs décident d'utiliser des techniques dynamiques, où l'ordonnanceur réagit au cours de l'exécution. Ils peuvent éventuellement être aidé d'un modèle d'exécution pour améliorer les décisions, mais compte tenu de la complexité des architectures NUMA, cela reste très difficile d'obtenir des prévisions de temps d'exécution précises.

Comme on l'a vu dans la section précédente, dans un modèle de programmation à base de tâches, le programme est représenté par un DAG. La "meilleure" exécution de l'application revient donc à calculer l'ordonnancement optimal de ce DAG sur les ressources disponibles. Malheureusement ce problème est NP-complet [Cook 1971], il faut donc trouver des approximations, ou ajouter des restrictions sur le DAG ou les ressources. Nous mentionnons ci-après quelques algorithmes offline, qui ont d'ailleurs parfois été adaptés pour être — partiellement — utilisés dans des méthodes online.

Dominant Sequence Clustering (DSC) [Yang 1994] est un algorithme permettant de trouver un ordonnancement optimal pour certaines classes de DAG, et de trouver des 2-approximations pour tout type de DAG, sur des architectures homogènes non bornées. Pour l'ordonnancement sur des ressources hétérogènes, Heterogeneous Earliest Finish Time (HEFT) [Topcuoglu 2002], et Critical-Path-on-a-Processor (CPOP) [Topcuoglu 2002], sont des heuristiques très populaires qui analysent le DAG de l'application et assignent une priorité aux tâches en fonction d'informations connues statiquement, telles que le coût des transferts de données vers chaque ressource, ou les durées estimées des tâches. CPOP ajoute en plus la prise en compte du *chemin critique* dans cette priorité.

On peut remarquer que HEFT a été adapté dans le cadre d'ordonnancement online d'application : l'ordonnanceur part d'une première estimation du temps d'exécution de chaque tâche, mais à la fin de chaque tâche il peut réagir en fonction du temps effectivement écoulé pour le calcul de cette tâche. Cette technique est par exemple utilisé dans StarPU [Augonnet 2011], et a été implémentée et analysée par rapport à d'autres stratégies dans XKaapi [Lima 2015].

Certaines applications que l'on a étudiée et utilisée sont des applications d'algèbre linéaire, et Jakub Kurzak et al. [Kurzak 2010] ont étudié l'ordonnancement *offline* «*static pipeline*» [Kurzak 2008, Kurzak 2009] pour ce type d'application, et l'ont comparé à différentes implémentations *online*, telle que Cilk, OpenMP (version 3.0), ou encore un prédécesseur d'OmpSs : SMPSs [BSC 2008]. Si la version ordonnancée *offline*

restait meilleure, certains ordonnancements *online* étaient très proches, et leur conclusion rappelait que les performances de ces ordonnancements *online* pouvaient encore être améliorées.

2.4.2 Ordonnancement *online*

Contrairement à l'ordonnancement offline, les techniques d'ordonnancement online vont prendre les décisions concernant l'ordonnancement des tâches à l'exécution. Ces techniques n'ont pas besoin d'information préalable sur les tâches, cela les rend donc beaucoup plus réactives aux aléas de l'exécution que les techniques offline. En contrepartie, la prise de décision à l'exécution introduit un surcoût pour le support exécutif, se répercutant directement sur le temps total d'exécution.

Nous détaillons ci-après deux types d'ordonnancement online populaires : l'ordonnancement de listes glouton et le vol de travail.

2.4.2.1 Algorithmes de listes gloutons

L'ordonnancement glouton fonctionne assez simplement : le support exécutif maintient une unique liste de tâches *prêtes* à être exécutées. À chaque fois qu'un processeur devient disponible, l'ordonnanceur lui assigne une tâche prête. Il met ensuite à jour la liste des tâches prêtes, et recommence l'étape précédente jusqu'à ce qu'il n'y ait plus de tâche à exécuter.

Les performances d'un tel algorithme en fonction de la manière de trier la liste de tâches ont été bornées par Graham [Graham 1966]. Les heuristiques pour le choix de la tâche prête à exécuter sont nombreuses, et la plupart d'entre elles prennent en compte la position de la tâche sur le *chemin critique*. Nous faisons un état de l'art des publications à ce sujet et en lien avec nos travaux dans la section 3.1.

2.4.2.2 Vol de travail

Le vol de travail [Blumofe 1996] fait partie des techniques d'ordonnancement online les plus répandues, et est notamment utilisé dans les supports exécutifs étudiés lors de cette thèse. Cette section revient donc en détail sur les mécaniques clés de ce type d'ordonnancement.

Au contraire des algorithmes de listes gloutons, le vol de travail fonctionne à la base avec une files de tâches prêtes par thread. Le principe est le suivant : à chaque fois qu'un thread devient inactif, celui-ci va aller «voler» du travail dans une file de tâches prêtes. S'il réussit à récupérer du travail (une tâche), il va l'exécuter. Une fois sa tâche terminée, il va indiquer cette dépendance comme satisfaite dans les successeurs de la tâche, et si certaines ont toutes leurs dépendances de satisfaites, il va les introduire dans une des files de tâches prêtes.

On constate donc qu'il y a deux moments clés où le thread doit prendre une décision importante : le choix de la file pour l'ajout des nouvelles tâches prêtes, et le choix de la victime pour le vol.

Placement d'une tâche prête. La deuxième prise de décision concerne le placement des tâches. Lorsqu'un thread termine une tâche, il va devoir indiquer aux successeurs de cette tâche les dépendances qui viennent d'être satisfaites, s'il y en a. Si pour l'un

des successeurs toutes les dépendances sont satisfaites, alors cette tâche devient prête, et il faut sélectionner une file où la placer.

Encore une fois de nombreuses heuristiques sont possibles. Un choix relativement commun dans le vol de travail introduit par Cilk est de faire en sorte que le thread place cette tâche dans sa propre file de tâches.

Dans la section 5.3.2, nous revenons sur comment nous avons pu agir au niveau des deux prises de décisions précédentes, afin de prendre en compte la topologie des machines NUMA.

Choix de la victime. Lors du vol de travail, le thread voleur se retrouve à devoir choisir à qui envoyer une requête de vol parmi les files de tâches disponibles. Il y a évidemment plusieurs heuristiques possibles, et concrètement la "bonne" heuristique dépend directement du nombre de files de tâches disponibles et de leurs caractéristiques, mais aussi du type de la ressource effectuant la requête de vol.

Il y a quelques principes simples qui sont généralement utilisés, tels que commencer par voler dans sa propre file de tâches avant d'aller voler ailleurs, mais généralement quand il y a peu d'information sur les tâches et les ressources, il suffit de choisir une victime aléatoirement.

2.4.3 Offline vs Online, lequel choisir ?

Les applications peuvent avoir plusieurs sources d'irrégularités : d'une part cela peut tout simplement venir de la structure de l'application elle-même, mais encore du jeu de données en entrée de l'application, ou bien même des caractéristiques de l'architecture sur laquelle s'exécute l'application. Cette dernière source d'irrégularités peut être particulièrement présente sur les architectures hétérogènes ou NUMA, étant donné que sur celles-ci les conditions d'accès à la mémoire peuvent être difficilement prévisibles.

Cela veut donc dire que même pour des applications à première vue régulières, une technique de base *online* telle que le vol de travail peut être envisagée. En revanche il est important que cette approche soit complétée par des techniques a priori *offline*, afin de rassembler un maximum d'informations sur les tâches exécutées, pour au final améliorer la prise de décision. De nombreux travaux utilisent une première approche *offline* pour faire une première répartition de la charge de travail, et ajustent ensuite l'équilibrage au fur et à mesure de l'exécution de l'application. C'est le cas par exemple de Durand et al. [Durand 2013], qui proposent un ordonnanceur de boucle adaptatif pour répondre à l'irrégularité des itérations de boucle dans certains types de programme. Dans un contexte de tâches avec dépendances, Gautier et al. [Gautier 2007] effectuent un premier partitionnement du graphe de tâches avant d'utiliser du vol de travail pour équilibrer la charge. Dans un contexte hétérogène, Gautier et al. [Gautier 2013] ont étudié les noyaux impliqués dans la factorisation de Cholesky, marqué ceux a priori inefficace sur GPU, et ont montré que des heuristiques de vol de travail prenant en compte ces informations permettaient d'obtenir de meilleurs résultats que des ordonnancements purement *offline*. Agullo et al. [Agullo 2016] tirent une conclusion similaire : ils ont montré qu'un apport minimum d'informations obtenues *offline* à propos des noyaux de Cholesky permet

d'améliorer grandement les performances de l'application à travers des supports exécutifs *online*.

2.5 Évolution d'un modèle de programmation : OpenMP

OpenMP [OpenMP Architecture Review Board 2015] est le standard de-facto pour exploiter les machines parallèles à mémoire partagée.

Jusqu'à la version 2.0 incluse, OpenMP ne proposait qu'un nombre restreint de fonctionnalités, principalement tournées vers la parallélisation de boucles régulières. L'évolution de son utilisation a amené le comité de standardisation d'OpenMP à ajouter le concept de *tâche* à partir de la version 3.0, en s'inspirant de modèles de programmation déjà existants tel que Cilk. La version 4.0 d'OpenMP étend les tâches pour ajouter les dépendances de données, et introduit le support pour du matériel de plus en plus utilisé : les accélérateurs. On y voit aussi l'apparition de mécanismes permettant de contrôler le placement des threads, des directives SIMD, ou encore des instructions atomiques. La version actuelle d'OpenMP est la version 4.5, qui ajoute certaines fonctionnalités aux constructions existantes, et introduit le concept de *taskloop* : des boucles dont les groupes d'itérations forment des tâches indépendantes. La version en cours de préparation est OpenMP 5.0 ; parmi diverses améliorations, elle devrait notamment ajouter le support d'une interface utilisable par des outils de visualisation de traces d'exécution et de debugging interactif souhaitant se greffer au plus proche d'OpenMP, ainsi que le support des réductions sur les tâches.

Les sections suivantes décrivent les différentes fonctionnalités d'OpenMP, dans l'ordre chronologique où elles sont apparues.

2.5.1 Fonctionnement de base

L'utilisation d'OpenMP repose sur deux ensembles de fonctionnalités : le premier est accessible à travers l'utilisation de `pragma` (des directives de compilation) dans le code, et le second est accessible à travers une API dédiée. OpenMP a été pensé dès le départ comme un modèle *fork-join* (illustré sur la figure 2.6)

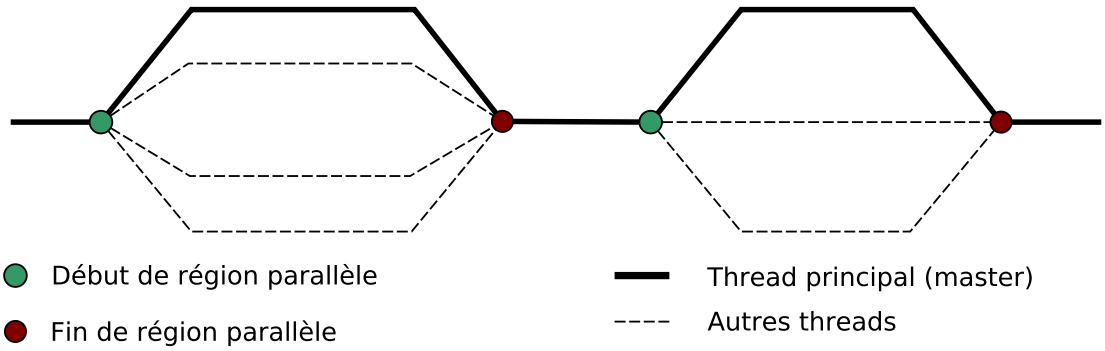

Figure 2.6: Ligne de vie d'un programme dans un modèle fork-join

Le modèle fonctionne de la manière suivante : le programme est exécuté séquentiellement jusqu'à la rencontre d'une région parallèle (`pragma omp parallel`, comme

illustré dans le listing 2.7). À partir de là, plusieurs threads sont créés (ou réutilisés) pour exécuter le bloc de code suivant en parallèle. Cet ensemble de threads forme une *team* de threads pour la région parallèle. À la fin de la région parallèle, tous les threads se synchronisent et l'exécution séquentielle est reprise. Le programmeur peut contrôler combien de threads sont utilisés dans la *team* créée lors de l'entrée dans la région parallèle.

Pour complémenter les fonctionnalités au niveau du langage, OpenMP propose également plusieurs fonctionnalités à travers une API. Il est possible par exemple de récupérer dynamiquement le nombre total de threads, l'indice du thread courant, ou encore d'ajuster l'ordonnanceur de boucles.

Listing 2.7: Région parallèle minimale

```
1 int main()
2 {
3   #pragma omp parallel
4   {
5     // Code à exécuter en parallèle
6   }
7   return 0;
8 }
```

Lors de l'exécution de l'application c'est un programme spécifique - le *support exécutif* qui est en charge de l'équilibrage de charge, et de l'affectation du "travail" aux différents threads. Si le langage OpenMP est standard et bien défini, les performances dépendent directement du support exécutif.

2.5.2 Boucles

L'une des constructions de partage de travail - ou *worksharing construct* - est la boucle. Exprimée au moyen d'un `pragma omp for`, elle permet d'indiquer que la boucle à laquelle est attaché le pragma doit être exécutée en parallèle.

Les itérations de la boucle peuvent être réparties sur les différents cœurs de calcul de différentes manières :

statiquement : chaque cœur se voit attribuer un nombre prédéfini d'itérations, typiquement N/P avec N le nombre d'itérations et P le nombre de processeurs. Dans ce genre de cas le compilateur peut directement prendre en charge le découpage.

dynamiquement : l'attribution d'une itération, ou d'un groupe d'itérations (*chunk*) se fait lors de l'exécution du programme par le support exécutif. Cela permet notamment de pouvoir faire de l'équilibrage de charge si toutes les itérations ne sont pas parfaitement régulières.

Il existe d'autres options pour contrôler la taille des groupes d'itérations, voire faire varier la taille de ces groupes au fur et à mesure de l'exécution. Le programmeur a bien sur un contrôle total sur ces options, soit directement dans le code à l'aide de clauses (`schedule`), soit lors de l'exécution via des fonctions de l'API.

Le listing 2.8 illustre la parallélisation d'une boucle simple effectuant l'initialisation d'un tableau. Avec 40 itérations, 4 threads demandés, et un `schedule` statique, chaque thread se verra donc attribuer 10 itérations.

Listing 2.8: Exemple d'utilisation d'un for

```
#define SIZE 40
int main()
{
  double tableau[SIZE];
  #pragma omp parallel num_threads(4)
  {
    #pragma omp for schedule(static)
    for (int i = 0; i < SIZE; i++) {
      tableau[i] = i;
    }
  }
  return 0;
}
```

Cette construction existe depuis les toutes premières versions d'OpenMP.

2.5.3 Tâches

Les modèles de programmation à base de tâches permettent d'exprimer du parallélisme à grain fin. L'un des avantages majeurs de ce modèle est qu'il permet au support exécutif d'assigner dynamiquement les différentes tâches, s'adaptant ainsi très bien aux déséquilibrages de charge. Il permet également de composer très facilement des parties de code avec des caractéristiques différentes.

Une *tâche* OpenMP peut être vue comme la plus petite *quantité de travail* qu'un thread OpenMP peut exécuter. Les tâches peuvent être créées par un thread OpenMP et exécutées par n'importe quel thread de la région parallèle. Comme la création de tâches à l'exécution du programme est beaucoup plus économique que la création et la synchronisation de threads, le développeur peut pousser la parallélisation de son application encore plus loin : il peut considérer la parallélisation de portions de code qui avaient un grain trop fin pour être parallélisées avec des threads.

Dans la version 3.0 d'OpenMP la synchronisation des tâches est effectuée grâce au mot clé `taskwait`, qui indique au support exécutif d'attendre la complétion des tâches générées jusqu'à ce point dans la région parallèle, avant de reprendre l'exécution. Le développeur de l'application est responsable de la création et de la synchronisation explicite des tâches, mais c'est le support exécutif qui est en charge de l'affectation des tâches aux threads pendant l'exécution du programme.

Le listing 2.9 illustre ce concept en reprenant l'exemple d'une factorisation de Cholesky, donné dans la section 2.3.4.

La version 4.0 d'OpenMP [OpenMP Architecture Review Board 2013] pousse le concept de tâche plus loin en ajoutant le mot clé `depend`, spécifiant les modes d'accès de chaque variable partagée utilisée par la tâche pendant son exécution. Le mode d'accès peut être `in`, `out`, ou `inout` selon que la variable correspondante soit respectivement lue comme entrée, écrite en sortie, ou à la fois lue et écrite par la tâche en question. Cette information peut ensuite être traitée par le support exécutif pour décider si une tâche est prête à être exécutée ou s'il faut d'abord attendre la complétion d'une ou plusieurs autres tâches, ce qui permet de s'affranchir d'une synchronisation par `taskwait` dans

Listing 2.9: Création et synchronisation explicite de tâches

```
// On suppose ici que l'expression A(i, j) retourne un pointeur
// vers le début du bloc i,j de la matrice A.
for (int k = 0; k < n_blocs; k++) {
  #pragma omp task
  dpotrf(A(k, k));
  #pragma omp taskwait
  for (int m = k+1; m < n_blocs; m++) {
    #pragma omp task
    dtrsm(A(k, k), A(k, m));
  }
  #pragma omp taskwait
  for (int m = k+1; m < n_blocs; m++) {
    for (int n = k+1; n < m; n++) {
      #pragma omp task
      dgemm(A(k, n), A(k, m), A(n, m));
    }
    #pragma omp task
    dsyrk(A(k, m), A(k, k));
    #pragma omp taskwait
  }
}
```

Listing 2.10: Exemple de tâches avec dépendances

```
for (int k = 0; k < n_blocs; k++) {
  #pragma omp task depend(inout:A(k,k))
  dpotrf(A(k, k));
  for (int m = k+1; m < n_blocs; m++) {
    #pragma omp task depend(in:A(k, k)) depend(inout:A(k, m))
    dtrsm(A(k, k), A(k, m));
  }
  for (int m = k+1; m < n_blocs; m++) {
    for (int n = k+1; n < m; n++) {
      #pragma omp task depend(in:A(k, n), A(k, m))
                       depend(inout:A(n, m))
      dgemm(A(k, n), A(k, m), A(n, m));
    }
    #pragma omp task depend(in:A(k, m)) depend(inout:A(k, k))
    dsyrk(A(k, m), A(k, k));
  }
}
```

énormément de cas.

Le listing 2.10 donne un exemple de tâches avec dépendances, où la variable `foo` est initialisée puis affichée.

La dernière extension aux tâches a vu le jour dans OpenMP 4.5, qui introduit la notion de *priorité* sur les tâches, permettant d'aider le support exécutif à choisir quelle tâche prête exécuter en priorité. L'extension suivante devrait être la réduction sur les tâches, qui devrait voir le jour avec OpenMP 5.0, et pour laquelle nous avons déjà montré qu'elle pouvait permettre d'améliorer le parallélisme de certaines applications [Virouleau 2014].

2.5.4 Vectorisation

Nous avons vu dans la section 2.1.1.2 que les processeurs disposent d'instructions vectorielles - *SIMD* - depuis le début des années 2000. Avec la multiplication des architectures et de la taille des registres, beaucoup de nouvelles instructions spécifiques à certaines tailles de registre sont apparues.

Si la vectorisation automatique n'est pas possible, il est coûteux pour le programmeur de modifier son code en fonction des architectures. OpenMP 4.0 tente de résoudre ce problème par l'introduction d'une construction `simd`, qui permet au programmeur d'indiquer quelles parties de son code peuvent être vectorisées et avec quelles contraintes (longueur maximale, alignement, quelles sont les variables d'itérations, ...). On peut voir ça comme une façon portable d'indiquer au compilateur comment vectoriser un code applicatif complexe.

Le listing 2.11, donné comme exemple par [Xinmin 2014] illustre la parallélisation et la vectorisation d'une opération complexe sur un tableau. Dans cet exemple les itérations seront réparties sur les différents threads, et au sein de chaque thread le code sera vectorisé. Du aux dépendances entre les différentes itérations de la boucle, le compilateur ne pourrait pas vectoriser automatiquement ce code. En revanche le programmeur peut identifier que pour une itération donnée, deux lectures sont faites à 18 cases d'écart dans le tableau. Une vectorisation utilisant un vecteur de taille inférieure ou égale à 18 est donc possible sans changer la sémantique du programme. L'utilisation de la clause `safelen` permet d'indiquer cela au compilateur.

Listing 2.11: Vectorisation d'une opération complexe sur des tableaux

```
#define N 1000000
float x[N][N], y[N][N];
#pragma omp parallel
{
  #pragma omp for
  for (int i = 0; i < N; i++) {
    #pragma omp simd safelen(18)
    for (int j = 18; j < N-18; j++) {
      x[i][j] = x[i][j-18] + sinf(y[i][j]);
      y[i][j] = y[i][j+18] + cosf(x[i][j]);
    }
  }
}
```

2.5.5 Accélérateurs

La majorité des supercalculateurs intègre des accélérateurs (comme des GPUs). L'exploitation de ces accélérateurs impose parfois l'usage d'un langage spécifique au constructeur (tel que Cuda).

Bien que des standards, comme OpenCL [Stone 2010], existent pour cibler différents types d'accélérateurs, cibler ce type d'architecture via OpenMP était impossible.

OpenMP 4.0 introduit la construction `target`, qui permet de demander au compilateur de créer une tâche à partir de la région de code sélectionnée, qui peut ensuite être exportée sur accélérateur. Plusieurs clauses permettent de spécifier des dépendances, ainsi que l'ensemble des données à transférer vers l'accélérateur, et l'ensemble des données à rapatrier depuis l'accélérateur à l'issue des calculs.

Listing 2.12: Addition sur un accélérateur

```
#define SIZE 40
int main()
{
  double tableau[SIZE];
  #pragma omp target map(tofrom: tableau[0:SIZE])
  {
    for (unsigned i = 0; i < SIZE; i++)
      tableau[i] = i;
  }
  return 0;
}
```

Si la norme indique le support de plusieurs accélérateurs, ils doivent cependant être tous du même type, et la transformation du code C/C++/Fortran du bloc est laissée au compilateur implémentant la norme. Dans le cas d'utilisation d'appels de fonctions, le programmeur doit indiquer qu'elles disposeront d'une définition pour l'*hôte* et l'*accélérateur* via une construction `declare target`.

Ces contraintes font qu'aujourd'hui la construction n'est pas clairement adoptée telle quelle par les programmeurs d'architectures hétérogènes.

2.5.6 Placement des threads

La version 4.0 d'OpenMP voit aussi l'ajout d'un ensemble de fonctionnalités ayant pour but de donner du contrôle au programmeur sur le placement des threads OpenMP sur la topologie physique de la machine.

Le premier ajout est celui du concept de *places* : il s'agit d'un moyen de représenter des emplacements physiques sur lesquels les threads OpenMP peuvent venir se placer, et permet d'indiquer au support exécutif quelles ressources physiques lui sont disponibles. Avant d'être introduit dans OpenMP, ce concept avait déjà été expérimenté séparément par le support exécutif de GCC — libGOMP, dans lequel la liste des *places* était contrôlée via la variable d'environnement `GOMP_AFFINITY`, mais également par le support exécutif d'Intel — libIOMP, qui utilisait lui la variable d'environnement `KMP_AFFINITY`.

Dans OpenMP le contrôle de la sélection des *places* se fait via la variable d'environnement `OMP_PLACES`, et la syntaxe est directement inspirée de ses prédécesseurs. Elle peut être une liste précise d'indices de cœurs physiques, ou prendre la valeur d'éléments plus génériques de la machine tels que *sockets* ou *cores*.

Il n'y a pas de dépendance entre le nombre de threads dans une région parallèle (spécifié par `OMP_NUM_THREADS`) et la liste des *places* : il peut y avoir plus de threads que de *places* ou inversement, sans que cela empêche l'exécution du programme.

Le second ajout est la possibilité de spécifier la manière dont sont affectés les threads aux *places*, à l'aide de la variable d'environnement `OMP_PROC_BIND`.

À travers cette option il est possible de demander à ce que les threads soient groupés proches les uns des autres sur les *places* (*close*), ou au contraire les plus éloignés possible (*spread*). S'il est possible d'expérimenter avec ces valeurs, il faut absolument éviter de désactiver le placement fixe des threads (valeur *false*) : cela permettrait au système d'exploitation de migrer les threads d'une région parallèle sur les différentes places, ce qui pourrait entraîner une perte de la localité des données, et donc une dégradation des performances. Une bonne valeur par défaut dans un contexte NUMA est de positionner `OMP_PROC_BIND` à *true*, afin de fixer les threads sur les cœurs. Il suffit ensuite d'utiliser l'expressivité des *places* pour spécifier exactement quels cœurs utiliser.

Cette section a décrit l'ensemble des connaissances et concepts de base nécessaires à la pleine compréhension de ce manuscrit. Cela concerne à la fois : le matériel, les architectures des processeurs et leur association pour former une machine NUMA ; le support du matériel au niveau du système d'exploitation ; les concepts exprimés dans les modèles de programmation, tels qu'OpenMP ; et enfin les détails d'implémentation des supports exécutifs.

Le chapitre suivant rentre beaucoup plus en détails sur les travaux de l'état de l'art en relation avec nos contributions, qui se basent sur les connaissances de ce chapitre.

« In the good old days physicists repeated each other's experiments, just to be sure. Today they stick to Fortran, so that they can share each other's programs, bugs included. »

Edsger W. Dijkstra

3 État de l'art

L'objectif de ce chapitre est de donner un aperçu des travaux existants dans le contexte de cette thèse. Comme on peut le voir sur la figure 3.1, un certain nombre d'acteurs sont impliqués dans l'exécution d'une application.

Les parties application, système d'exploitation, et matériel ont été traitées dans le chapitre précédent ; ce chapitre se concentrera dans un premier temps sur une partie précise des supports exécutifs, en abordant les différentes techniques

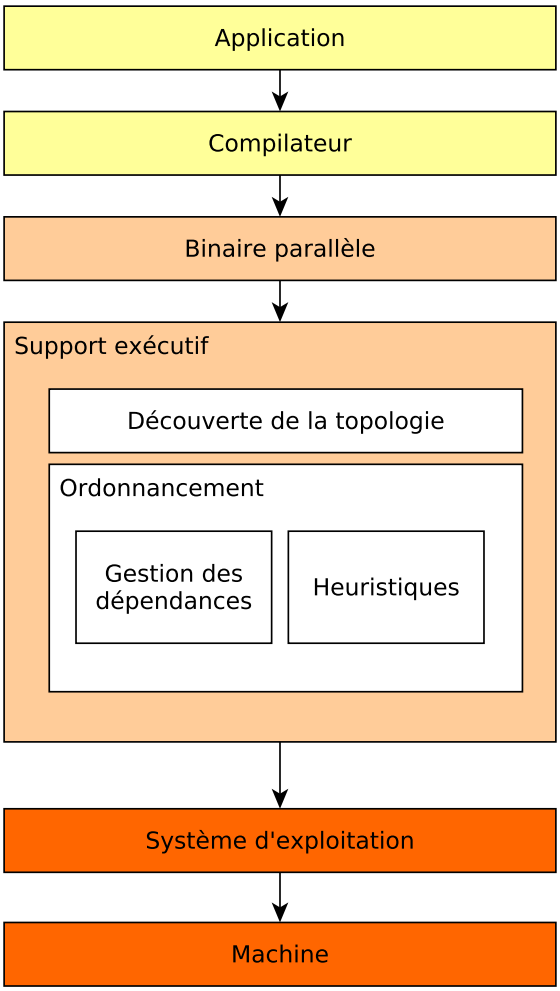

Figure 3.1: Schéma des acteurs impliqués dans l'exécution d'une application

d'ordonnancement présentes dans l'état de l'art pour cibler spécifiquement les machines NUMA, puis nous regarderons en détails certains supports exécutifs existants, et enfin nous ferons un point sur les compilateurs OpenMP.

3.1 Techniques d'amélioration de la localité des données

La problématique de la localité des données a été étudiée de manière extensive par la communauté, en particulier dans le contexte des architectures NUMA. Elle est essentielle car elle permet de limiter les transferts de données entre nœuds, d'améliorer le temps d'accès aux données, et donc d'améliorer les performances. Parmi les nombreuses publications sur le sujet, deux d'entre elles font l'effort d'en présenter une synthèse didactique, énumérant un certain nombre de bonnes pratiques à adopter s'agissant de l'exécution de tâches en contexte NUMA.

Reinman [Reinman 2015] a regroupé différents travaux selon le type de tâches qu'ils ciblaient, et propose en conclusion des recommandations lorsque l'on cherche à ordon-

nancer des programmes à base de tâches sur des machines NUMA.

L'auteur distingue principalement deux types de tâches : compute-bound et memory-bound. Si certaines des 8 recommandations sont très spécifiques aux types de tâches, d'autres décrivent bien les points qu'il faut avoir en tête lorsque l'on cible les architectures NUMA. Il indique notamment l'importance de la distribution des données de l'application sur l'ensemble des nœuds NUMA, et l'importance de conserver la localité des données, lors du vol de travail.

Dans le cadre spécifique des tâches OpenMP, **Terboven et al. [Terboven 2012]** ont fait une étude du comportement général des tâches sur les architectures NUMA, et donnent des indications assez générales sur les bases pour aborder le problème, comme par exemple assurer le placement fixe d'un thread sur cœur de calcul, ou encore les types d'ordonnancement prévus dans le standard qui sont les plus efficaces dans ce cas. Cela a eu pour conséquence l'apparition dans OpenMP 4.0 des *places* et des stratégies de placement des threads (*proc bind*).

3.1.1 Groupement des calculs ensemble

Les techniques utilisées par les travaux suivants ont toutes en commun le groupement des calculs sur des éléments topologiquement proches dans la hiérarchie. L'objectif derrière cette idée est de "cloisonner" la distribution du travail (par exemple les itérations de boucles) afin de limiter les accès distants et encourager la réutilisation du cache.

Olivier et al. [Olivier 2012, Olivier 2013] font l'état des problèmes se posant lors de l'équilibrage de charge sur les architectures NUMA. Ils notent en particulier que s'il permet de limiter l'inactivité de certains cœurs, il entraine aussi une augmentation des cache miss ainsi qu'une augmentation des accès distants, plus coûteux, et donc une augmentation du temps d'exécution des tâches.

Leur principale contribution s'oriente autour d'une stratégie d'ordonnancement prenant en compte la hiérarchie de la mémoire au sein de l'architecture. Ils définissent un domaine de voisinage, qu'ils appellent *shepherd*, qui regroupe un ensemble de cœurs voisins dans la hiérarchie. Cet ensemble peut être défini à plusieurs niveaux : il peut être défini pour un seul cœur, pour un ensemble de cœurs partageant le même cache L3, ou pour un socket comprenant plusieurs processeurs multicœurs.

Ils proposent ensuite une extension au vol de travail : ils définissent un *shepherd* (et donc une file de tâches) par nœud NUMA, et ils proposent que chaque *shepherd* aille voler un ensemble de tâches lorsque leur propre file de tâches est vide. Au sein d'un *shepherd*, le travail est réparti entre les threads associés aux cœurs d'une manière LIFO, qui permet un ordonnancement en profondeur dans le but de rester au plus proche de l'exécution et de favoriser la réutilisation des caches.

Clet-Ortega et al. [Clet-Ortega 2014] proposent des techniques de vol de travail favorisant les vols locaux plutôt que distants. Ils constatent qu'une file centralisée introduit un fort coût de gestion compte tenu du fait que le nombre de tâches créées peut être élevé, et proposent l'utilisation d'une file de tâches par thread du support exécutif pour pallier ce problème. Cela introduit du même coup une vision décentralisée de

la gestion des tâches, et en couplant ça avec une analyse de la topologie via hwloc, ils introduisent la notion de voisins pour un thread.

La deuxième partie de leur contribution consiste à étendre le vol de travail pour privilégier les vols dans les files voisines, restreignant ainsi les calculs sur des groupes de cœurs physiques proches. Cette approche est reprise par Tahan et al. [Tahan 2014]. Chaque thread dispose d'une liste privée des autres files de threads, triée par la distance au cœur sur lequel s'exécute le thread.

Pilla et al. [Pilla 2014] proposent des extensions au support exécutif du modèle de programmation Charm++ [Kale 1993], ciblant spécifiquement les architectures NUMA et les applications itératives. À chaque pas de temps, leur ordonnanceur va tenter de corriger un déséquilibre de charge en déplaçant des tâches d'un cœur à un autre. Pour évaluer si une tâche devrait être déplacée sur un autre cœur, ils prennent en compte plusieurs facteurs : la charge du cœur, la quantité de communication que la tâche effectue avec d'autres tâches du cœur, le facteur NUMA entre les deux cœurs, et la quantité de communication locale que la tâche effectuerait sur son cœur actuel. Charm++ a l'avantage de leur fournir directement des informations sur les performances des tâches aux précédents pas de temps. Pour résumer ils vont favoriser le rééquilibrage de charge sur des cœurs voisins, et considérer un équilibrage sur un cœur distant si les caractéristiques de la tâche le permettent.

3.1.2 Distribution initiale des données et placement des calculs

Nous avons ici regroupé les techniques consistant à contrôler le placement de ses données, et à essayer d'ajuster au mieux l'ordonnancement pour que le calcul s'effectue proche des données. Cette approche assez naturelle vise d'une part à réduire la contention au sein d'un même contrôleur mémoire, mais également à minimiser les accès distants coûteux.

HPF [Koelbel 1994] (High Performance Fortran) illustre très bien ces idées. Cette extension de Fortran 90 ne concerne pas du tout les architectures NUMA, mais les concepts qui y sont développés ont servi d'inspiration pour nos travaux décrits dans le chapitre 5.

Le principe de cette extension est de distribuer la *possession* de chaque élément des tableaux manipulés sur les différents cœurs, et de répartir la charge de travail (les instructions ou groupes d'instructions) sur les cœurs qui possèdent les éléments que ces instructions manipulent. Ils proposaient également un ensemble complet de fonctionnalités pour contrôler le découpage des données.

Pousa Ribeiro et al. [Ribeiro 2009] propose une bibliothèque externe pour contrôler l'allocation de données sur les différents nœuds NUMA. Ils proposent plusieurs types de mécanismes :

- bind : un contrôle précis du nœud sur lequel doit être allouée la donnée
- cyclic : pour distribuer de manière cyclique les pages (ou groupes de pages) de la mémoire allouée sur un ensemble de nœuds.

- random : pour distribuer aléatoirement les pages (ou groupes de pages) de la mémoire allouée sur un ensemble de nœuds.

Contrairement à *numactl*, cette bibliothèque permet d'avoir une distribution basée sur les blocs de données effectivement manipulés par l'application. Elle peut s'utiliser soit via une modification du code, soit via un compilateur source à source.

Gautier et al. [Gautier 2007] propose un ordonnancement dynamique d'un graphe de tâches avec dépendances en deux étapes : dans un premier temps le graphe est partitionné et réparti sur le cluster, et dans un second temps l'équilibrage de charge est fait à l'aide du vol de travail. Une approche similaire, mais dans le cadre d'une approche purement *offline*, est utilisée par **Gustedt et al. [Gustedt 2017]** qui proposent une extension à ORWL [Clauss 2010]. Cette extension permet d'utiliser les informations sur l'application et la topologie de l'architecture pour d'une part déterminer une matrice de communication entre les tâches, et d'autre part déterminer une affectation automatique des ressources et des calculs associés sur la topologie.

Durand et al. [Durand 2013] abordent le thème de l'ordonnancement de boucles irrégulières sur machine NUMA. D'une part ils utilisent le concept de *domaine de voisinage* défini dans la section précédente pour avoir une vue hiérarchique de la machine, et assurent la distribution des données de l'application sur l'ensemble des *domaines de voisinage*. D'autre part ils étendent le support exécutif pour rendre les boucles *adaptatives* : lorsqu'un thread n'a plus d'itérations à exécuter, il peut aller voler un autre thread qui lui donnera une partie des itérations non traitées qui lui ont été attribuées.

Cette approche est intéressante dans notre contexte, car au delà de montrer qu'ils compensent l'irrégularité des itérations, ils montrent surtout l'importance de la distribution des données manipulées.

Wittman et Hager [Wittmann 2011] proposent une approche très intéressante compte tenu des constructions disponibles en 2011 dans le standard OpenMP (tâches indépendantes). La supposition de départ est une application dans laquelle chaque tâche lit et écrit un unique bloc de données. Ils proposent l'introduction d'une structure similaire au *domaine de voisinage* par nœud NUMA. Dans cette structure ils stockent une file de blocs de données.

Dans un premier temps le programmeur distribue équitablement l'ensemble des blocs de données dans l'ensemble des files des *domaines de voisinage*. Ensuite il va créer autant de tâches que de blocs de données existants. Lors de l'exécution, la tâche va déterminer dans quel *domaine de voisinage* elle se trouve, et manuellement retirer un bloc de données de la file correspondante pour effectuer le calcul.

Si cette approche a le mérite d'améliorer significativement la localité des données, certains aspects sont assez contraignants. D'une part le programmeur se retrouve à effectuer lui même des actions qui appartiennent typiquement au support exécutif, et pourraient être faites de manière transparente. Et d'autre part, aucun contrôle n'est fait pour s'assurer que le nombre de tâches sur chaque *domaine de voisinage* est le bon (seul le nombre global de tâches est assuré). Ils résolvent ce problème en permettant à une tâche d'aller voler un bloc de données dans un autre voisinage, si le sien est vide.

Al-Omairy et al. [Al-Omairy 2015] ont concentré leurs travaux sur l'inversion de matrices symétriques et la factorisation de Cholesky. Ils ont étendu le support exécutif et le modèle de programmation d'OmpSs afin de permettre de prendre en compte la distance aux données lors du vol de travail. Leur premier axe est de restreindre le vol de travail à l'intérieur du nœud : les threads ne peuvent voler que dans une file commune par nœud NUMA. Le second est de tracer la localité des données : ils utilisent des tâches d'initialisation pour détecter où sont allouées les données. Lors de l'exécution du programme cette information est utilisée pour ordonnancer une tâche sur le nœud où il y a le plus de données manipulées par la tâche.

Terboven et al. [Terboven 2016] ont effectués une première analyse d'un support de l'affinité pour les tâches OpenMP. Leur étude s'est portée sur deux points : l'affinité d'une tâche vers un thread ou une donnée, et l'affinité d'un groupe de tâches entre elles. Le premier point est très proches de notre proposition de clause affinité [Virouleau 2016b], et les deux articles ont tous les deux été présentés lors d'IWOMP 2016. Leur proposition d'affinité se base comme nous sur les réflexions du comité de standardisation, néanmoins certains points clés de l'implémentation divergent. Pour déterminer l'emplacement d'une donnée (et satisfaire la demande d'affinité), ils se basent sur le placement des threads de calcul (*OMP_PLACES*) et sur les précédentes références à la même donnée. Ils cherchent ensuite à regrouper les tâches utilisant les même réferences ensembles. De notre côté nous récupérons dynamiquement l'emplacement de l'allocation physique des données à l'aide d'un appel système. Une autre différence importante est que dans leur approche une affinité est *stricte* par défaut : seuls les threads situé sur la ressource possédant la donnée peuvent exécuter la tâche. Dans notre proposition l'affinité est un indice de placement que le support exécutif peut ignorer selon les besoins de l'ordonnancement, mais que le programmeur peut rendre obligatoire, rejoignant dans ce cas l'approche précédente.

En contexte hétérogène plusieurs travaux s'attaquent à l'ordonnancement de tâches en ciblant plusieurs GPU. Le problème de localité des données que l'on rencontre sur les architectures NUMA est encore plus critique lorsque l'on cible ce type d'accélérateur. Plusieurs travaux [Hermann 2010, Lima 2015, Augonnet 2011] utilisent une approche dérivée de HEFT, en utilisant des informations sur les performances estimées des tâches à ordonnancer pour guider le placement dynamique de ces tâches sur CPU ou sur GPU.

Bleuse et al. [Bleuse 2014] proposent un algorithme d'ordonnancement hétérogène prenant en compte l'affinité entre une tâche et ses données. Pour décider du placement d'une tâche prête, ils considèrent les données déjà présentes sur les différents GPUs, et placent la tâche dans la file du GPU sur lequel le plus de données sont présentes. Ils montrent une réduction significative de la quantité de données transférée au cours de l'exécution, ce qui se traduit sur certaines classes d'application par un gain important de performance comparé aux stratégies basées sur HEFT ou du vol de travail standard.

3.1.3 Migration dynamique des données et conservation de la localité

Dans cette dernière catégorie, les travaux effectués vont chercher à garantir la localité des données en regroupant les calculs, mais aussi en déplaçant les données manipulées par les calculs pendant l'exécution, pour réduire les accès distants. Cela peut être fait à deux niveaux : soit au niveau du support exécutif, soit au plus bas niveau, directement dans le noyau du système. Dans le premier cas on a potentiellement une meilleure connaissance des données pour faire de meilleurs choix, mais cela restreint le type d'application ou impose une action de la part du programmeur. Si l'on se place au niveau du noyau les informations ne sont disponibles qu'à l'exécution, et nécessitent une analyse. Cela a l'avantage de cibler n'importe quelle application sans la modifier, mais le potentiel de gain est un peu plus faible.

ForestGOMP [Broquedis 2010a] est un support exécutif ciblant spécifiquement les architectures NUMA. Ils constatent que le parallélisme que permet d'exprimer OpenMP 2.5 est plat, et que cela pose problème lors de l'exploitation des machines NUMA. Étant donné que le parallélisme imbriqué peut être vu comme une forme de hiérarchie, ils proposent de superposer ce type de parallélisme sur la topologie des machines NUMA. Pour cela ils regroupent les threads imbriqués en *bulles*, qui pourront ensuite être ordonnancés sur différents niveaux de la hiérarchie mémoire.

ForestGOMP repose sur deux composants :

- MARCEL : ce composant est en charge de l'ordonnancement des *bulles* sur la hiérarchie de la machine. Son objectif est de les placer au mieux, c'est-à-dire faire en sorte qu'elles soient exécutées par des cœurs d'un même nœud, et les déplacer si l'équilibrage de charge le nécessite.

- MaMI : ce composant gère le placement et la migration des données manipulées par les *bulles*. Il prend en charge une distribution initiale des données des différentes *bulles*, qui sera prise en compte par MARCEL lors du placement initial des *bulles*. Lors de l'équilibrage de charge, MaMI permet la migration des données associées à la *bulle* en marquant les pages manipulées, et en forçant le déplacement physique des pages proches du prochain thread qui touchera une page marquée (*next-touch*).

Ils montrent des gains significatifs de performance et une réduction de la contention sur des benchmarks populaires tels que STREAM [McCalpin 1995].

Drebes et al. [Drebes 2014] ont implémenté des stratégies d'ordonnancement ciblant les architectures NUMA dans OpenStream. Le support exécutif fonctionne par vol de travail (avec une file de tâches par cœur), et récupère des informations sur la hiérarchie mémoire lors de son initialisation. Les tâches expriment leurs dépendances de données sur des *streams* (implémentés par des *buffers*), dans lesquels plusieurs d'entre elles peuvent écrire ou lire des données. Ces dépendances permettent non seulement de déterminer l'ordre d'exécution des différentes tâches, mais également de connaître précisément l'ensemble des données manipulées par plusieurs tâches, ainsi que leur taille.

Leurs stratégies visent à améliorer la localité et la distribution des données au cours de l'exécution en jouant sur plusieurs facteurs.

- Lors de la sélection d'une file de tâches à voler, la stratégie va favoriser le vol sur les cœurs voisins.

- Lors du placement des tâches prêtes, la stratégie va déterminer le nœud sur lequel sont alloués ses *buffers* de données en entrée, et placer la tâche sur un cœur de ce nœud.

- La gestion de l'allocation des *buffers* des tâches est faite de manière à garder le maximum de données locales. Lorsque l'allocation d'un *buffer* est nécessaire (par exemple avant l'exécution d'une tâche qui écrit dedans), le support exécutif va regarder les différentes tâches qui contribuent à ce *buffer* ainsi que la taille de leur contribution. À partir de là ils définissent la notion de *dépendance forte* sur la tâche qui contribue le plus, et alloue le *buffer* sur le nœud où va s'exécuter cette tâche. Si la tâche à l'origine de l'allocation a été volée, l'allocation est faite localement et ignore la *dépendance forte* : cela permet de corriger des problèmes constatés sur des schémas de dépendances en forme d'arbre.

Ils montrent que la combinaison de ces facteurs permet d'augmenter significativement le rapport du nombre d'accès locaux sur le nombre d'accès distants, et ce sur des classes d'applications variées.

Diener et al. [Diener 2014] proposent d'attaquer le problème de la localité des données au sein même du système d'exploitation. Ils implémentent, au sein du noyau Linux, un mécanisme d'observation et d'analyse des accès effectués aux pages mémoire par les différents threads. À partir de ces observations, ils peuvent effectuer deux opérations : d'une part décider de migrer certaines pages mémoire sur un nœud différent, en modifiant directement la table des pages. D'autre part ils peuvent décider de migrer certains threads sur un nœud NUMA plus adapté. Ils montrent que la combinaison de ces deux opérations permet d'améliorer significativement les performances de certains benchmarks courants, sans avoir à effectuer de modifications au niveau du code source.

Des travaux similaires ont été effectués par Yu et al. [Yu 2017] sur des architectures disposant de mémoires hétérogènes.

Conclusion

Tous ces travaux ont eu un impact sur les idées que nous avons développés au cours de la thèse, mais ceux prenant en compte à la fois la hiérarchie mémoire, le placement des calculs par rapport aux données, et le contrôle du placement des données nous semblent les plus intéressants dans notre contexte.

Les travaux que nous mettons en avant dans le chapitre 5 se basent sur une combinaison des idées évoquées : nous suivons une approche où l'on tire parti des informations présentes concernant la hiérarchie de la machine, et où nous proposons de contrôler à la fois le placement des données, et à la fois la localité des données tout au long de l'exécution de l'application.

Contrairement aux travaux décrits dans cette section, nous avons accès à plus d'informations via les dépendances des tâches. Nous proposons également une clause supplémentaire pour permettre à l'utilisateur de fournir des informations explicites concernant l'affinité éventuelle entre les tâches et leurs données. Cela nous permet donc d'effectuer des choix pertinents vis à vis de l'application, en nous basant sur l'emplacement physique réel des données des tâches, que l'utilisateur peut également distribuer lors de l'initialisation de l'application via une autre clause spécifique.

3.2 Supports exécutifs

Il existe un certain nombre d'autres supports exécutifs, pour OpenMP comme pour d'autres modèles de programmation. Les sections ci-après introduisent ceux ayant des thématiques très proches de cette thèse.

3.2.1 XKaapi

Kaapi [Gautier 2007] est un support exécutif, à base de tâches avec dépendances, ciblant originellement les clusters de processeurs multicœurs. Il a ensuite été étendu pour cibler spécifiquement les architectures hétérogènes, et à au passage été renommé XKaapi [Gautier 2013]. Il repose sur hwloc pour découvrir la topologie de la machine, et utilise ces informations à de multiples endroits.

Le moteur d'ordonnancement de XKaapi fonctionne par vol de travail [Lima 2015], et implémente les étapes critiques de *sélection* et de *placement* décrites dans la section 2.4.2.2. Il est facile d'ajouter des heuristiques additionnelles pour ces deux étapes, ce qui nous a permis d'implémenter dans ce support exécutif les extensions décrites dans la section 5.3.2.

Le nombre de files de tâches repose sur les informations fournies par hwloc : XKaapi implémente une file de tâches par niveau de la hiérarchie (i.e. : une file par cœur, une file par nœud NUMA, etc.), qui sont éventuellement utilisées par les heuristiques.

Pour la gestion de ces files, XKaapi implémente le protocole THE [Frigo 1998] proposé par Cilk. Ce protocole permet de faire, dans la plupart des cas, des accès concurrents à la même file de tâches de manière non bloquante. Le principe est le suivant : le *voleur* - distant - va venir prendre des tâches en tête de file, et la *victime* (ou le thread local) va venir ajouter ou retirer des tâches en queue de file. Le seul conflit se produit lorsque la file n'a qu'un seul élément, se traduisant par la prise d'un verrou.

En plus de l'utilisation de ce protocole, XKaapi peut effectuer de l'agrégation de requêtes de vols : lorsque plusieurs voleurs vont effectuer des requêtes sur la même victime, seul le premier voleur arrivé va effectuer la requête de vol, et récupérer suffisamment de tâches pour l'ensemble des voleurs [Besseron 2009]. Les gains théoriques liés à ce mécanisme ont été étudiés par Tchiboukdjian et al. [Tchiboukdjian 2010].

Pour observer le comportement des applications exécutées, XKaapi dispose d'un outil de génération de traces. Cela permet une analyse pointue du comportement de l'application, via les compteurs de performance matériels et une analyse par type de tâche. Cet outil nous a permis de faire des observations préliminaires déjà très poussées, sur une étude de cas abordée dans la section 4.3.2.

XKaapi est principalement utilisé comme prototype de recherche, et a été utilisé pour l'implémentation de certains travaux proches des thématiques de cette thèse, en particulier celle de la localité des données [Durand 2013, Bleuse 2014, Lima 2015].

Du point de vue de la compatibilité avec OpenMP, elle a été progressive et plusieurs solutions existent ou ont existé : KaCC [Le Mentec 2011] était un compilateur source à source basé sur ROSE [Quinlan 2003] qui implémentait les dépendances par dessus les tâches indépendantes d'OpenMP 3.0, et permettait donc à XKaapi d'exécuter les programmes OpenMP à base de tâches. Lorsqu'Intel a commencé à travailler sur l'implémentation d'OpenMP 4.0 dans Clang, un nouveau compilateur source à source basé sur Clang a vu le jour : KStar. Ce compilateur, supporté par l'ADT Inria du même nom, avait pour but de cibler XKaapi et StarPU à partir des pragmas OpenMP. Une fois que le support d'OpenMP dans Clang a été stable, le support d'XKaapi dans KStar a été abandonné.

La manière actuelle de cibler XKaapi est d'utiliser une couche de compatibilité pour OpenMP au niveau binaire, nommée libKOMP [Broquedis 2012]. Cette couche implémente à la fois les ABIs de libGOMP et libOMP, ce qui permet de l'utiliser pour exécuter des programmes OpenMP 4.0 directement en compilant via GCC ou Clang, et en changeant le support exécutif chargé à l'exécution.

3.2.2 libGOMP

libGOMP [Novillo 2006] est le support exécutif OpenMP fourni avec le compilateur GCC.

Au niveau des fonctionnalités, il implémente la totalité du standard OpenMP 4.5. Comme la majorité des supports exécutifs, libGOMP réutilise les threads qui sont créés entre différentes régions parallèles successives, pour éviter d'avoir à payer le coût de destruction/création d'un thread inutilement. Les gestions des constructions à base de boucles et de tâches sont complètement séparées dans le support exécutif. Vis-à-vis de la hiérarchie de l'architecture cible, il n'y a aucune disposition particulière pour essayer de la prendre en compte.

Pour la gestion des tâches, libGOMP ne dispose que d'une seule file de tâches par *team* [libGOMP 2018], donc une seule file pour l'ensemble des threads, et l'ordonnancement est effectué à travers un algorithme glouton. Si fonctionnellemment cette caractéristique n'est pas un problème, cela peut avoir un impact sur les performances compte tenu du fait que tous les threads devront sérialiser leurs accès à la même struture de données. Cela se voit d'ailleurs sur la figure 2.4 illustrant l'impact de la granularité des tâches : pour des petites tailles de bloc (et donc un grand nombre de tâches), libGOMP est loin derrière à cause du surcoût entrainé par la gestion de la liste de tâches. Pour amoindrir ce problème, libGOMP limite le nombre de tâches total dans la file de tâches : si un thread essaie de pousser une tâche et que cette limite est atteinte, alors la tâche sera invoquée séquentiellement par le thread. En pratique le nombre limite de tâches est égal à 64 multiplié par le nombre de threads utilisés.

Néanmoins, en tant que support exécutif grand public et largement utilisé, il constitue une référence intéressante.

3.2.3 libOMP

libOMP est le support exécutif OpenMP fourni avec le compilateur Clang, directement basé sur le support exécutif d'Intel fourni avec ICC. Ils partagent donc exactement les mêmes caractéristiques.

Compte tenu du fait qu'il a été développé à la base par des développeurs d'Intel, une partie de ses fonctionnalités ont été motivées par l'exploitation du matériel produit par Intel comme le Xeon Phi.

De manière similaire à libGOMP, la gestion des boucles et des tâches est séparée, et les threads (et même les *teams* et leurs structures de données associées) sont réutilisés par les régions parallèles successives.

En revanche libOMP se distingue de libGOMP de par ses structures de données : chaque thread d'une *team* possède une file de tâches propre. Il fonctionne par vol de travail, et les heuristiques de base pour les fonctions d'ordonnancement sont les suivantes : la sélection d'une file à voler a lieu aléatoirement parmi les files de tâches disponibles ; lors de vols successifs, le voleur essaye en priorité la dernière file dans laquelle il a réussi à voler une tâche. Le placement des tâches prêtes a lieu dans la file du thread courant.

Comme dans libGOMP, si le nombre de tâches dépasse un certain seuil (dans ce cas précis, si la file de tâches est pleine), alors la tâche c'est pas réellement enfilée mais elle est directement exécutée via un appel séquentiel.

Bien que ce mécanisme n'ait pas été initialement conçu pour permettre d'interchanger des stratégies, cela proposait une base suffisamment solide pour accueillir les extensions que nous proposons dans le chapitre 5. Les modifications que nous avons apportées à ce support exécutif sont détaillées dans la section 5.4.1.

3.2.4 OmpSs

OmpSs [Duran 2011] est un modèle de programmation visant à étendre OpenMP, en particulier le support du parallélisme asynchrone (à base de tâches avec dépendances par exemple), et de l'hétérogénéité. La syntaxe et les détails dans l'utilisation peuvent être légèrement différents, mais les constructions et concepts restent les mêmes. OmpSs est composé d'un compilateur, *Mercurium*, et d'un support exécutif *Nanos++*.

Du point de vue de la gestion des tâches, Nanos peut fonctionner soit par un algorithme de liste, soit par vol de travail. Par défaut l'ordonnanceur fonctionne à l'aide d'une unique file de tâches à priorité, néanmoins certains ordonnanceurs fonctionnent avec une file de tâches par cœur. Il ne dispose pas d'ordonnanceur prenant en compte la localité des données, mais certains d'entre eux disposent de files de tâches associées à certains éléments de la hiérarchie (cœur ou nœud NUMA), afin de favoriser le vol de tâche «proche».

3.2.5 OpenStream

OpenStream [Pop 2013] est un modèle de programmation par flots de données dérivant directement d'OpenMP 3.0. Le programmeur définit des flots de données ainsi que des tâches opérant en lecture et/ou écriture sur une certaine quantité de don-

nées d'un flot (appelée *window*). Concrètement les flots de données peuvent être vus comme des tableaux, et les tâches opèrent sur un certain nombre d'éléments contigus de celui-ci. Le support exécutif étudie ensuite l'ordre d'écriture dans les différentes parties d'un flot pour construire un graphe de dépendance des tâches, qui sera ensuite ordonnancé sur la machine.

Ce modèle se rapproche donc très fortement des tâches avec dépendances qui sont apparues dans la version suivante d'OpenMP. OpenStream utilise un support exécutif avec des extensions pour les architectures NUMA, nous l'avons présenté plus en détails dans la section 3.1.3.

3.2.6 StarPU

StarPU [Augonnet 2011] est une librairie de programmation parallèle à base de tâches avec dépendances. Le programmeur décrit les tâches et leurs différentes implémentations (CPU, GPU, ...), ce qui permet à son support exécutif hétérogène permet de cibler aussi bien des processeurs standards que des accélérateurs. La seule restriction est qu'il n'est pas possible d'utiliser des tâches récursives.

StarPU utilise des techniques avancées d'ordonnancement sur ressources hétérogènes, et propose différentes techniques d'ordonnancement en fonction du but recherché. D'un point de vue performances, les ordonnancements de tâches disponibles peuvent être soit purement *online* (tel que le vol de travail - *ws*), ou dériver de techniques initialement *offline* comme leurs ordonnanceurs *dm*, où un ordonnancement initial similaire à HEFT est effectué.

3.2.7 QUARK

QUARK [Kurzak 2013] (QUeing And Runtime for Kernels) fut le support exécutif privilégié pour la bibliothèque d'algèbre linéaire PLASMA, qui utilise maintenant OpenMP, et dont certaines de nos applications sont adaptées.

Il fonctionne lui aussi à base de tâches, qui sont exclusivement des fonctions de l'utilisateur. La création de tâches se fait à l'aide d'appels au support exécutif, et en plus d'un pointeur sur la fonction tâche le programmeur indique les variables manipulées et le type d'accès effectué.

Cela permet donc à QUARK de déterminer un ordre d'exécution sur les tâches pour son ordonnancement. L'avantage principal de QUARK par rapport aux autres modèles de programmation similaires est qu'il propose des extensions spécifiques à certains algorithmes d'algèbre linéaire présents dans PLASMA.

3.3 Compilateurs et interopérabilité

Comme nous avons pu le voir dans la figure 3.1, il faut bien faire la distinction entre support exécutif et compilateur. Nous avons vu que les supports exécutifs sont cruciaux pour l'exécution d'applications parallèles, néanmoins pour faire la transition entre les applications et eux il faut passer par une étape de compilation.

Peu importe le modèle de programmation le principe reste le même : le code source va contenir des directives (`#pragma`) ou des appels de fonctions décrits par le modèle

de programmation. Le code source va ensuite être transformé en un binaire contenant un ensemble d'appels au support exécutif (l'*Abstract Binary Interface* ou *ABI*).

L'ABI est spécifique au support exécutif, a priori un compilateur est donc fortement couplé avec un support exécutif, bien qu'en pratique il existe des compatibilités que nous aborderons dans la section 3.3.2.

3.3.1 Un point sur l'état des compilateurs

Dans le cas spécifique d'OpenMP, nous allons nous intéresser à trois compilateurs populaires : GCC, ICC, et clang.

GCC : il est probablement le compilateur le plus populaire pour Linux. Il est open source, très utilisé, et donc amélioré en permanence ; ses développeurs ont toujours été très réactifs aux changements du standard OpenMP, et il implémente la dernière version du standard (OpenMP 4.5) depuis la version 6.1.

Pour ce qui est du support exécutif, GCC génère du code spécifiquement pour l'ABI de libGOMP.

ICC : C'est le compilateur propriétaire d'Intel ; étant donné que les développeurs d'Intel ont été très proactifs pour l'ajout du support d'OpenMP dans Clang, le compilateur supporte lui aussi la dernière version du standard (ainsi que quelques fonctionnalités de la version 5.0, la prochaine mouture du standard) depuis ICC 17.

Il génère du code spécifiquement pour l'ABI de son propre support exécutif open source (libIOMP [1]), qui correspond également à l'ABI utilisée par Clang.

ICC est généralement privilégié pour sa capacité à générer du code performant pour les architectures Intel, que ce soit pour des optimisations de vectorisation ou pour l'utilisation d'accélérateurs tel que le Xeon Phi (KNL).

Clang : Ce compilateur open source du projet LLVM reçoit des contributions régulières de la part d'entreprises majeures telles que Google, Apple, Intel, ou encore ARM. Contrairement à GCC, le support d'OpenMP est assez récent et a été ajouté d'un bloc. Des développeurs d'Intel ont d'abord ajouté un support partiel dans un clone de Clang : *clang-omp* [2]. Il y a ensuite eu un effort d'ingénierie pour l'inclure dans Clang, avec un changement de license du support exécutif pour être compatible avec l'infrastructure LLVM. Compte tenu de l'implémentation il génère du code utilisant l'ABI du support exécutif d'Intel, qui a par la même occasion été intégré dans le projet LLVM.

Il supporte entièrement la norme OpenMP 4.5 depuis sa version 3.9.

Le code source de Clang est aussi très bien documenté et facile d'accès, ce qui nous a conduit à le choisir comme base pour les extensions d'OpenMP que nous décrivons dans le chapitre 5.

[1] https://www.openmprtl.org/
[2] https://clang-omp.github.io/

3.3.2 Compatibilité

Nous venons de décrire un certain nombre de compilateurs et de supports exécutifs. Nous avons également vu qu'un compilateur génère du code spécifiquement pour une ABI qui est ensuite implémentée par le support exécutif, rendant les compilateurs et les supports exécutifs a priori incompatibles entre eux.

Pour un programmeur, le moyen le plus pratique de comparer les performances de différents supports exécutifs (pour un modèle de programmation donné), ce serait en fait de compiler le programme avec un compilateur donné, puis de pouvoir changer le support exécutif juste avant l'exécution (via un ajustement du `LD_PRELOAD` ou du `LD_LIBRARY_PATH`).

C'est aussi de l'intérêt des développeurs des supports exécutifs de les rendre le plus accessible possible. Il existe donc en fait plusieurs couches de compatibilité, que nous allons décrire ci-après.

Vis-à-vis des compilateurs décrits dans la section précédente, la compatibilité est quasi totale : Clang et ICC génèrent du code pour la même ABI, et libOMP implémente une couche de compatibilité entre l'ABI de libGOMP et libOMP. S'il est impossible d'utiliser libGOMP pour exécuter du code compilé par ICC ou Clang, toutes les autres substitutions sont possibles.

S'agissant des autres supports exécutifs : OmpSs dispose de son propre compilateur OpenMP source à source, *Mercurium*, pour cibler *Nanos++*. *KStar* est un compilateur OpenMP soure à source basé sur le frontend de Clang, permettant de cibler les supports exécutif XKaapi et StarPU. Ce dernier a été écrit lorsqu'il n'y avait pas de support officiel d'OpenMP dans Clang, et quand la génération de code pour les tâches avec dépendances n'était pas encore disponible dans le prototype d'Intel. Le support de XKaapi à travers KStar a été ensuite abandonné quand Clang a officiellement supporté OpenMP 4.0 en utilisant un support exécutif, basé sur celui d'Intel, libOMP. Ce support a été abandonné au profit de libKOMP, une couche de compatibilité avec les ABI de libGOMP et libOMP. Elle permet, une fois le programme compilé à l'aide de Clang, ICC, ou GCC, de substituer le support exécutif par défaut pour utiliser libKOMP, en changeant la bibliothèque dynamique chargée.

Conclusion

Cette section a décrit l'ensemble des techniques et outils qu'il nous semble important de connaître afin de cerner le cadre de cette thèse. Un nombre important des articles évoqués concerne le support exécutif. Nous avons effectués plusieurs extensions et modifications de deux supports exécutifs : XKaapi et libOMP. Les détails sont présentés dans le chapitre 5.

Avant de rentrer dans ces détails, nous allons en premier lieu nous intéresser aux machines et applications que nous allons utiliser, et en particulier voir comment nous pouvons les caractériser pour identifier les informations et paramètres les plus pertinents à prendre en compte lors de l'ordonnancement. Le chapitre 4 présente dans un premier temps CarToN - ***Cha**racterization **To**ol for **N**UMA Architecture*, ainsi qu'une mise en application sur nos machines d'expérimentation (en section 4.2), et sur une étude de cas d'application (en section 4.3).

Partie II

Étude approfondie des machines NUMA, et amélioration de leur utilisation à travers OpenMP

4

Caractérisation des architectures NUMA

Les chapitres précédents ont décrit l'existant en terme de matériel et de logiciel. Ce chapitre va se concentrer sur l'analyse et la caractérisation des architectures NUMA et de l'exécution d'application sur celles-ci. La première section fait la description d'un outil, CarToN, que nous avons développé dans le but de faciliter cette opération. La section 4.2 fait une description et une étude, à l'aide de CarToN, des deux machines expérimentales que nous avons utilisées dans nos expériences. La section 4.3 montre l'utilisation de CarToN sur une étude de cas : la factorisation de Cholesky. L'objectif de ce chapitre est de dégager un axe d'amélioration possible du support exécutif à partir des constats fait sur les applications et les machines.

4.1 Exécution précise de noyaux

Cette partie se concentre sur la présentation de CarToN, un outil que nous avons créé afin de faciliter la réalisation d'expériences avec un contrôle précis sur le placement de noyaux à exécuter ainsi que leurs données. Le but de ces expériences est d'analyser le comportement des parties de calcul critiques aux performances d'une application, sur une architecture donnée.

4.1.1 Besoins pour un outil spécifique: CarToN

Il est en général assez facile d'étudier le comportement global d'une application, et d'observer les variations de ce comportement lorsqu'on change certains détails dans l'exécution, comme par exemple via l'utilisation de *numactl*.

En revanche, si l'on connait bien son application, on a envie de pouvoir étudier le comportement précis de certaines de ses parties critiques afin de pouvoir identifier ce qui cause son comportement global.

La suite naturelle de cette identification est de déterminer s'il existe des améliorations possibles pour ce comportement local, et comment l'améliorer en pratique.

Dans le cas d'une application à base de flots de données, chaque partie de l'application est bien identifiée, et correspond à un nœud dans le graphe de tâches. Toutes les données manipulées par une partie de l'application sont facilement identifiées également, puisqu'il s'agit des connexions entre les nœuds du graphe de tâches.

Dans le contexte d'une machine NUMA, le temps d'exécution d'une tâche dépend à la fois du placement de son exécution, ainsi que du placement de ces données. On a donc envie de pouvoir étudier le comportement individuel de chaque type de tâche en fonction de son placement et du placement des données qu'elle accède. Une fois cela fait, cela permettra d'identifier des potentielles variations de comportement, et ajuster les heuristiques d'ordonnancement pour prendre en compte ces variations.

Nous n'avons pas trouvé d'outil existant répondant précisément à ce besoin. BOAST [Videau 2017] est un outil assez proche dans la thématique d'optimisation de noyaux applicatif : l'utilisateur fourni des noyaux et un ensemble d'optimisations possibles, BOAST en déduit un espace à explorer et va rechercher automatiquement les paramètres optimaux.

Dans notre cas l'objectif n'est pas d'optimiser le code, mais de replacer le noyau applicatif dans une série de situations se présentant au cours de l'exécution réelle de l'application, afin de déterminer des stratégies d'ordonnancement qui pourrait favoriser l'exécution de ce noyau applicatif dans de bonnes conditions. De fait nous avions également besoin d'un outil qui nous permette de contrôler et faire varier le placement des noyaux et de ses données sur la topologie de la machine, et qui permette une exécution simultanée de noyaux applicatifs.

C'est à ce besoin que répond CarToN: une fois que l'utilisateur a isolé les parties critiques de son application, CarToN lui permet de définir lesquelles il souhaite exécuter et où, et lui garantir cette exécution, avec un certain nombre de variables observables.

Dans la suite de ce chapitre, on appellera l'ensemble des paramètres décrivant cette expérience un *scenario*.

4.1.2 Description d'un scenario

Ce que l'on appelle ici un *scenario* n'est ni plus ni moins que la description d'une expérience. Par exemple on pourrait vouloir "observer les performances en gigaflops d'une multiplication de matrices carrées sur le cœur 0 d'une machine". C'est un scénario simple, et l'exemple que l'on prendra pour illustrer les points un peu plus formel qui vont suivre.

En pratique un scénario est défini par les éléments suivants :

- Un ensemble de données et variables ;
- Une liste d'actions à effectuer ;
- Un ensemble de caractéristiques à observer.

Il est important que le format de description d'un scénario soit humainement lisible, et ne conduise pas à une recompilation systématique du programme. C'est donc une description en YAML [Ben-Kiki 2009] qui a été choisie.

Modèle d'exécution. Le flot d'exécution est le suivant : Dans un premier CarToN commence par charger le scénario fourni par l'utilisateur, créer les différentes données, et analyser les actions pour déterminer l'ensemble des cœurs physiques qui seront utilisés au cours des actions. Pour chacun des cœurs utilisés, un thread est créé et attaché à ce cœur. De plus, une file d'actions (FIFO) est créée pour ce thread. Les actions sont poussées dans les files d'actions correspondantes, dans l'ordre du fichier. Une fois l'ensemble des actions chargées, les threads exécutent chacun les actions présentes dans leur file, et vont déclencher les mesures de chaque paramètre observé avant et après chaque action.

Une unique primitive de synchronisation est disponible, sous la forme d'une action prédéfinie à insérer dans le scénario : une barrière sur un ensemble arbitraire de cœurs.

L'architecture de l'outil est donc simple, avec peu de logique relative au contrôle de l'exécution des tâches et à la synchronisation, ce qui permet de minimiser le «bruit» lors des expériences.

Les sections suivantes reviennent sur les différentes caractéristiques définissant un scénario, avec des exemples concrets d'utilisation.

4.1.2.1 Données et variables

Elles sont indispensables car c'est là-dessus que vont se baser les actions du scénario.

L'utilisateur doit fournir les noms et types des variables utilisées en paramètre des différents noyaux, elles peuvent être réutilisées par différents noyaux.

CarToN ne prend pas en charge l'allocation ou l'initialisation des données. Dans le cas de variables simples comme des constantes, elles peuvent être directement affectées dans le scénario. Dans le cas de variables complexes, l'utilisateur doit déclarer une action d'initialisation (avec ses paramètres) dans CarToN, pour pouvoir l'utiliser ensuite dans le scénario pour initialiser les données. Cette action peut être soit une fonction implémentée comme un module de CarToN, soit être un point d'entrée dans une bibliothèque externe.

CarToN met néanmoins à disposition une fonction d'allocation ne faisant aucune réutilisation de page, avec une politique explicite d'allocation physique des pages en *first-touch*.

Pour revenir à l'exemple du scénario simple ou l'on souhaite exécuter une multiplication de matrices carrées - `dgemm` - sur un cœur donné, nous avons besoin de trois matrices `a`, `b`, et `c`, ainsi qu'une largeur en nombre d'éléments pour les matrices manipulées, `size`.

Voici concrètement à quoi ressemblerait la déclaration de ces données :

Listing 4.1: Exemple de déclaration de variables

```
data:
  - a:
    - type: "double *"
  - b:
    - type: "double *"
  - c:
    - type: "double *"
  - size:
    - type: "int"
    - value: 256
```

Nous pouvons voir ici que `size` est initialisée directement, mais les pointeurs `a`, `b`, et `c` seront initialisés plus tard par une action.

Déclarations multiples. Il est possible de créer un ensemble de variables en se basant sur un modèle paramétré de nom. Un nom de variable peut contenir un ensemble de caractères (`[a-zA-Z]+`) encadré par des chevrons (<>). Cet ensemble de caractères est interprété comme un paramètre du nom de la variable, et la déclaration devra alors contenir l'ensemble des valeurs que ce paramètre peut prendre. L'ensemble des valeurs est soit : un ensemble d'entier désignant, à la manière d'une boucle, les limites de l'espace d'itération sous la forme `[début, fin, pas]` ; ou bien un ensemble de chaines de caractères qui seront substituées dans le nom de la variable.

L'exemple suivant illustre cette syntaxe en déclarant les variables `a0`, `a2`, `a4`, `a6`, `array_input`, et `array_output` :

Listing 4.2: Exemple de déclaration de variables paramètrées

```
data:
  a<i>:
    i: [0, 7, 2]
    type: "double *"
  array_<name>:
    name: ["input", "output"]
    type: "int *"
```

4.1.2.2 Actions

C'est là où l'utilisateur décrit effectivement les noyaux exécutés au cours du scénario. Il indique une série d'actions à exécuter, et avec quels paramètres.

Définition d'une action. Les actions peuvent être prédéfinies par CarToN, ou encore être implémentées par l'utilisateur comme un module de CarToN, ou finalement être un point d'entrée dans une bibliothèque. Le listing 4.3 donne un exemple d'une définition d'action qui serait externe à CarToN, où l'on ciblerait la fonction `cblas_dsyrk` de la bibliothèque OpenBLAS.

Listing 4.3: Exemple de définition d'une action externe

```
declare_actions:
  - library: "openblas"
    # Identifiant de l'action
    name: "dsyrk"
    # Point d'entrée dans la bibliothèque
    entry_point: "cblas_dsyrk"
    params:
    - "int"
    - "int"
    - "int"
    - "int"
    - "int"
    - "double"
    - "double *"
    - "int"
    - "double"
    - "double *"
    - "int"
```

Utilisation dans un scénario. Un scénario doit contenir une entrée `actions`, qui est le tableau d'actions à réaliser au cours du scénario. Chaque action peut avoir les caractéristiques suivantes :

kernel : chaine de caractères identifiant l'action. Cet attribut n'a pas de valeur par défaut et est obligatoire.

core : nombre entier indiquant le cœur sur lequel exécuter l'action. Il n'a pas de valeur par défaut, et est obligatoire.

params : liste de variables à passer en paramètre de l'action, leur nom doit correspondre à des données déclarées dans la section précédente.

repeat : nombre entier indiquant le nombre de fois que cette action doit être répétée. La valeur par défaut est 1.

CarToN dispose d'une action prédéfinie : la barrière. Son identifiant est `barrier`, et pour cette action spécifiquement le traitement de l'attribut `core` est un peu différent : il peut ne pas être spécifié, et auquel cas la barrière sera effectuée sur l'ensemble des cœurs, mais il peut également être un ensemble de cœurs pour restreindre la portée de la barrière.

Si on continue à décrire l'exemple simple d'une multiplication de matrices carrées, il faut que l'on effectue les actions suivantes : l'initialisation de chaque matrice (via une action `init_blas_bloc`, implémentée au préalable par l'utilisateur dans CarToN, qui prend en paramètre un pointeur et une largeur de matrice), et le lancement du dgemm une fois que ces matrices sont initialisées. Afin d'avoir une mesure plus précise du comportement du noyau, on peut indiquer une répétition du noyau, ici on choisit 50 pour l'exemple.

Le listing 4.4 montre un exemple réalisant ce scénario.

Listing 4.4: Exemple d'actions à réaliser pour une multiplication de matrice

```
actions:
  - kernel: init_blas_bloc
    params:
    - a
    - size
    core: 0
  - kernel: init_blas_bloc
    params:
    - b
    - size
    core: 0
  - kernel: init_blas_bloc
    params:
    - c
    - size
    core: 0
  - kernel: dgemm
    params:
    - a
    - b
    - c
    - size
    core: 0
    repeat: 50
```

Ici les trois initialisations et le calcul ont lieu sur le même cœur, et sont déroulés dans l'ordre de création, il n'y a donc pas lieu d'utiliser une synchronisation.

Actions paramétrées De manière similaire à la déclaration de données, il est possible de paramétrer les actions. Cela peut être particulièrement pratique dans le cas où l'on souhaite observer l'exécution concurrente de plusieurs noyaux de calcul.

Cela se fait à l'aide d'une action spéciale *for*, qui doit définir plusieurs attributs : `var`, qui contient le nom de la variable d'itération, et `actions`, qui contient la liste des actions à effectuer. Pour définir l'espace d'itération, l'utilisateur doit spécifier un et un seul des deux attributs suivants : `limits`, qui s'exprime sous la forme `[début, fin, pas]` et permet d'exprimer les limites de la boucles ; ou `values`, qui indique une liste explicite des valeurs que peut prendre la variable.

Le listing 4.5 fait une utilisation de cette syntaxe pour exécuter 4 dgemm simultanément sur les cœurs 0, 1, 2, et 3, en ayant au préalable initialisé les données nécessaires.

Listing 4.5: Exemple de déclaration d'actions synchronisées

```
data:
  # Déclaration de a0, a1, a2, a3, etc
  a<i>:
    i: [0, 3, 1]
    type: "double *"
  b<i>:
    i: [0, 3, 1]
    type: "double *"
  c<i>:
    i: [0, 3, 1]
    type: "double *"
actions:
  # Initialise a0, b0, et c0 sur le coeur 0,
  # a1, b1, et c1 sur le coeur 1, etc.
  - for:
      var: name
      values: ["a", "b", "c"]
      actions:
        - for:
            var: i
            limits: [0, 3, 1]
            actions:
              - kernel: init_blas_bloc
                params:
                - <name><i>
                - size
                core: <i>
  # Synchronisation avant de lancer les dgemm
  - kernel: barrier
  # Lancement d'un dgemm sur le coeur 0 utilisant a0, b0, et c0,
  # et d'un dgemm sur le coeur 1 utilisant a1, b1, et c1, etc.
  - for:
      var: i
      limits: [0, 3, 1]
      actions:
        - kernel: dgemm
          params:
          - a<i>
          - b<i>
          - c<i>
          - size
          core: <i>
          repeat: 50
```

Cette syntaxe permet d'exprimer des scénarios relativement complexe de manière compacte. En revanche simplement exécuter ces actions ne nous donnera pas grand chose, il faut donc définir un ensemble de caractéristiques à observer pendant leur exécution.

4.1.2.3 Observateurs

CarToN utilise des *Observateurs* pour enregistrer certaines caractéristiques au cours de la vie du programme. Un observateur peut avoir plusieurs attributs :

name : un identifiant d'observateur existant dans CarToN.

params : les paramètres à passer lors de la création de l'observateur.

kernels : une liste d'identifiants d'actions sur lesquelles appliquer cet observateur.

CarToN propose de base deux observateurs élémentaires :

- *time* : le temps passé dans l'action (en millisecondes).
- *papi* : permettant de relever des compteurs de performances à travers PAPI.

L'utilisateur peut implémenter lui-même des observateurs additionnels au sein de CarToN. Dans notre cas nous avons implémenté des observateurs spécifiques à certains noyaux d'algèbre linéaire, qui dérivent du temps passé dans l'action et qui indique la performance équivalente en Gflops.

La figure 4.6 illustre à quoi ressemblerait la section du scénario si nous souhaitions observer la performance de dgemm en Gflops, le nombre de cycles, ainsi que le nombre de *cache miss* de niveau 3 pendant l'exécution de chaque dgemm.

Listing 4.6: Exemple de déclaration d'observateurs

```
watchers:
  - name: flops_dgemm
    # Le nombre de flops dépend de la taille de la matrice,
    # qu'il faut donc donner en paramètre.
    params:
      - size
    kernels:
      - dgemm
  - name: papi
    params:
      - PAPI_TOT_CYC
      - PAPI_L3_TCM
    kernels:
      - dgemm
```

L'ensemble des compteurs à observer étant passé tel quel à PAPI, il est donc de la responsabilité de l'utilisateur de fournir un ensemble de compteurs compatibles entre eux.

L'observation se faisant sur la base d'une seule action, une ligne récapitulative est générée à partir des données des observations. Si l'utilisateur a indiqué une action avec un `repeat` de 50, il y aura donc 50 lignes avec les valeurs récoltées pour chaque action.

4.1.2.4 Notes d'implémentation

La syntaxe décrite et les fonctionnalités décrites ici sont celles que devraient contenir CarToN une fois terminé. Pour des raisons de temps, un certain nombre de fonctionnalités n'ont pas encore été implémentées : les syntaxes paramétrées (déclaration de variables et d'actions paramétrées) ; les actions utilisateurs chargées depuis des bibliothèques externes (seules les actions implémentées comme des modules sont utilisables) ; et le filtrage des observateurs par action (le filtrage a lieu en dur dans le code pour le moment).

4.1.3 Application et exemples de scénarios

CarToN nous a servi dans deux types de contexte : pour la caractérisation des machines sur lesquelles nous avons effectuées nos expériences, et pour l'étude détaillées des parties critiques des applications que nous avons utilisées.

La section 4.2 dresse un profil détaillé des machines et présente une utilisation de CarToN pour mesurer certaines caractéristiques de la machine, qui n'aurait pas été facilement mesurable à travers d'autres outils. La section 4.3 présente une étude de cas de l'une des applications que nous avons utilisé : la factorisation de Cholesky. Elle revient sur le fonctionnement de l'outil, les observations préliminaires que nous avons effectué, et la valeur ajoutée qu'a eu CarToN dans la compréhension détaillée et l'amélioration des performances de l'application.

4.2 Présentation et caractéristiques des machines

Nous présentons dans cette section deux machines NUMA de générations et de caractéristiques différentes. La première, *idchire*, est basée sur des processeurs Intel Sandy Bridge, et possède un nombre important de nœuds NUMA. La seconde, *brunch*, est basée sur des processeurs Intel plus récents de la génération Broadwell, et dispose d'un nombre assez faible de nœuds NUMA.

4.2.1 idchire

La machine idchire a été fabriquée par SGI, modèle UV 2000 [SGI 2012]. Elle est équipée de 24 processeurs Intel(R) Xeon(R) CPU E5-4640 (Sandy Bridge), cadencés à 2.4 GHz. Chacun de ces processeurs est associé à 31 Go de RAM pour former un nœud NUMA, et dispose de 8 cœurs physiques partageant 20 Mo de cache L3 (20-ways associatif). Chacun des cœurs a accès à 32 Ko de cache L1 (données) et 256 Ko de cache L2 (8-ways associatif). Les latences et bandes passantes relatives à chacun des niveaux de cache sont présentées dans le tableau 2.1.

La machine entière dispose donc de 192 cœurs physiques, et de 744 Go de RAM. Les processeurs Sandy Bridge disposent de l'extension vectorielle AVX, permettant

d'effectuer 4 additions et 4 multiplications de nombres flottant à double précision en un cycle, portant le pic de performance théorique de la machine à 3.6 TFLOPs.

4.2.1.1 Topologie

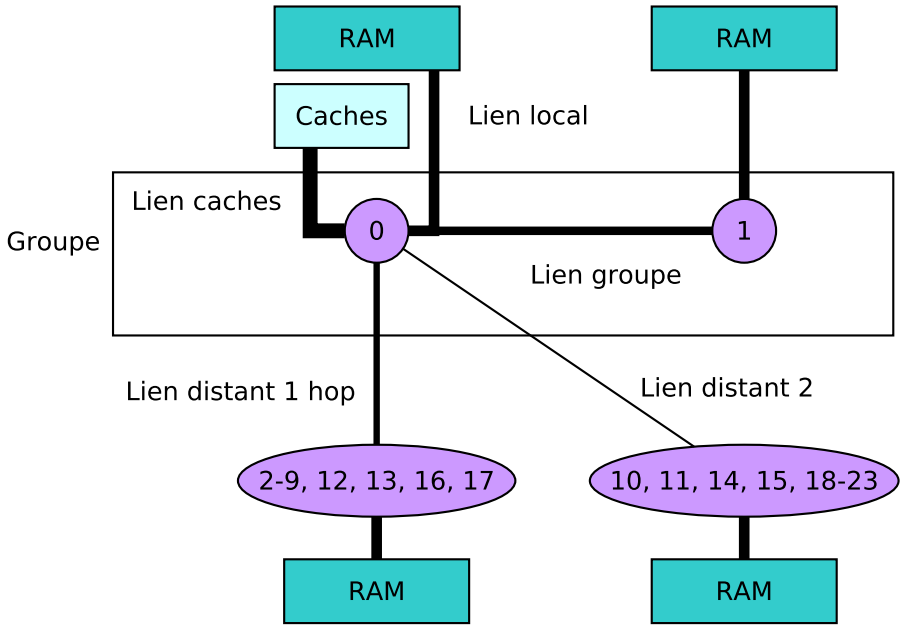

Figure 4.1: Topologie schématique vu du nœud 0

L'interconnexion des nœuds NUMA est effectué à travers l'Intel *Quick Path Interconnect* (QPI). La topologie de la machine expose une hiérarchie à plusieurs niveaux, la Figure 4.1 présente la hiérarchie de la machine du point de vue du nœud 0. Chaque nœud est d'abord associé à un autre nœud pour former un groupe. Ces groupes sont ensuite interconnectés entre eux et sont accessibles en deux rebonds maximum dans le système d'interconnexion. Pour chaque nœud il y a 12 nœuds situés à un rebond, et 10 nœuds situés à deux rebonds.

La Figure 4.2 présente la bande passante nœud à nœud, en fonction du nœud source et du nœud destination. La mesure est via une copie de tableau (`memcpy`) de 200 Mo, entre un tableau alloué et initialisé sur le nœud source, et un tableau alloué sur le nœud destination. Elle fait apparaître clairement une diagonale où la bande passante est significativement plus grande, illustrant le coût d'un accès mémoire local comparé à un accès distant. On peut également constater que la bande passante point à point est symétrique.

Vu du nœud 0 et en point à point, on a donc 4 niveaux de bande passante en fonction du lien utilisé : 3.15 Go/s pour le lien local, 1.27 Go/s pour le lien du groupe, 1.12 Go/s pour le lien distant avec 1 hop, et enfin 1.0 Go/s pour le lien distant avec 2 hops.

Ces mesures permettent de quantifier la pénalité d'un accès distant, mais la simple «carte» obtenue ne suffit pas à caractériser complètement les temps d'accès aux nœuds

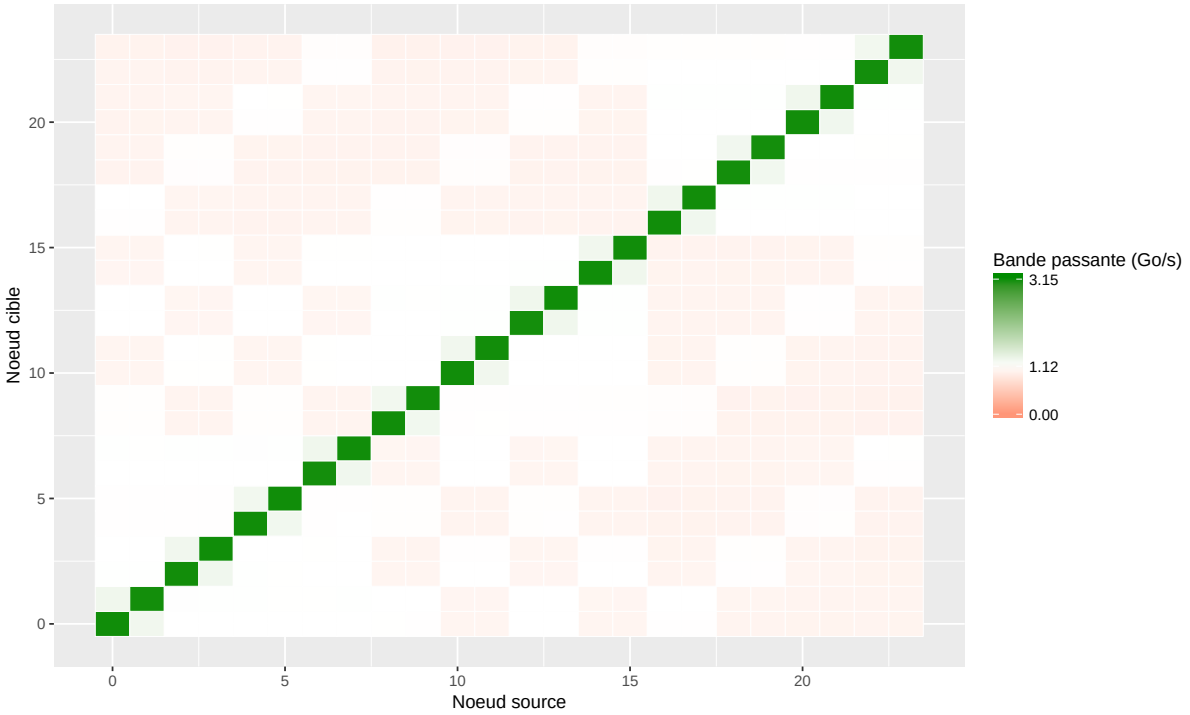

Figure 4.2: Carte de la bande passante d'idchire

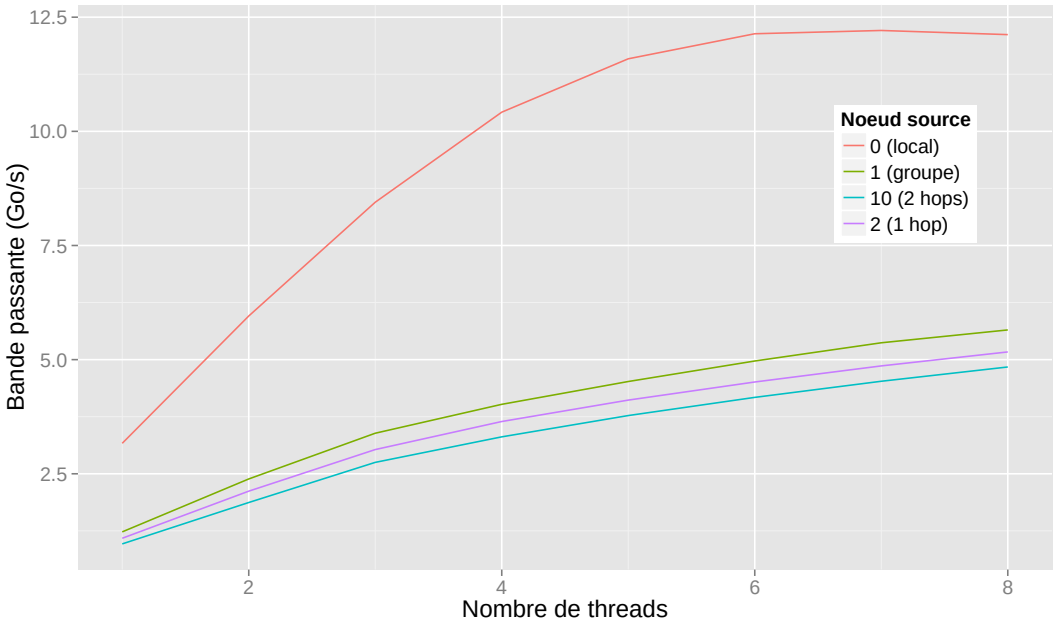

Figure 4.3: Bande passante cumulée vers le nœud 0, en fonction du nombre de threads effectuant une copie et du nœud source

NUMA, puisque qu'une seule communication ne va pas saturer la bande passante totale disponible, ni même illustrer l'impact de la contention.

4.2.1.2 Mesure des liens

En dehors de l'accès aux caches locaux, il y a 4 liens à quantifier, identifiés sur la Figure 4.1. Afin de mesurer la largeur de chacun des liens, nous avons défini des scénarios spécifiques pour mesurer la bande passante en entrée du nœud 0, en fonction du nœud distant possédant la donnée.

Le scénario est le suivant : deux tableaux de 200 Mo sont alloués, un sur le nœud 0, et un qui est alloué et initialisé sur un autre nœud, qui est le nœud source. La seule action est une copie du tableau distant (via `memcpy`), depuis le nœud source vers le nœud 0, et exécutée sur un cœur du nœud 0.

Étant donné qu'il y a 8 cœurs sur le nœud 0, nous avons effectué jusqu'à 8 copies simultanées (effectuées sur des tableaux indépendants).

L'annexe A.1 donne l'exemple du scénario utilisé pour la saturation du lien «groupe», qui effectue de 1 à 8 copie simultanées depuis le nœud 1 vers le nœud 0, et qui a permis de générer la courbe verte dans la figure 4.3 (nœud source 1).

La figure 4.3 regroupe les résultats de la bande passante cumulée, en fonction du nombre de copies simultanées ayant lieu, et en fonction du nœud d'où provient les données. Faire les copies depuis le nœud 0 permet de quantifier la largeur du lien local (qui atteint au maximum 12.2 Go/S) ; les copies depuis le nœud 1 permettent de quantifier le lien groupe (qui plafonne à 5.6 Go/s) ; les copies depuis le nœud 2 permettent de quantifier le lien distant avec 1 hop (au maximum 5.1 Go/s) ; et enfin les copies depuis le nœud 10 permettent de quantifier le lien distant avec 2 hops, qui atteint au maximum 4.8 Go/s.

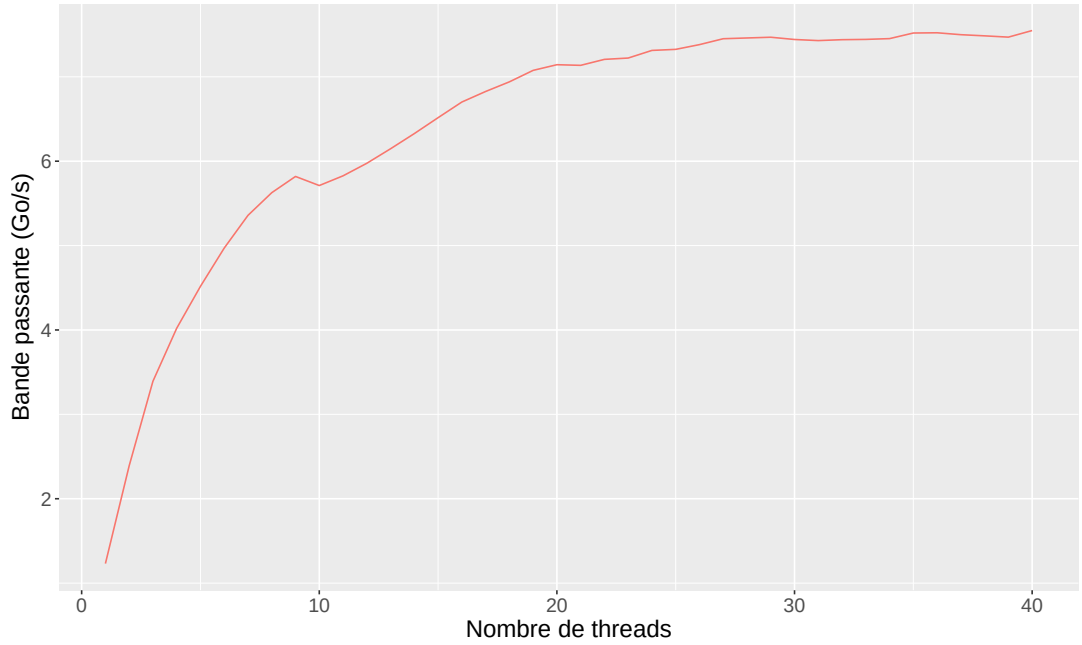

Figure 4.4: Bande passante cumulée depuis le nœud 0 vers plusieurs nœuds distants, en fonction du nombre de threads distants effectuant une copie

Pour terminer la caractérisation de ces liens, nous avons finalement mesuré la bande passante maximale que l'on peut obtenir en «sortie» d'un nœud, en saturant

les différents liens. Pour ce faire nous allons effectuer simultanément plusieurs copies d'un tableau de 200 Mo alloué et initialisé sur le nœud 0, vers un tableau alloué sur un nœud distant. Le thread effectuant la copie est placé sur le nœud distant.

La figure 4.4 montre la bande passante cumulée lors d'une copie du nœud 0 vers plusieurs nœuds distants, en fonction du nombre de threads effectuant une copie. Pour cette expérience la manière de placer les threads sur les nœuds distants est importante : les threads sont d'abord placés de manière à remplir progressivement le nœud 1 (nombre de threads de 1 à 8), qui passent par le lien groupe ; puis ce sont les nœuds 2, 3, et 4 qui sont remplis progressivement (threads 9 à 32), qui passent par le lien distant 1, et enfin le nœud 10 est rempli (threads 33 à 40), qui passent par le lien distant 2.

La bande passante maximale en sortie de nœud est donc de 7.5 Go/s, et commence à saturer aux alentours d'une vingtaine de copies distantes.

4.2.2 brunch

Cette machine est équipée de 4 processeurs Intel(R) Xeon(R) CPU E7-8890 v4 (Broadwell), cadencés à 2.2 GHz.

Chacun de ces processeurs est associé à 378 Go de RAM pour former un nœud NUMA, ils disposent de 24 cœurs physiques partageant 60 Mo de cache L3 (20-ways associatifs). Chacun des cœurs a accès à 32 Ko de cache L1 (données) et 256 Ko de cache L2. Les latences et bandes passantes relatives à chacun des niveaux de cache sont présentées dans le tableau 2.1.

La machine entière dispose donc de 96 cœurs physiques, et de 1.5 To de RAM. Les processeurs Broadwell disposent d'instructions FMA [1], permettant d'effectuer 8 additions et multiplications de nombres flottants à double précision en un cycle, portant le pic de performance théorique de la machine à 3.3 TFLOPs.

4.2.2.1 Topologie

L'interconnexion des nœuds NUMA est effectuée à travers l'Intel *Quick Path Interconnect* (QPI). Contrairement à idchire, la topologie de la machine est relativement plate : les nœuds sont directement connectés les uns aux autres, et seule la notion d'accès distant ou local permet de distinguer une hiérarchie. La topologie complète de la machine est représentée sur la figure 4.5

La Figure 4.6 présente la bande passante nœud à nœud en fonction de la source et de la destination, mesurée à l'aide d'une copie de tableau (`memcpy`) de 200 Mo. Bien qu'une diagonale se dégage clairement, la différence entre accès local et accès distant n'est de l'ordre que de 10% : la bande passante point à point pour l'accès local est d'environ 3.6 Go/s, et d'environ 3.3 Go/s pour le point à point sur un nœud distant.

4.2.2.2 Mesure des liens

De même que pour idchire, nous avons effectué des observations complémentaires pour caractériser plus précisément les liens locaux et distants, afin de déterminer leur saturation. Les résultats de ces expériences ont été rassemblés sur la figure 4.7. Il

[1] *Fused Multiply-Add*, permettant d'effectuer une addition et une multiplication en une étape

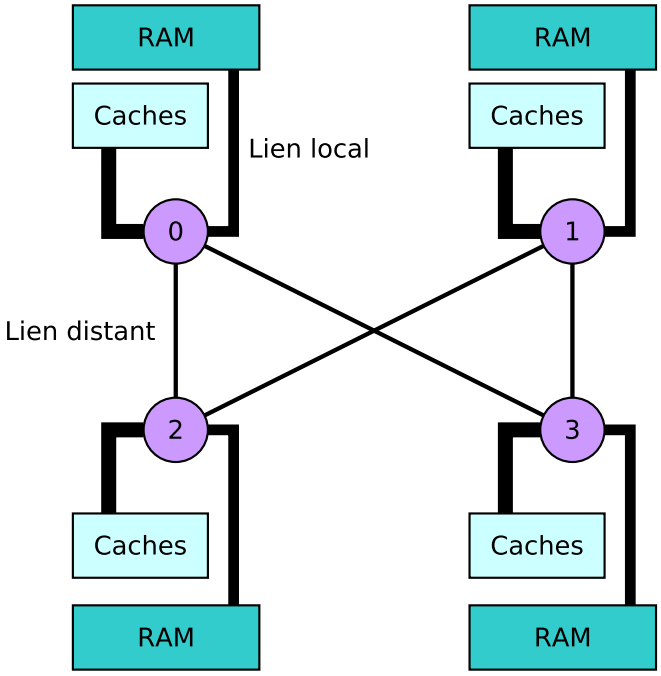

Figure 4.5: Topologie schématique complète de brunch

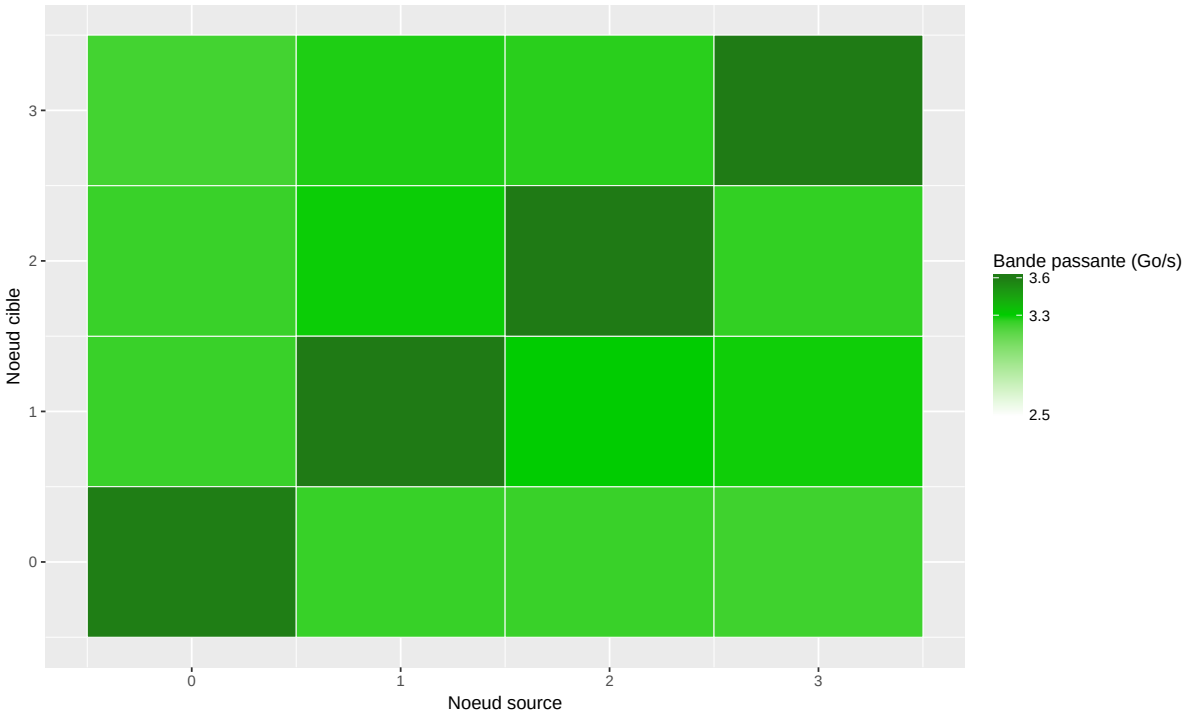

Figure 4.6: Carte de la bande passante de brunch

s'agit de la même approche que pour la figure 4.3, et nous avons effectué des copies simultanées depuis un nœud source vers le nœud 0.

En plus de constater que la bande passante cumulée est bien plus importante que

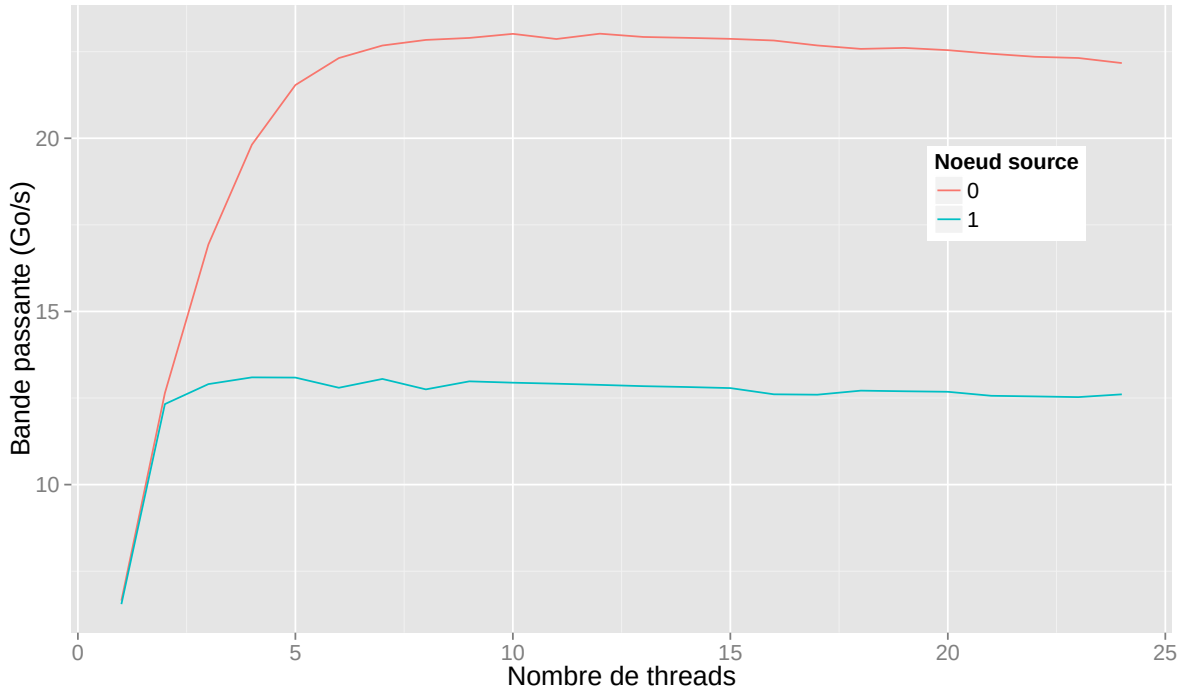

Figure 4.7: Bande passante cumulée vers le nœud 0, en fonction du nombre de threads effectuant une copie et du nœud source

sur idchire, on peut voir que les liens arrivent à saturation avec le même nombre de copies simultanées (environ 6), même si proportionnellement ça ne représente que 20% d'utilisation du nœud.

Ces résultats sur les capacités physiques des machines vont avoir un impact important dans la section suivante : nous allons faire une étude de cas d'une application - la factorisation de Cholesky, et nous allons en étudier individuellement les parties critiques. Certaines de ces parties critiques peuvent utiliser un large ensemble de données, en pleine charge de la machine les performances seront donc limitées par les résultats que nous venons de décrire.

4.3 Une étude de cas : Cholesky

Afin de mettre en application nos analyses nous avons choisi comme cas d'étude une application d'algèbre linéaire très étudiée et bien connue : la factorisation de Cholesky. Une manière standard de paralléliser les applications d'algèbre linéaire est de découper le problème en l'appliquant à différentes sous parties (ou *blocs*) des matrices.

Nous allons étudier en détails l'algorithme de Cholesky par blocs tel qu'implémenté dans PLASMA [Kurzak 2013], dont nous donnons le code dans le listing 4.7. Nous allons voir quelles sont ses parties critiques et leurs comportements, et nous allons également voir comment nous avons pu améliorer son exécution.

Listing 4.7: Algorithme de Cholesky par bloc tel qu'exprimé dans PLASMA

```
for (int k = 0; k < n_blocs; k++) {
  DPOTRF(A(k, k));

  for (int m = k+1; m < n_blocs; m++) {
    DTRSM(A(k, k), A(k, m));
  }

  for (int m = k+1; m < n_blocs; m++) {
    DSYRK(A(k, m), A(k, k));

    for (int n = k+1; n < m; n++) {
      DGEMM(A(k, n), A(k, m), A(n, m));
    }
  }
}
```

4.3.1 Description générale

La factorisation de Cholesky a pour but de résoudre l'équation suivante :

$$A = L * L^T$$

Où A est une matrice symétrique définie positive à coefficients réels, et L est l'inconnue, une matrice triangulaire inférieure.

Pour paralléliser la résolution de cette équation, nous allons découper la matrice A par bloc, et appliquer un algorithme de Cholesky par bloc. Nous pouvons donc caractériser une factorisation de Cholesky par sa taille de bloc et sa largeur en nombre de blocs.

L'algorithme de résolution par bloc repose sur quatre algorithmes basiques d'algèbre linéaire tirés des *BLAS - Basic Linear Algebra Subprograms* - décrits ci-dessous :

POTRF(A) Ce noyau effectue la factorisation de Cholesky de base sur une matrice symétrique définie positive A.

TRSM(A, B) Ce noyau résoud l'équation suivante : $A * X = B$, où A est une matrice triangulaire, et B une matrice générique. B est écrasée par la matrice solution X.

SYRK(A, C) Ce noyau effectue l'opération suivante : $C := A * A^t + C$, où A est une matrice générique, et C est une matrice symétrique.

GEMM(A, B, C) Ce noyau effectue une multiplication de matrices génériques, définie de la manière suivante : $C := A * B + C$, où A, B, et C sont des matrices génériques.

Pour permettre de mieux représenter l'algorithme du listing 4.7, les opérations se produisant sur chaque bloc de la matrice au rang `k` sont illustrées sur la figure 4.8

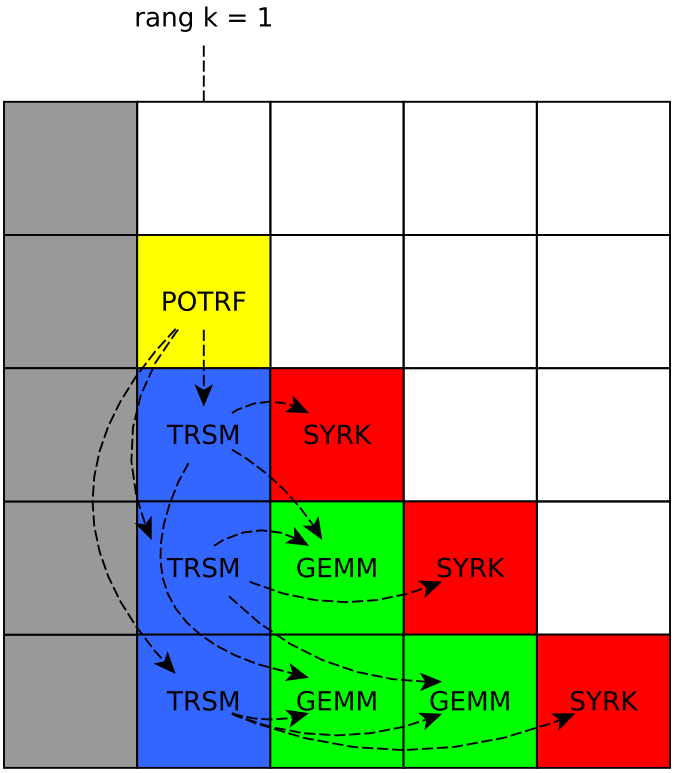

Figure 4.8: Itération du rang k de la factorisation de Cholesky

À chaque itération, un **POTRF** est d'abord effectué sur le bloc diagonal de l'itération. Les blocs de la colonne sont ensuite mis à jour via des **TRSM**, à la suite desquels les autres blocs restant peuvent être mis à jour par des **GEMM** (ou **SYRK** pour les blocs diagonaux). Le parallélisme de l'algorithme est donc principalement libéré par les **POTRF** ainsi que les **TRSM**. Cela peut être illustré par la Figure 4.9, qui donne le graphe de dépendances d'une factorisation de Cholesky de largeur 5.

Le nombre de tâches créées pour une largeur de matrice L (en nombre de blocs), le nombre d'opérations arithmétiques flottantes (ou *flops*) en fonction de la taille de bloc N, ainsi que l'intensité opérationnelle[2] sont résumés dans le tableau 4.1.

Les **GEMM** sont donc très largement majoritaires dans l'algorithme quand la largeur de la matrice augmente. En revanche en terme de Flops et d'intensité opérationnelle, on peut constater que tous les noyaux présentent des chiffres d'un ordre de grandeur équivalent.

4.3.2 Observations préliminaires et limites

Cette section et les suivantes montrent des évaluations reposant sur une bibliothèque BLAS, sauf indication contraire, la version utilisée est OpenBLAS 2.19. La figure 2.4 a montré qu'on pouvait observer l'impact de certains paramètres, tels que la taille de bloc ou le support exécutif, sur les performances globales.

[2]Définie par $\frac{Flops}{bytes}$

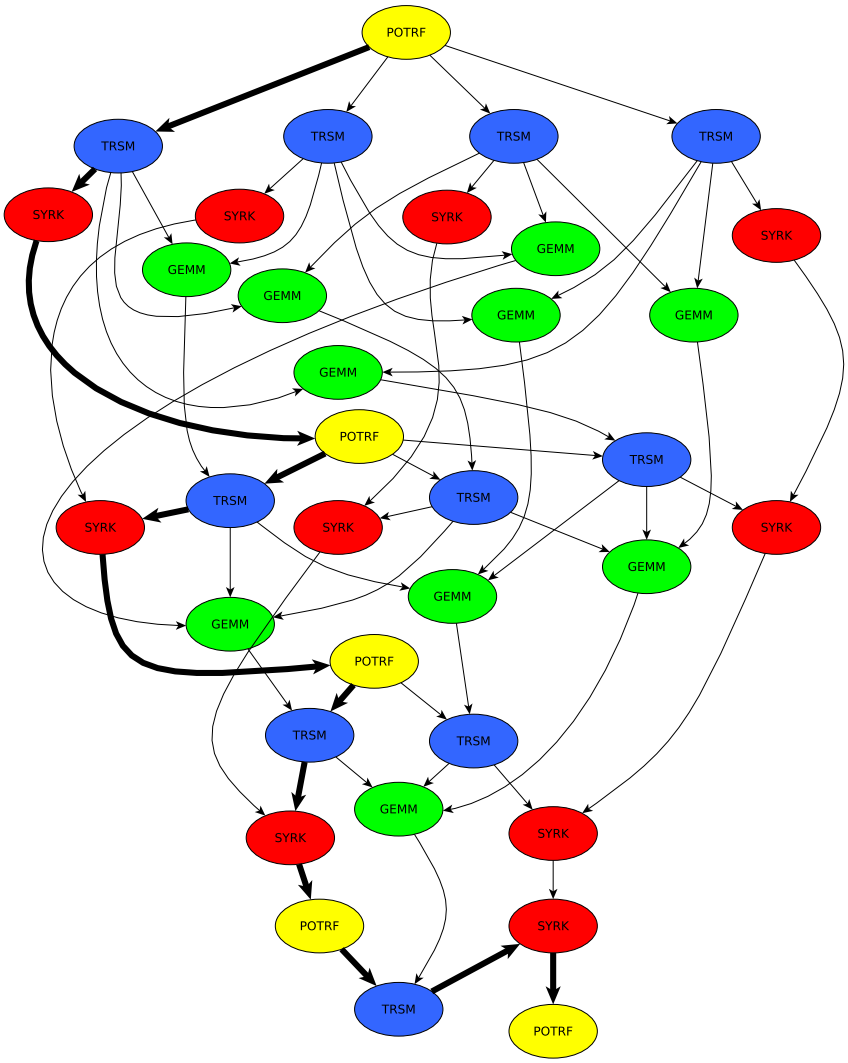

Figure 4.9: DAG d'un Cholesky de largeur 5

Noyau	Nombre pour une largeur de matrice L	Flops pour une taille de bloc N [Blackford 1999]	Intensité Opérationnelle
POTRF	L	$\frac{N^3}{3} + \frac{N^2}{2} + \frac{N}{6}$	$\frac{N}{24} + o(N)$
TRSM	$\frac{L*(L-1)}{2}$	N^3	$\frac{N}{8} + o(N)$
SYRK	$\frac{L*(L-1)}{2}$	$N^2 * (N+1)$	$\frac{N}{8} + o(N)$
GEMM	$\frac{L^3}{6} - \frac{L^2}{2} + \frac{L}{3} + 1$	$2 * N^3$	$\frac{N}{4} + o(N)$

Table 4.1: Nombre et complexité des différents noyaux

Certains supports exécutifs tels que Kaapi, StarPU, OpenStream, ou encore OmpSs permettent d'aller plus loin via un système de traces, permettant d'observer certaines caractéristiques de tâches particulières.

Pour illustrer cela, prenons un exemple d'évolution des performances de Cholesky en fonction du nombre de cœurs utilisés, montré sur la figure 4.10. Dans cet exemple

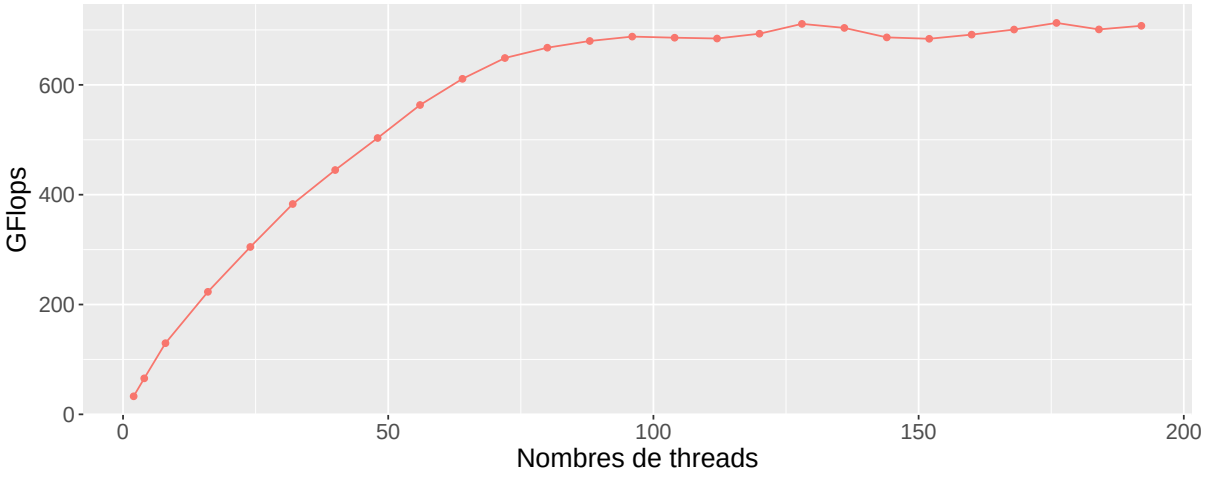

Figure 4.10: Évolution des performances de Cholesky avec libKOMP, pour une taille de matrice de 8192 et une taille de bloc de 224

la taille de matrice est de 8192, la taille de bloc de 224, et le support exécutif utilisé est libKOMP, sans aucune extension relatives à nos travaux sur l'affinité des données, que nous présentons dans le chapitre 5.

En activant le support des traces, on peut avoir plus de détails sur l'exécution individuelle de chacune des tâches.

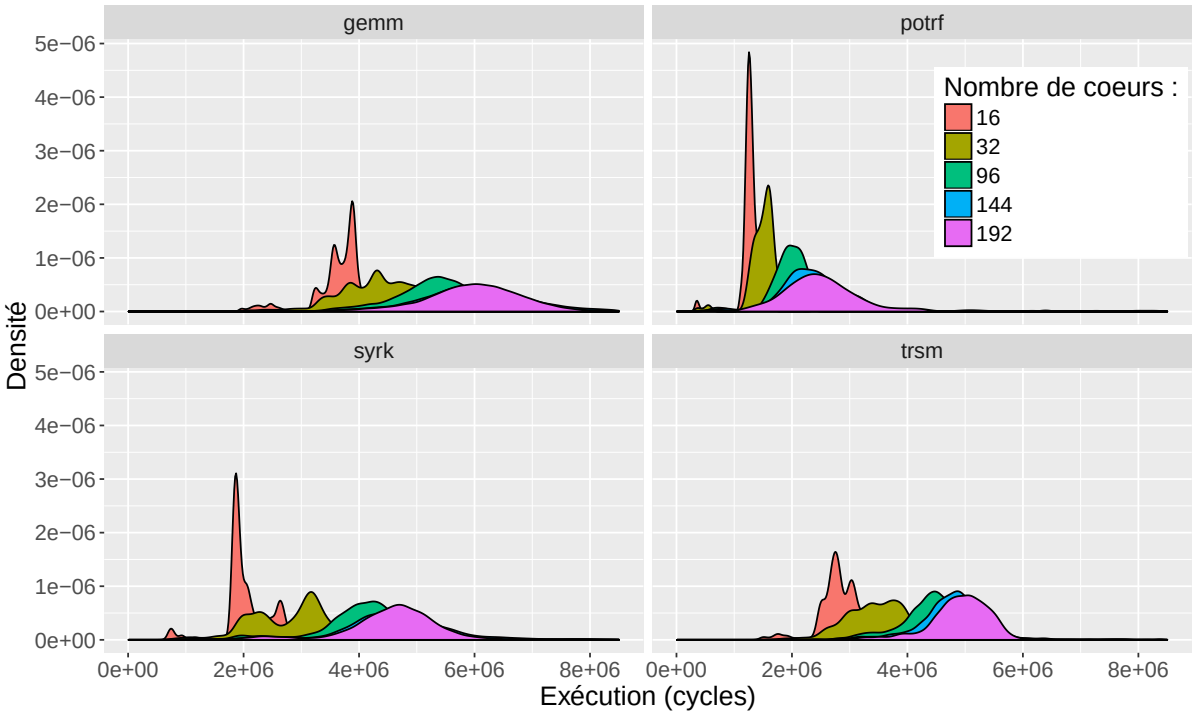

Figure 4.11: Distribution des différents noyaux en fonction du nombre de cœurs

La figure 4.11 montre, pour chaque type de tâche (ou *noyau*), la répartition du temps d'exécution (en cycles) en fonction du nombre de cœurs.

On constate deux types de répartitions :

- Une distribution relativement restreinte des cycles de chaque noyau, typiquement observée pour 16 cœurs. La distribution peut avoir plusieurs pics : pour **GEMM** sur 16 cœurs par exemple. Cela pourrait être expliqué par le nombre de cas possibles pour le placement des blocs de données manipulés par le noyau : avec 16 cœurs il y a 2 nœuds impliqués, et donc 4 cas pour la position des blocs (3 locaux, 2 locaux, 1 local, 3 distants), ce qui pourrait correspondre aux 4 niveaux de performances observés.

- Une distribution relativement large, typiquement observée pour un grand nombre de cœur. Pour un **GEMM** sur 192 cœurs, le nombre de cycles nécessaires pour l'exécution peut varier du simple au double !

Nous souhaiterions identifier d'où vient cette évolution dans la distribution, afin d'éventuellement réussir à la corriger en limitant sa largeur.

Malheureusement il n'existait pas, à notre connaissance, d'outil permettant d'isoler une (ou plusieurs) tâches d'une application, et permettant de changer certains paramètres prédéfinis pouvant avoir un impact sur le temps d'exécution de la tâche. Nous avons donc utilisé CarToN dans le but de comprendre et d'analyser plus en profondeur nos observations préliminaires.

4.3.3 Caractérisation détaillée des noyaux via CarToN

L'objectif de cette section est de décrire le processus expérimental nous ayant permis d'analyser et comprendre le comportement des quatre noyaux de Cholesky : **POTRF**, **TRSM**, **SYRK**, **GEMM**, qui a finalement abouti à des améliorations du support exécutif. Nous allons donc aborder d'une part les types de scénarios exécutés via CarToN, puis illustrer les résultats que nous avons obtenus avec des exemples significatifs.

4.3.3.1 Description des scénarios

Afin d'étudier le comportement de chaque noyau impliqué dans Cholesky, nous avons défini des scénarios au cours desquels les noyaux sont exécutés avec un contrôle sur les conditions d'exécution.

Pour un noyau donné (parmi **POTRF**, **TRSM**, **SYRK**, **GEMM**), le scénario de base est le suivant :

- Allocation et initialisation des données sur un nœud précis pour une taille de bloc donnée.

- Exécution d'un certain nombre de répétitions du noyau choisi (par défaut 50) sur ces données, en plaçant le thread de calcul soit sur un cœur du même nœud que les données, soit sur un cœur distant.

- Observation de la performance en FLOPS.

Ce scénario de base est donné en entier dans l'annexe A.2, pour l'exécution d'un **GEMM** local de taille 256.

Pour évaluer le comportement des noyaux en fonction de la charge de la machine, nous avons créé des scénarios exécutant simultanément plusieurs noyaux sur des données complètement indépendantes, et où le démarrage de l'exécution des noyaux est synchronisé. L'annexe A.3 illustre l'un de ces scénarios, qui permet d'exécuter 8 **GEMM** en parallèle, sur des données locales complètement indépendantes, avec une taille de matrice de 256.

Il y a plusieurs paramètres que l'on peut faire varier pour changer les conditions d'exécution :

- Le nombre de noyaux s'exécutant simultanément ;
- L'utilisation de données distantes ou locales ;
- La taille du bloc sur lequel appliquer le noyau.

Les trois sections suivantes décrivent l'impact des changements de ces paramètres, et illustrent certaines caractéristiques des machines utilisées qui donnent des opportunités pour de possibles améliorations du support exécutif.

4.3.3.2 Exécutions simultanées

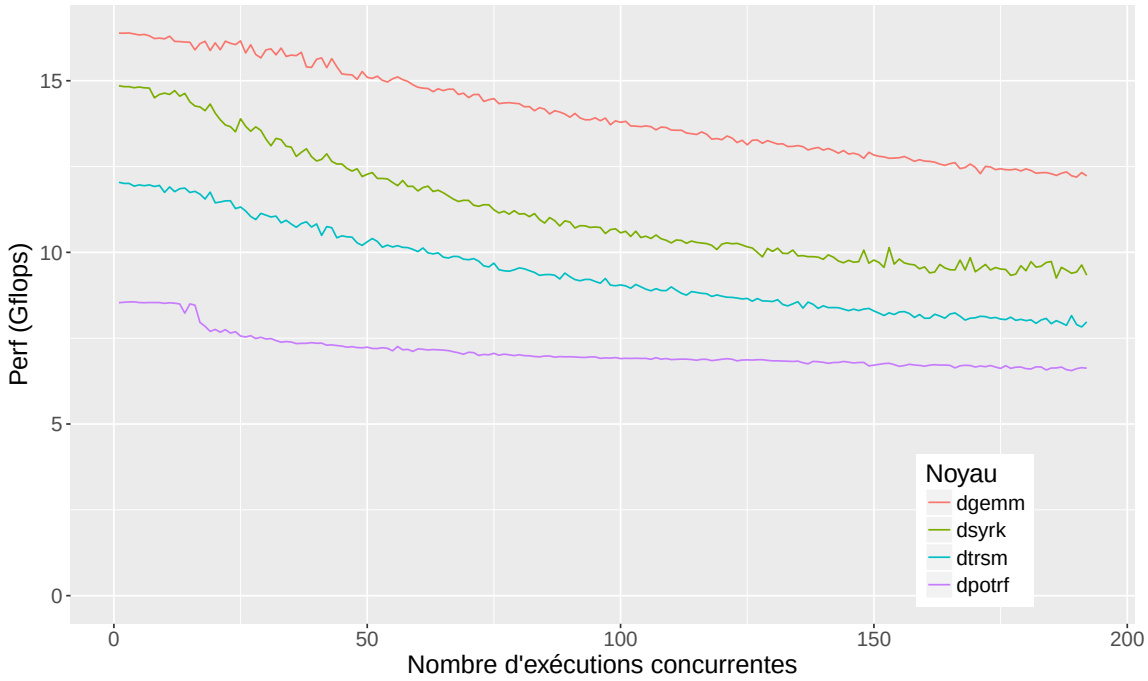

Figure 4.12: Performances des noyaux (B=256) avec données locales sur idchire

Afin d'évaluer l'impact de la charge de la machine, nous avons lancé des scénarios avec un nombre variable d'exécutions simultanées de chacun des noyaux. Pour le remplissage de la machine, nous avons placé les exécutions linéairement sur la machine, remplissant progressivement un nœud après l'autre.

La figure 4.12 montre la performance moyenne (en GFlops) de chaque noyau, en fonction du nombre de cœurs exécutant simultanément des noyaux. Par exemple pour déterminer le point d'abscisse 144 sur la figure pour un **GEMM**, nous avons exécuté 50 répétitions de **GEMM** sur chacun des 144 premiers cœurs de la machine idchire (remplissant donc les 18 premiers nœuds), de manière indépendante et simultanée. La performance moyenne pour ce point est obtenue en faisant la moyenne des performances sur l'ensemble des exécutions. Pour ce cas la taille de bloc a été fixée à 256, avec les données allouées et initialisées localement.

Pour une taille de bloc de 256, la quantité maximale de données utilisée par l'un des noyaux (**GEMM**) est de 256*256*8*3 = 1.5 Mo. La taille du cache L2 étant de 256 Ko, le jeu de données ne tient pas dans ce cache. Mais avec un nœud de 8 cœurs exécutant 8 exécutions concurrentes, la quantité totale de données utilisée serait au pire de 12.58 Mo, soit environ 50% des 20Mo de cache L3 disponible. On pourrait donc s'attendre à ce que la performance moyenne des noyaux ne soit impactée que par des effets locaux aux nœuds. Néanmoins les courbes montrent clairement une dégradation des performances de chaque noyaux lorsque la charge de la machine augmente. Ce comportement a également été observé pour d'autres tailles de blocs. Sur brunch le comportement a été également observé, avec néanmoins une dégradation moindre.

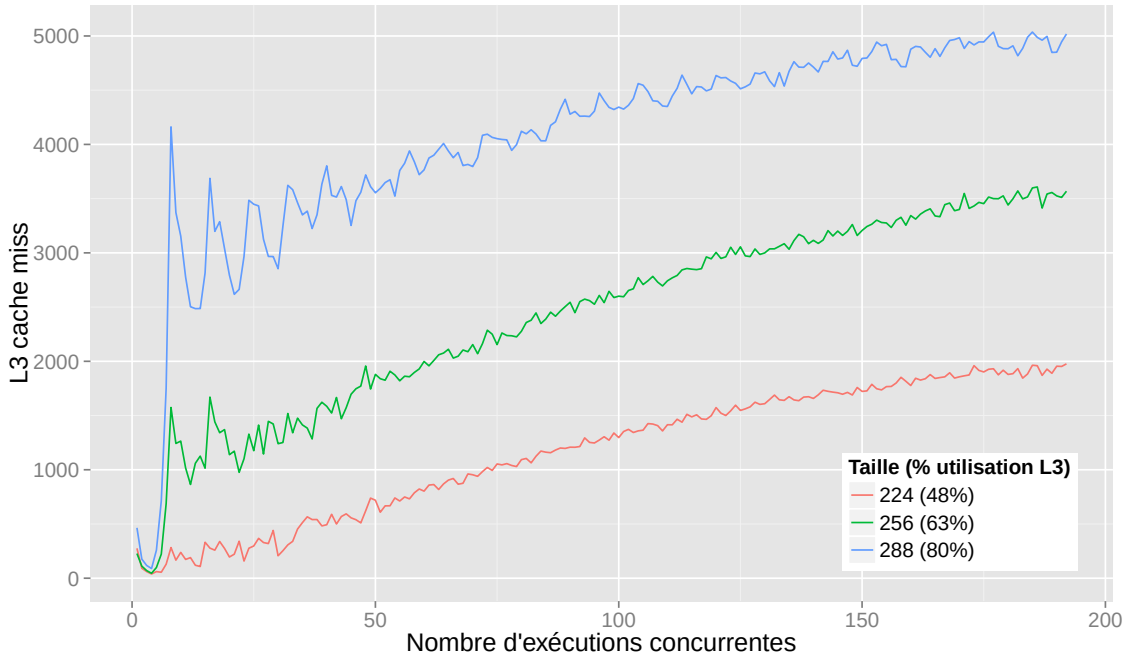

Figure 4.13: Cache miss au niveau L3 en fonction de la taille de bloc et du nombre d'exécutions concurrentes, pour un **GEMM** avec des données locales sur idchire

La figure 4.13 montre l'évolution du nombre moyen de cache miss au niveau L3 par noyau, en fonction du nombre d'exécutions concurrentes et de la taille de bloc. Compte tenu du fait qu'il y a 50 répétitions des noyaux, que l'ensemble des données locales manipulées par les **GEMM** tient théoriquement dans le cache L3, et que le cache n'est pas vidé entre chaque répétition, alors la première exécution devrait rendre le cache L3 «chaud» et permettre aux autres exécutions de s'exécuter sans cache miss

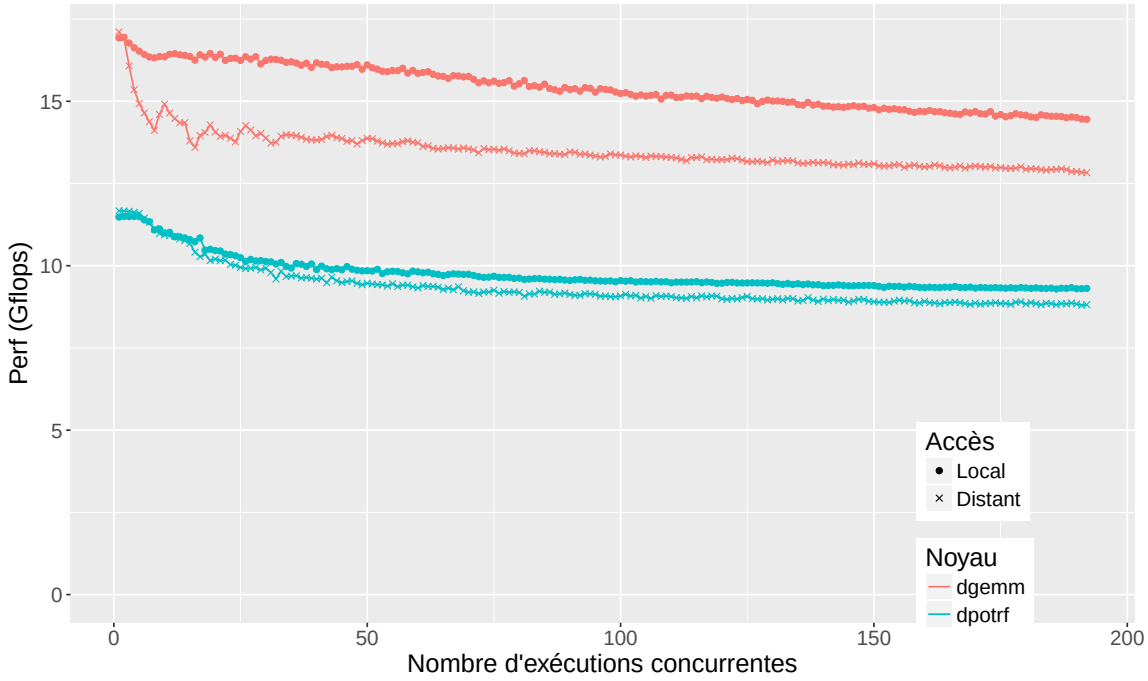

Figure 4.14: Performances GEMM, POTRF (B=512) avec données distantes sur *idchire*

au niveau L3. Nous pouvons voir avec ces mesures que malgré cette supposition théorique, en pratique on observe bien des cache miss au niveau L3 ! Nous avons pu observer via *hubstats*[3] que ces caches miss entrainaient une augmentation du trafic lié à la cohérence de cache sur l'ensemble des nœuds, ce qui explique donc que la baisse de performance soit généralisée, plutôt que purement locale à un nœud. Ces cache miss sont très probablement dus à l'associativité du cache L3 et à des conflits d'adresses.

4.3.3.3 Impact de la localité des accès

Les expériences de bande passante dans la section précédente ont montrées des différences significatives dans les temps d'accès à la mémoire locale et distante. Nous avons donc déroulé des scénarios avec des noyaux utilisant des données distantes ou des données locales afin de pouvoir les comparer, et éventuellement déceler des comportements typiques. Lorsque les données sont dites «locales», l'ensemble des données est allouée sur le nœud local. Lorsque les données sont dites «distantes», l'ensemble des données est alloué sur le nœud suivant (numériquement) dans la hiérarchie de la machine.

Les figures 4.14 et 4.15 illustrent les performances de deux noyaux, **GEMM** et **POTRF**, exécutés en concurrence sur des blocs de 512, en fonction du type d'accès, sur idchire et brunch, respectivement.

Avec une telle taille de bloc, l'ensemble des données pour tous les **POTRF** tient dans le cache L3, mais ce n'est pas le cas pour **GEMM**. Pour les **POTRF** la dégradation de performances est moindre : les données tiennent dans le cache L3, il y a donc un

[3]hubstats est un outil SGI permettant d'observer certains trafics NUMALink spécifiques, tels que la quantité de données reçues/envoyées liée à la cohérence de cache.

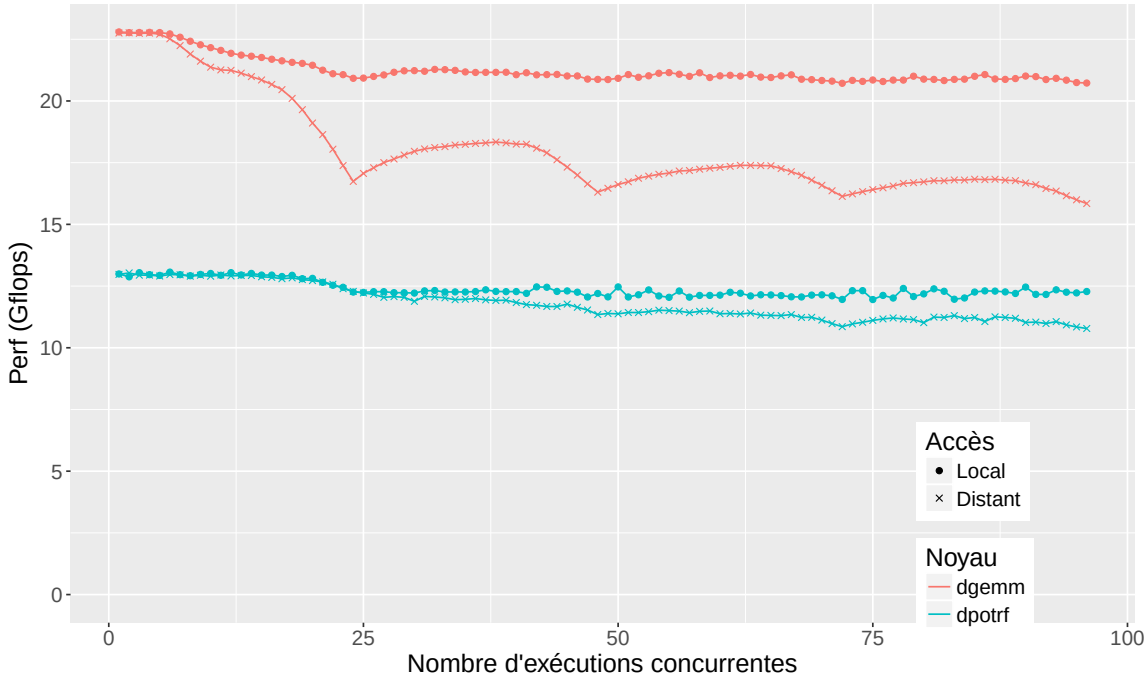

Figure 4.15: Performances GEMM, POTRF (B=512) avec données distantes sur *brunch*

coût pour rapatrier les données, mais une fois les données dans le cache L3 il n'y a plus besoin de faire d'accès distants.

Pour les **GEMM**, une utilisation de seulement quelques cœurs ne montre pas une différence de performances flagrante, en revanche le lien en sortie de nœud arrive assez vite à saturation (voir section 4.2.1.2), ce qui entraine une dégradation massive de performances. À la fois pour idchire et brunch, on peut observer l'impact de la bande passante sur les performances : les exécutions sont placées de manière à remplir progressivement les différents nœuds de la machine, et on peut observer que les courbes sont en dent de scie avec une période égale à la taille des nœuds sur chaque machine (8 sur idchire, 24 sur brunch).

Cela devient évident lorsqu'on regarde plus en détail le passage d'un nœud à un autre. Sur la figure 4.15, la courbe pour les **GEMM** distants remonte progressivement entre 24 et 36 cœurs utilisés : le fait de faire la moyenne des noyaux sur l'ensemble des cœurs cache légèrement le phénomène de saturation du lien. En revanche ce phénomène devient évident lorsque l'on affiche la moyenne du temps d'exécution par cœurs pour certains des points de la courbe, comme illustré sur la figure 4.16.

Le premier panneau montre la distribution des **GEMM** et des **POTRF** pour une exécution sur 24 cœurs concurrents situés sur le même nœud. Cette distribution montre un unique pic bien défini pour chaque noyau (environ 12.5 GFlops pour **POTRF**, et 16.5 pour **GEMM**). En revanche le passage à 26 cœurs (avec donc 2 cœurs situés seuls sur un autre nœud), montre une distribution avec deux pics : un pic important correspondant au 24 premiers cœurs, et un second pic plus petit, montrant des performances beaucoup plus grandes, pour les 2 cœurs situé sur l'autre nœud. Les autres panneaux montrent l'évolution de ces pics pour arriver au panneau 48, où les deux nœuds sont complètement utilisés.

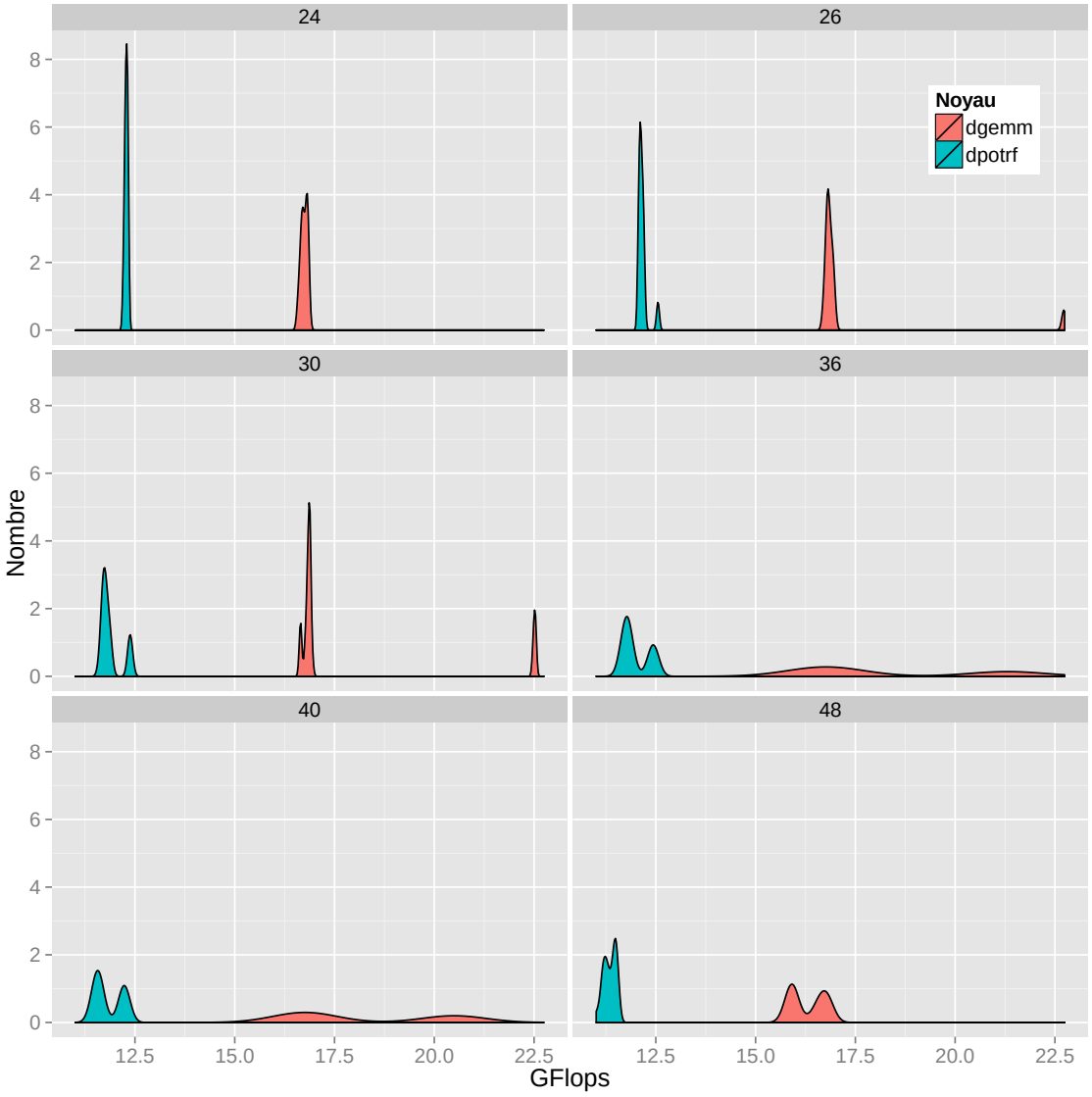

Figure 4.16: Distribution des performances de chaque noyau (Bloc = 512), en fonction du nombre total d'exécutions concurrentes, sur brunch.

L'impact de la localité des données est majeur dans le cas où le jeu de données manipulé par l'ensemble des cœurs ne tient pas dans les caches L3. Étant donné que la plus grande proportion des noyaux de Cholesky manipule 2 ou 3 blocs, la dégradation de performance devrait être importante lorsque la taille de bloc dépasse environ 320.

4.3.3.4 Impact de la taille de bloc

La figure 4.17 montre la performance des **GEMM** et **POTRF** sur idchire en fonction de la taille de bloc et du nombre de cœurs utilisés (tous les accès sont locaux).

La taille de bloc n'a pas d'impact significatif sur le phénomène observé précédemment de dégradation des performances. Elle a bien un impact sur le niveau de performance globale de chaque noyau (qui dépend de l'implémentation des BLAS), mais le comportement général de chaque noyau reste le même.

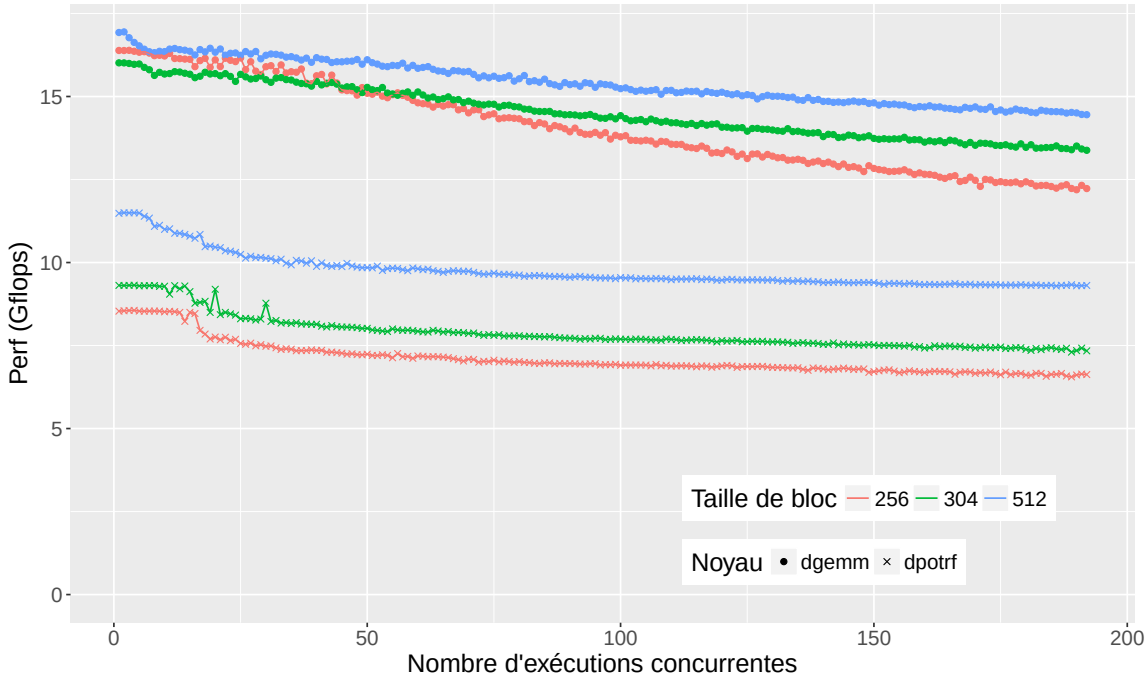

Figure 4.17: Performances des noyaux avec données locales sur idchire

Cette conclusion est similaire lors d'un placement distant des données.

4.3.3.5 Impact de la bibliothèque BLAS

Pour comparer l'impact potentiel de la bibliothèque implémentant les BLAS, nous avons configuré CarToN pour qu'il utilise soit OpenBLAS (2.19), Intel MKL (11.3), ou ATLAS (3.10.3). La figure 4.18 montre la performance des **GEMM** et **POTRF** sur *brunch*, pour une taille de bloc de 256 avec des données locales, en fonction de la bibliothèque BLAS utilisée.

Bien qu'on puisse constater une différence de performance brute, il y a dans tous les cas une baisse (relativement faible) des performances avec l'augmentation du nombre d'exécution concurrentes, bien qu'elles soient toutes indépendantes. Pour rappel sur *brunch* chaque nœud dispose de 24 cœurs, et pour les **GEMM** on peut deviner une baisse sur ces 24 premiers cœurs, qui d'autant plus flagrante que la performance est grande. Cela pourrait tout simplement s'expliquer par le fait qu'on atteint la limite de bande passante cumulée disponible sur le nœud local. Pour les autres cas, l'impact de la bibliothèque semble concerner principalement le pic de performance de chaque noyau, sans différence majeure sur l'allure générale des courbes.

4.3.4 Bilan et discussions

La figure 4.11 avait montrée, à l'aide des traces obtenues à partir d'une exécution, que nous pouvions observer une baisse et une dispersion des performances individuelles des noyaux de l'application Cholesky. CarToN nous a permis d'identifier principalement deux facteurs pouvant être à l'origine de ce phénomène : tout d'abord une partie de la baisse de performance générale est due à la baisse de performance

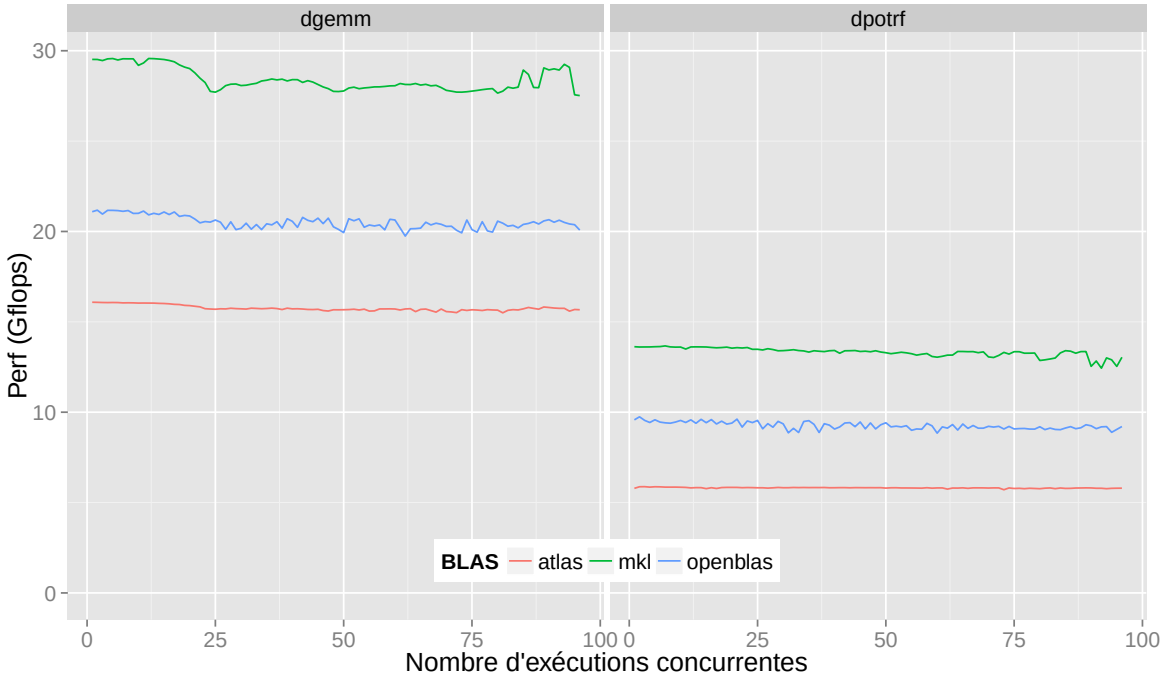

Figure 4.18: Comparaison des bibliothèques BLAS sur *brunch*, données locales, taille de bloc = 256

des noyaux qui accompagne l'augmentation de la charge de la machine, provenant de l'augmentation du trafic lié à la cohérence de cache, qui est une conséquence des cache miss au niveau L3 liés à l'associativité du cache. L'autre partie de cette baisse de performance, ainsi que la dispersion des temps d'exécution, peuvent être attribuées à une absence de contrôle de la localité des données au cours de l'application.

Le chapitre suivant est axé sur les différentes extensions que nous avons proposées dans le modèle de programmation et le support exécutif, afin d'essayer de mitiger le manque de localité des données. En pratique les observations faites à travers CarToN nous ont également servies à sélectionner des expériences et paramètres pertinents lors de l'évaluation des stratégies de vol de travail proposées avec la factorisation de Cholesky.

5 Utilisation et amélioration d'OpenMP

Ce chapitre regroupe les différentes contributions que nous avons faites spécifiquement dans le contexte d'OpenMP. Cela inclu tout d'abord une suite de benchmarks,

KASTORS, que nous avons créée face à l'absence de benchmarks utilisant les tâches avec dépendances d'OpenMP. Elle regroupe donc un ensemble de programmes implémentés à l'aide des tâches avec dépendances d'OpenMP, qui sont décrits dans la section 5.1. Nous décrivons ensuite des extensions dédiées à l'amélioration de la localité des données dans la section 5.2. La section 5.3 aborde les modifications faites au niveau du support exécutif pour qu'il puisse avoir une vision hiérarchique de la machine, et qu'il puisse exploiter les informations fournies par les extensions du langage OpenMP proposées. Enfin toutes ces propositions sont évaluées dans la section 5.4.

5.1 Préambule : une suite de benchmarks pour OpenMP 4.0, les KASTORS

Cette section présente KASTORS, une suite de benchmarks spécifiquement créée pour rassembler des applications variées utilisant les tâches avec dépendances d'OpenMP.

5.1.1 Motivation pour une nouvelle suite de benchmarks

Le support pour les applications à base de flots de données dans OpenMP est arrivé avec la version 4.0. Kurzak et al. [Kurzak 2010] ont montré l'intérêt en terme de performances des synchronisations point à point par dépendances de données par rapport aux synchronisations globales. Cette comparaison a eu lieu entre des applications exprimés dans deux modèles de programmation à base de tâche : Cilk, où les synchronisations sont explicites, et SMPsS [BSC 2008], où les synchronisations sont exprimées à l'aide des dépendances de données.

La version 4.0 d'OpenMP est sortie au démarrage de cette thèse, nous n'avions donc pas d'applications de référence utilisant les dépendances de données à travers OpenMP, et nous avons décidé d'introduire une suite de benchmarks - les KASTORS [Virouleau 2014] - spécifiquement orientée vers cette fonctionnalité. Elle regroupe des applications utilisées dans le HPC : des algorithmes d'algèbre linéaire dense, un stencil, et une factorisation LU d'une matrice creuse.

Il existe évidemment plusieurs suites de benchmarks à destination des architectures à mémoire partagée : PARSEC [Bienia 2008], SPECOMP [Aslot 2001], Rodinia [Che 2010], ou encore les NAS [Bailey 1994] sont des suites de benchmarks basées sur des versions antérieures d'OpenMP (2.0, 3.0), et donc utilisant principalement des boucles parallèles. La Barcelona OpenMP Task Suite [Duran 2009] (BOTS) propose des applications à base de tâches OpenMP 3.0 afin d'évaluer les implémentations existantes d'OpenMP en fonction de la manière de générer les tâches et de la répartition de la charge de travail.

Certaines applications que nous avons inclues dans KASTORS proviennent de différents benchmarks ou bibliothèques existantes. Deux applications, SparseLU et Strassen, ont été tirées des BOTS et adaptées pour ne plus utiliser de synchronisations globales. Trois applications d'algèbre linéaire dense, Cholesky, QR, et LU, ont été tirées des PLASMA [Kurzak 2013], une bibliothèque développée à ICL/UTK quit met à disposition un grand nombre d'algorithmes d'algèbre linéaire dense, optimisés pour les architectures multi-cœurs.

Nous avions donc assez d'éléments de base pour construire un ensemble d'applications, avec en tête les objectifs suivants :

- Rassembler et proposer une suite d'applications exploitant les dépendances introduites avec OpenMP 4.0 ;
- Comparer la version utilisant des synchronisations point à point (via des dépendances) à la version utilisant des synchronisations globales. Le but étant de montrer que le support exécutif peut gérer les synchronisations plus finement, et par conséquent améliorer les performances sans changer l'algorithme ;
- Permettre d'expérimenter des extensions à OpenMP, dont celles présentées dans les sections 5.2 et 5.3.

5.1.2 Description des applications

Les sections suivantes décrivent les six applications de la suite : d'où elles viennent et comment nous les avons étendus pour utiliser les dépendances de données.

5.1.2.1 Factorisations de Cholesky, QR, et LU

Ces trois applications ont été extraites des PLASMA, version 2.6 [PLASMA 2013]. Dans les PLASMA, chaque algorithme est écrit dans différents modèles de programmation : il existe une version parallélisée statiquement utilisant les threads POSIX, et une autre basée sur QUARK [YarKhan 2011], un modèle de programmation par flot de données, et utilisant un ordonnancement dynamique.

Les trois algorithmes que nous avons sélectionnés sont les factorisations de Cholesky, QR, et LU, respectivement nommés DPOTRF, DGEQRF, et DGETRF dans les BLAS. Ils opèrent tous sur des matrices de nombres flottants à double précision (type `double`).

Nous avons adapté l'implémentation originale de PLASMA pour retirer plusieurs niveaux d'encapsulation des fonctions, et ainsi faciliter la lecture et la maintenabilité du code. Les listings 5.1 et 5.2 montrent respectivement la version dynamique originale, et les transformations que l'on a faites pour porter le code en OpenMP 4.0.

Listing 5.1: Format de l'algorithme dynamique

```
wrapper_algorithm_call(matrice_description, plasma_specific_args
    ...) {
  // code séquentiel
  for (...) {
    // packing des paramètres
    QUARK_Insert_Task(wrapper_blas_function, packed_parameters);
  }
  // code séquentiel
  for (...) {
    // packing des paramètres
    QUARK_Insert_Task(
        wrapper_another_blas_function,
        packed_parameters);
```

```
  }
  // code sequentiel
}
```

Listing 5.2: Format de l'algorithme OpenMP

```
algorithm_call(matrice_description) {
    // code séquentiel
    for (...)
#pragma omp task depend(inout:array[...])
        blas_function(...);
    // code séquentiel
    for (...)
#pragma omp task depend(inout:array[...])
        another_blas_function(...);
    // code séquentiel
}
```

5.1.2.2 Jacobi

Jacobi est un algorithme itératif pour résoudre un système linéaire d'équations. À chaque itération les différents éléments d'un tableau sont mis à jour en suivant la même formule dépendant des éléments voisins (Stencil).

En pratique cette méthode résout l'équation de Poisson sur le carré unitaire [0,1]x[0,1], qui est divisé en une grille de NxN points espacés régulièrement. Le noyau de calcul principal est un Stencil à 5 points, en 2 dimensions. Ce noyau est appliqué successivement jusqu'à ce qu'une convergence soit détectée.

Nous avons implémenté plusieurs versions par blocs de ce noyau : une reposant sur l'utilisation de boucles parallèles OpenMP (illustrée dans le listing 5.3), une basée sur des synchronisations globales (illustrée dans le listing 5.4), et une basée sur les tâches avec dépendances (illustrée dans le listing 5.5).

Listing 5.3: Boucle itérative principale de Jacobi utilisant des for OpenMP

```
for (it = itold + 1; it <= itnew; it++) {
  // Save the current estimate.
  #pragma omp for collapse(2)
  for (int j = 0; j < ny; j += block_size)
    for (int i = 0; i < nx; i += block_size)
      copy_block(nx, ny, i/block_size, j/block_size, u_, unew_,
         block_size);

  // Compute a new estimate.
  #pragma omp for collapse(2)
  for (int j = 0; j < ny; j += block_size)
    for (int i = 0; i < nx; i += block_size)
      compute_estimate(i/block_size, j/block_size, u_, unew_, f_,
                       dx, dy, nx, ny, block_size);
}
```

Listing 5.4: Boucle itérative principale de Jacobi utilisant des tâches avec synchronisation globales

```
for (it = itold + 1; it <= itnew; it++) {
  // Save the current estimate.
  for (int j = 0; j < ny; j += block_size) {
    for (int i = 0; i < nx; i += block_size) {
      #pragma omp task shared(u_, unew_)
      copy_block(nx, ny, i/block_size, j/block_size, u_, unew_,
         block_size);
    }
  }

  #pragma omp taskwait

  // Compute a new estimate.
  for (int j = 0; j < ny; j += block_size) {
    for (int i = 0; i < nx; i += block_size) {
      #pragma omp task shared(u_, unew_, f_)
      compute_estimate(i/block_size, j/block_size, u_, unew_, f_,
                       dx, dy, nx, ny, block_size);
    }
  }
  #pragma omp taskwait
}
```

Listing 5.5: Boucle itérative principale de Jacobi utilisant des tâches avec dépendances

```
// Calcul des voisins
#define west(i) ((i==0) ? i : i - block_size)
#define east(i) ((i==nx-block_size) ? i : i + block_size)
#define north(j) ((j==ny-block_size) ? j : j + block_size)
#define south(j) ((j==0) ? j : j - block_size)

for (int it = itold + 1; it <= itnew; it++) {
  // Save the current estimate.
  for (int j = 0; j < ny; j += block_size) {
    for (int i = 0; i < nx; i += block_size) {
      #pragma omp task shared(u_, unew_)\
                  depend(in: unew[i][j]) \
                  depend(out: u[i][j])
      copy_block(nx, ny, i/block_size, j/block_size, u_, unew_,
                 block_size);
    }
  }

  // Compute a new estimate.
  for (int j = 0; j < ny; j += block_size) {
    for (int i = 0; i < nx; i += block_size) {
      #pragma omp task shared(u_, unew_)\
```

```
                    depend(out: unew[i][j])\
                    depend(in: f[i][j], u[i][j],\
                               u[west(i)][j], u[east(i)][j],\
                               u[i][north(j)], u[i][south(j)])
      compute_estimate(i/block_size, j/block_size, u_, unew_,
                       f_, dx, dy, nx, ny, block_size);
    }
  }
}
```

5.1.2.3 SparseLU

Cette application calcule la factorisation LU d'une matrice creuse. Nous avons modifié l'implémentation originale des BOTS pour ajouter des dépendances de données. Ces modifications sont décrites dans les listings 5.6 et 5.7.

Listing 5.6: LU utilisant des tâches indépendantes

```
for (k=0; k<NB; k++) {


  lu0(M[k*NB+k]);
  for (j=k+1; j<NB; j++) {


#pragma omp task shared(M)
    fwd(M[k*NB+k], M[k*NB+j]);
  }

  for (i=k+1; i<NB; i++) {


#pragma omp task shared(M)
    bdiv(M[k*NB+k], M[i*NB+k]);
  }
#pragma omp taskwait

  for (i=k+1; i<NB; i++) {
    for (j=k+1; j<NB; j++) {


#pragma omp task shared(M)
      bmod(M[i*NB+k],
           M[k*NB+j],
```

Listing 5.7: LU utilisant des tâches avec dépendances

```
for (k=0; k<NB; k++) {
#pragma omp task shared(M)\
    depend(inout: M[k*NB+k])
  lu0(M[k*NB+k]);
  for (j=k+1; j<NB; j++) {
#pragma omp task shared(M)\
    depend(in: M[k*NB+k])\
    depend(inout: M[k*NB+j])
    fwd(M[k*NB+k], M[k*NB+j]);
  }

  for (i=k+1; i<NB; i++) {
#pragma omp task shared(M)\
    depend(in: M[k*NB+k])\
    depend(inout: M[i*NB+k])
    bdiv(M[k*NB+k], M[i*NB+k]);
  }


  for (i=k+1; i<NB; i++) {
    for (j=k+1; j<NB; j++) {
#pragma omp task shared(M)\
        depend(in: M[i*NB+k])\
        depend(in: M[k*NB+j])\
        depend(inout: M[i*NB+j])
      bmod(M[i*NB+k],
           M[k*NB+j],
```

```
28              M[i*NB+j]);
29      }
30    }
31 #pragma omp taskwait
32 }
```

```
28              M[i*NB+j]);
29      }
30    }
31
32 }
```

5.1.2.4 Strassen

L'application Strassen utilise des décompositions de matrices pour calculer le produit de grandes matrices denses. De manière similaire à SparseLU, nous avons modifié l'implémentation des BOTS pour ajouter du parallélisme au niveau des additions dans l'algorithme, et nous avons exprimé des dépendances de données plutôt que d'utiliser une synchronisation à base de `taskwait`.

5.1.3 Résumé des performances

Nous avons évalué l'intérêt des tâches avec dépendances comparées aux tâches indépendantes dans l'article publié à IWOMP2014 [Virouleau 2014]. Les expériences ont été menées sur toutes les applications avec les supports exécutifs de GCC, libGOMP, et de Clang-omp (précurseur du support d'OpenMP dans Clang), libIOMP. Pour les applications tirée de PLASMA, nous les avons également comparées aux deux versions originales de PLASMA : statique ou basée sur le modèle par flot de données QUARK.

Les résultats ont montré que l'utilisation des dépendances n'avait jamais d'impact négatif sur les performances. En particulier pour les applications tirées des PLASMA, nous avons montré que la version OpenMP pouvait être aussi compétitive que la version écrite dans un support exécutif spécifique. Cela a pu confirmer que les supports exécutifs pouvaient gérer les synchronisations finement, éliminant ainsi les inactivités des threads dues aux `taskwait`, se traduisant du même coup par une amélioration des performances.

D'autres part, certaines de nos expériences ont permis de mettre en évidence d'autres limitations des tâches OpenMP. La figure 5.1 regroupe une comparaison de l'application de jacobi, en fonction de la version implémentée (à base de boucles ou de tâches avec dépendances) et du support exécutif, effectuée sur idchire. La scalabilité de l'application en elle-même est limitée, mais il n'y a pas de raison, a priori, qu'il y ait une dégradation de performances de l'application avec l'augmentation du nombre de cœurs. Jacobi est une application stencil, et chacune des tâches successives (ou groupe d'itérations pour la version à base de boucles) dépend énormément de la réutilisation du cache : la structure de tâches avec dépendances ne permet pas d'exprimer ce genre de contraintes, lors de l'ordonnancement certaines tâches vont se faire voler et briser la localité des données, introduisant une grosse dégradation des performances.

5.1.4 Discussions et perspectives

Le problème évoqué pour l'application Jacobi vient du fait que l'ordonnanceur n'est pas conscient du lien fort qui existe entre une tâche et ses données, et qu'il permet

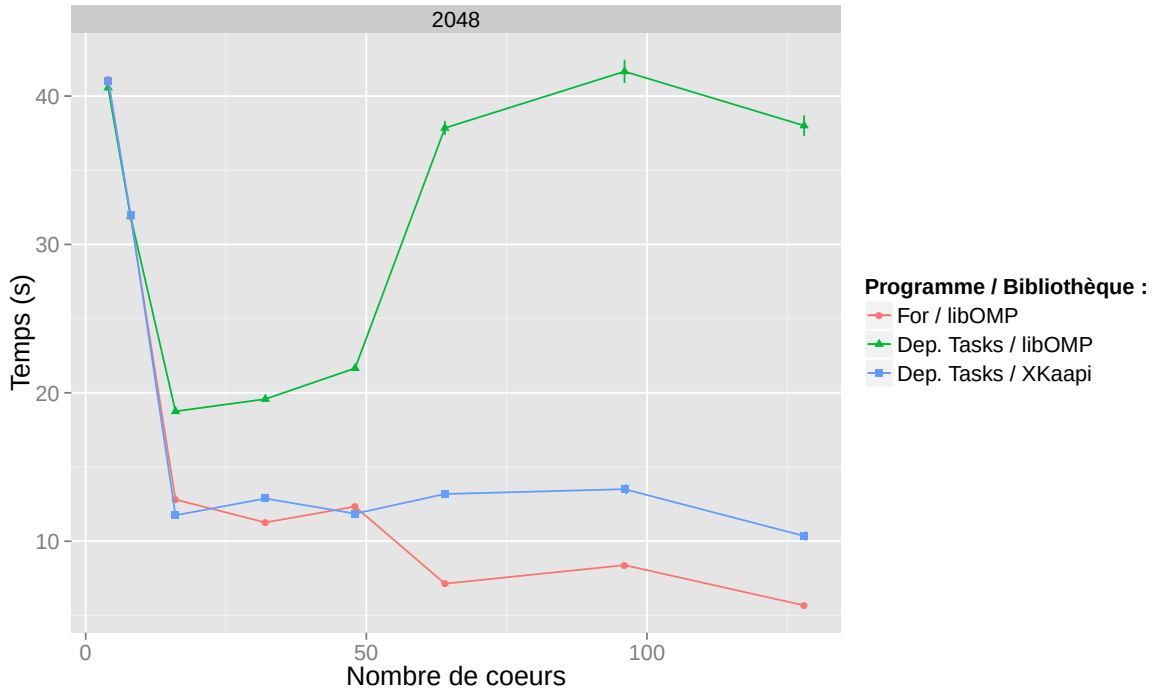

Figure 5.1: Comparaison des performances de Jacobi sur idchire, avec une taille de matrice de 49152 et une taille de bloc de 2048

à plusieurs tâches de s'exécuter loin de leurs données. Une solution possible pourrait être d'attacher les données à des bancs NUMA, et d'exprimer une *contrainte* entre une tâche et un cœur de la machine.

Les résultats de la section 4.3.3 ont également montrés l'importance de la proximité des données pour les différents noyaux de Cholesky. Il semble donc très intéressant de pouvoir introduire une clause permettant l'expression d'une *contrainte* entre une tâche et ses données, qui pourrait en fait bénéficier toutes les applications d'algèbre linéaire des KASTORS.

La section suivante aborde ce point, et la section 5.4 décrit les résultats obtenus à partir des KASTORS étendus avec nos propositions.

5.2 Amélioration de l'expressivité du langage

Nous avons proposé plusieurs extensions au langage et à l'API d'OpenMP, dont le but général est de faciliter le travail du support exécutif pour maintenir la proximité entre une tâche et ses données au cours de l'exécution du programme.

5.2.1 Description du besoin

Dans un contexte où l'on souhaite améliorer le contrôle sur les tâches et leurs données, OpenMP 4.0 ne propose que deux fonctionnalités : les `OMP_PLACES` et `OMP_PROCBIND`, mais cela n'a d'effet que sur le placement des threads sur la topologie, et non sur les tâches qui leur sont attribuées. Il n'y a rien au sein d'OpenMP qui permet d'exprimer

une relation entre une tâche et une donnée ou une partie de la topologie de la machine.

En dehors d'OpenMP, les programmeurs utilisent généralement des bibliothèques ou outils externes dans le but de contrôler le placement des données, tels que MAi [Ribeiro 2009] et MaMi [Broquedis 2010a]. L'efficacité de certaines approches peut être remise en cause (comme dans le cas de *numactl*), et les approches utilisant une bibliothèque externe peuvent être relativement intrusives.

Nous avons donc introduit deux types de constructions dans OpenMP : d'une part un moyen de contrôler la distribution initiale des données manipulées par les tâches, et d'autres part un moyen d'associer les tâches à ces données (ou mieux, à n'importe quelle partie de la machine).

5.2.2 Contrôle de la distribution des données

Utiliser une initialisation séquentielle des données de son application peut avoir des conséquences dramatiques sur les performances, tout spécialement lorsque l'on cible des machines NUMA. Les données se retrouvent dans ce cas sur un unique nœud, dont le bus mémoire va être complètement saturé lorsque les cœurs de tous les autres nœuds vont accéder aux données.

Classiquement les programmeurs effectuent l'allocation et l'initialisation des données de manière séquentielle, et utilisent *numactl* pour en faire la distribution. L'efficacité de cette méthode reste très moyenne, comme nous l'avons montré dans [Virouleau 2016a].

Ici on va supposer que, de la même manière que le programmeur exprime le parallélisme de son application à base de tâches, il initialise également les données de son application à l'aide de tâches dans une région parallèle séparée.

Dans la plupart des applications que nous avons utilisées nous avons constaté que l'initialisation des données suivait un schéma où l'allocation et l'initialisation de chacun des blocs est fait au fur et à mesure des besoins ou de la construction des structures de données de l'application.

L'initialisation des données devant être groupées a donc lieu dans une même tâche. Les solutions de l'état de l'art consistent à allouer, ou migrer, les données initialisées par ces tâches. Plutôt que d'utiliser une technique similaire, nous allons nous baser sur le principe portable du *first-touch*, décrit dans la section 2.2.

Nous proposons ici de distribuer les tâches d'initialisations selon une stratégie choisie par l'utilisateur. Gérer le placement des tâches est plus efficace et moins couteux que de migrer des données, et permet en fait d'arriver au même résultat : en distribuant les tâches d'initialisations, on distribue les tâches effectuant le *first-touch* des données, ce qui a pour effet de distribuer physiquement ces données. Par rapport à *numactl* cela présente également l'avantage de garder sur le même nœud NUMA les données s'étendant sur plusieurs pages.

Pour pouvoir indiquer au support exécutif quelles tâches marquer comme "tâches d'initialisation" et comment gérer leur placement sur la topologie, nous avons mis en place une clause s'appliquant sur une région parallèle :

```
1 init(random | cyclic | cyclicnuma)
```

Elle indique à l'ordonnanceur de tâches que pour la région parallèle courante les tâches prêtes sans dépendances devraient être distribuées sur la machine en suivant une stratégie :

random : distribution aléatoire sur les cœurs de la machine.

cyclic : distribution de manière cyclique sur les cœurs de la machine.

cyclicnuma : distribution cyclique sur les nœuds de la machine.

Malgré une restriction dans la manière de réaliser l'initialisation des données de l'application, cet ajout permet au programmeur de spécifier une distribution de données avec une modification minimale du code, et plus efficacement qu'avec les solutions existantes.

Si cette restriction peut être trop forte pour certaines applications, il est aussi possible pour les cas particuliers de spécifier une clause affinité stricte (définie dans la section suivante) sur chacune des tâches d'initialisation.

5.2.3 Ajout d'une clause affinité

Pour permettre d'exprimer une association entre une tâche et une donnée (ou un élément de la topologie), nous avons proposé l'introduction du mot clé `affinity` dans le langage OpenMP, qui a été présentée lors du Workshop International sur OpenMP (IWOMP) en 2016 [Virouleau 2016b]. Comme constaté dans le chapitre 4 et souvent mentionné dans la littérature, un point clé pour obtenir de bonnes performances sur des architectures NUMA est de garantir la proximité entre une tâche et ses ressources.

L'objectif de cette clause est donc de permettre à l'utilisateur de pouvoir spécifier un lien privilégié - une *affinité* - entre une tâche et un élément de l'architecture. On distingue donc trois types d'affinité que le programmeur pourrait avoir besoin d'exprimer :

affinité à un thread : le support exécutif devrait essayer d'ordonnancer la tâche sur le thread donné.

affinité à un nœud NUMA : le support exécutif devrait essayer d'ordonnancer la tâche sur n'importe quel thread du nœud NUMA donné.

affinité à une donnée : quand une tâche devient prête pour l'exécution, le support exécutif devrait l'ordonnancer sur n'importe quel thread attaché au nœud NUMA sur lequel la donnée a été physiquement allouée.

De plus, le programmeur peut indiquer si cette affinité est *stricte*, indiquant que la tâche **doit** s'exécuter sur la ressource indiquée. Si le programmeur n'indique pas une affinité stricte, l'ordonnanceur peut décider d'exécuter la tâche sur une ressource différente, pour équilibrer la charge de calcul par exemple.

Cette extension visant les constructions de type tâche, elle a été implémentée comme une nouvelle clause pour la directive `task`. La syntaxe proposée est la suivante :

```
affinity([node | thread | data]: expr[, strict])
```

Dans tous les cas l'expression `expr` est un entier naturel ou un pointeur, qui est interprété d'une manière spécifique :

thread : `expr` est interprétée comme un id de thread. On définit ici la notion d'id de thread comme l'indice du thread au sein des `OMP_PLACES` pour la *team* OpenMP courante. Voici une illustration, en prenant une exécution sur quatre threads avec comme valeur `OMP_PLACES="{2},{5},{8},{9}"` :

- Les quatre *threads* qui constituent la *team* vont être ici placés sur quatre *cœurs* de la machine d'indice `2`, `5`, `8`, et `9`.
- Les *threads* étant toujours numérotés à partir de `0` dans une *team*, la correspondance entre thread et cœur sera donc la suivante : le thread d'indice `0` sera donc placé sur le cœur `2`, le thread d'indice `1` sera placé sur le cœur `5`, et ainsi de suite.

node : `expr` est interprétée comme un id de nœud NUMA. Comme pour le cas précédent, la notion d'id est définie relativement aux places de la *team* OpenMP courante. En reprenant l'exemple précédent, supposons que les cœurs 2,5,8, et 9 sont physiquement situés sur 2 nœuds NUMA différents. Il y aura alors 2 nœuds NUMA déduits des places, et les ids utilisés pourront être 0 ou 1.

data : `expr` est une adresse mémoire. Si le nœud NUMA associé à la donnée ne peut être déterminé, le nœud utilisé par défaut est le nœud local.

Si `expr` désigne une ressource hors limites, la valeur considérée par le support exécutif est prise modulo le nombre de ressources correspondantes.

5.2.4 Extension des fonctions du support exécutif

Si les points précédents décrivent des extensions directement au niveau des constructions OpenMP, il est également important de pouvoir fournir dynamiquement certaines informations au programmeur au cours de l'exécution du programme. Dans ce but nous avons également ajouté quelques fonctions à l'API d'OpenMP, dont le but est de fournir des informations à propos de l'architecture et de la *team* OpenMP courante :

```
// Retourne le nombre de nœuds NUMA dans la team
int omp_get_num_nodes(void);

// Retourne le nœud NUMA sur lequel
// la tâches est actuellement exécutée
int omp_get_node_num(void);

// Retourne le nœud NUMA sur lequel la donnée a été allouée
int omp_get_node_from_data(void *ptr);
```

Ces fonctions retournent des informations spécifiques à la *team* d'une région région parallèle OpenMP. Sur les machines sans support NUMA, nous considérons que tous les threads sont sur un unique nœud NUMA.

Nous avons également rendu accessible l'ajout d'affinité sur une tâche à une fonction de l'API :

```
1 void omp_set_task_affinity(omp_affinitykind_t k,
2                             uintptr_t ptr, int strict);
```

Cette fonction aura un impact sur la prochaine tâche créée dans la région. Les paramètres de la fonction correspondent aux paramètres de la clause :

omp_affinitykind_t k peut être soit `omp_affinity_thread`, `omp_affinity_node`, ou `omp_affinity_data`.

uintptr_t ptr correspond à une expression désignant la ressource.

int strict indique si l'affinité est stricte (`strict != 0`) ou non (`strict == 0`).

5.2.5 Notes d'implémentation

Les extensions ont été implémentées dans le compilateur Clang[1], en nous basant sur la version 3.9.

Pour intégrer ces extensions, nous avons dans un premier temps étendu différentes parties du *frontend* de Clang : la partie en charge de l'analyse syntaxique, pour permettre la compréhension des nouvelles clauses et l'enrichissement de l'arbre syntaxique abstrait avec les informations présentes ; et la partie en charge de l'analyse sémantique afin d'appliquer certaines restrictions sur les clauses (telles que les constructions sur lesquelles elles peuvent être appliquées, ou le type des données attendu).

Dans un second temps nous avons étendu la génération de code, en ajoutant différents point d'entrée dans l'ABI et en passant les données présentes dans l'arbre.

L'ajout des nouveaux points d'entrées dans l'ABI signifie qu'il faut également rajouter leur prise en compte dans le support exécutif, et que du code (contenant des affinités) compilé par notre version modifiée du compilateur n'est pas compatible avec des supports exécutifs n'implémentant pas ces points d'entrée.

L'extension du support exécutif et l'utilisation des informations fournies par le programmeur sont décrites dans la section suivante.

5.3 Extension du support exécutif

Cette partie complète la section précédente, elle se concentre sur l'exploitation des informations fournies par l'utilisateur, ainsi que sur la prise en compte des architectures NUMA côté support exécutif. Ces travaux ont été publiés à EuroPar 2016 [Virouleau 2016b].

Ces extensions ciblent des supports exécutifs fonctionnant par vol de travail (voir section 2.4.2.2), et ont été implémentées dans XKaapi ainsi que dans la nouvelle version de libKOMP (voir section 5.4.1.1).

[1] https://gitlab.inria.fr/openmp/clang

5.3.1 Hiérarchiser le support exécutif

La modification la plus importante consiste à hiérarchiser les files de tâches pour le vol de travail, puisque c'est là dessus que se base l'ordonnancement.

Des outils tels que hwloc permettent de donner des informations sur la hiérarchie, nous avons donc des informations précises sur quel cœur est placé physiquement au sein de quel nœud NUMA, et nous avons donc pu mettre en place une hiérarchie dans les files de tâches :

- Chaque cœur possède deux files de tâches : une publique, dans laquelle les autres cœurs peuvent venir voler ; et une privée, dans laquelle tout le monde peut venir ajouter des tâches, mais seul le cœur propriétaire peut venir voler (illustré sur la figure 5.2).

- Chaque nœud possède également deux files de tâches, suivant le même principe que précédemment. Pour la file privée, seuls les cœurs situés sur le nœud NUMA propriétaire peuvent venir voler des tâches.

Nous avons également fait en sorte d'allouer la mémoire manipulée par les différentes files sur le nœud NUMA sur lequel cette file est située. La figure 5.2 illustre la hiérarchie des files sur un exemple avec deux nœuds NUMA de huit cœurs (les files privées des cœurs ont été omises pour ne pas surcharger la figure).

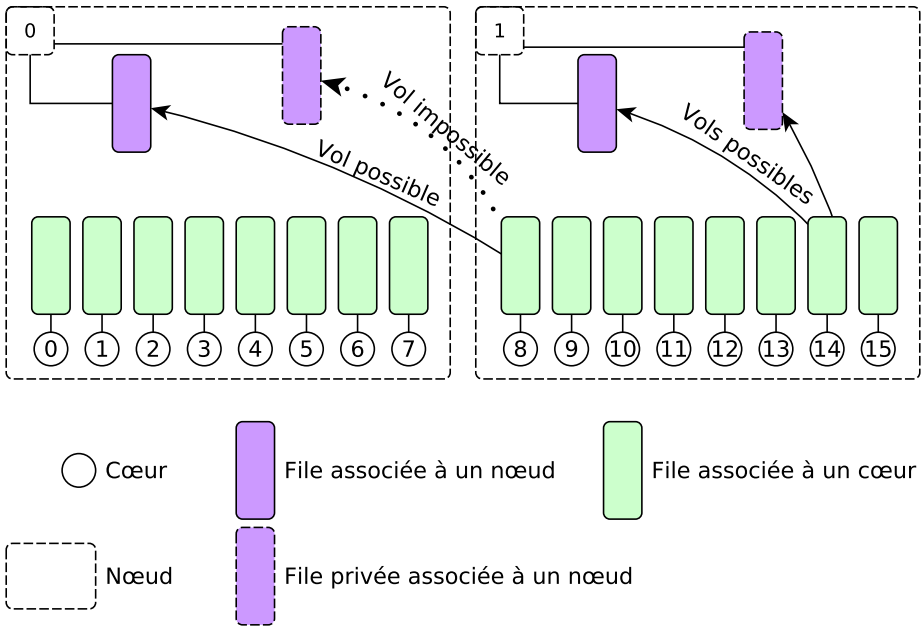

Figure 5.2: Schéma de la hiérarchie des queues publiques, avec 2 nœuds NUMA de 8 cœurs

5.3.2 Heuristiques basées sur la localité des données

Comme décrit dans la section 2.4.2.2, le vol de travail repose sur deux étapes essentielles : le choix d'une file où placer la tâche lorsqu'elle devient prête, et la sélection

d'une file de tâches lors du vol. Les deux sections suivantes décrivent les modifications que nous avons mises en place dans le support exécutif, dans l'objectif de prendre en compte le côté NUMA de l'architecture.

5.3.2.1 Distribution des tâches prêtes : stratégies de *placement*

Au cours de l'exécution, un thread qui termine une tâche va rendre prête d'autres tâches. L'ordonnancement par vol de travail dans Cilk place ces tâches dans la file locale du thread. Compte tenu des informations que nous avons sur les tâches et leurs données, d'autres choix sont possibles pour améliorer la localité des données.

Nous introduisons quatre stratégies différentes pour le placement des tâches prêtes, vis à vis des files de tâches hiérarchiques définies dans la section précédente. Deux d'entre elles sont indépendantes des données, alors que les deux autres prennent en compte les informations fournies par l'utilisateur via la clause `affinity` décrite dans la section 5.2.3, ou à défaut utilisent des informations issues des dépendances de données.

pLoc : le cœur responsable du placement de la tâche place celle ci dans sa propre file - *placement**Loc**al*.

pLocNum : la stratégie agie de manière similaire à la précédente, à ceci près que le cœur place celle ci dans la file de son nœud NUMA - *placement**LocalNuma*.

pNumaW : le cœur responsable du placement de la tâche regarde si une affinité de donnée est spécifiée, si aucune affinité n'est présente, il utilise une des dépendances en écriture de la tâche comme affinité. Il détermine ensuite le nœud NUMA sur lequel a été allouée la donnée, et place la tâche dans la file du nœud correspondant.

pNumaWLoc : la stratégie agie de manière similaire à la précédente, mais si le nœud NUMA déterminé correspond au nœud NUMA du cœur courant, alors la tâche est poussée dans la file locale du cœur directement.

Un détail distingue les deux premières stratégies des deux secondes : dans le cas des stratégies `pLoc` et `pLocNum`, le placement initial des données n'est pas pris en compte, alors que les stratégies `pNumaW` et `pNumaWLoc` utilisent toutes deux les informations sur l'allocation physiques des données manipulées par la tâche.

5.3.2.2 Équilibrage de charge dynamique : stratégie de *sélection*

Nous avons implémenté un ensemble de stratégie de sélection de files de tâches, qui sont utilisées lorsqu'un cœur inactif cherche à voler du travail. Ces stratégies ont conscience de la topologie de l'architecture sur laquelle est exécutée le programme.

Les deux premières sont similaires aux travaux effectués par Olivier et al. [Olivier 2012] :

sRand : sélection aléatoire d'une file parmi les files des cœurs.

sRandNuma : sélection aléatoire d'une file parmi les files des nœuds.

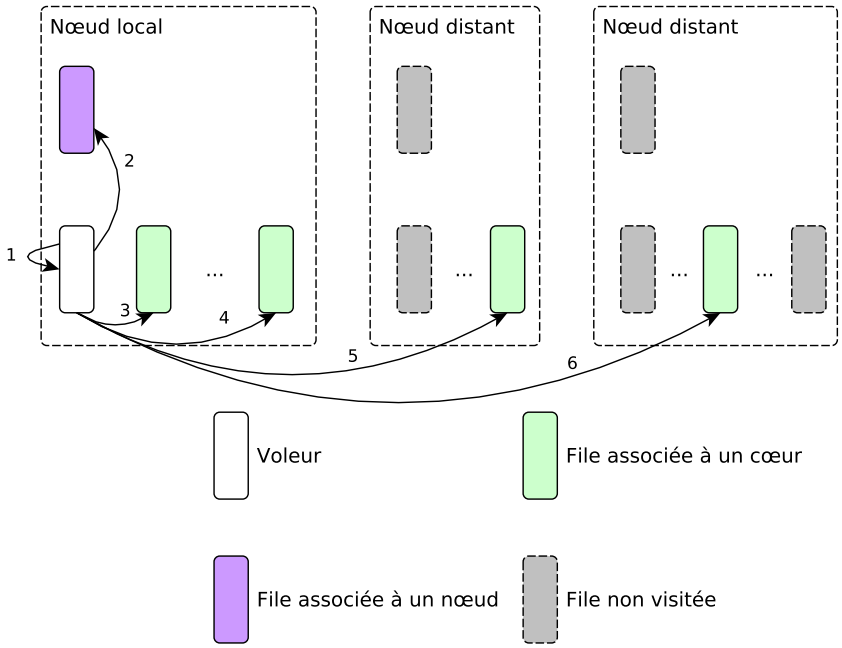

Figure 5.3: Illustration de la stratégie *sProc*, avec l'ordre de visite des files

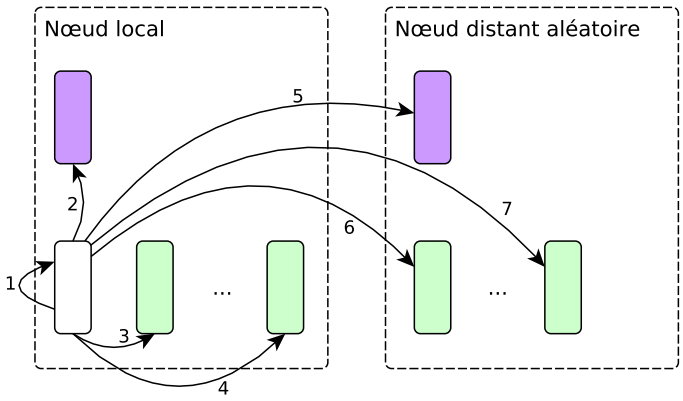

Figure 5.4: Illustration de la stratégie *sNumaProc*, avec l'ordre de visite des files

Ces deux stratégies pourront servir de base de comparaison avec les stratégies suivantes, prenant en compte les deux niveaux de hiérarchie disponibles, ainsi que la notion de « voisinage » entre cœurs.

Les quatre stratégies suivantes ont été définies : les deux premières font un usage important d'un seul niveau de hiérarchie, alors que les deux dernières utilisent les deux niveaux de hiérarchie disponibles, en donnant la priorité soit aux files des processeurs, soit aux files des nœuds NUMA.

sProc : Cette stratégie est illustrée sur la figure 5.3. Le voleur va visiter dans l'ordre : sa propre file (1), celle de son nœud NUMA (2), puis va parcourir uniquement les files des cœurs en commençant par ses voisins sur le nœud NUMA (3, 4), puis en choisissant au hasard parmi les cœurs distants restants (5, 6).

sNuma : Le voleur commence par visiter son nœud NUMA, puis les files des cœurs de son nœud, puis uniquement les files des nœuds NUMA distant.

sProcNuma : Dans un premier temps le cœur voleur visite sa propre file. Si aucune tâche n'est disponible, il va ensuite visiter les files de tâches des cœurs voisins situés sur le même nœud (dans l'ordre de leur numérotation). Si cela échoue de nouveau, il ira voler la file de son nœud NUMA. Si cela également, le reste de la topologie sera parcourue de manière similaire : un nœud sera choisi aléatoirement, les files des cœurs puis celle du nœud seront visitées.

sNumaProc : Cette stratégie est illustrée sur la figure 5.4 ; elle est similaire à la précédente, mais l'ordre de parcours est inversé : après avoir visité sa propre file (1) le voleur regarde d'abord les files des nœuds NUMA (2) avant de regarder les files des cœurs (3, 4). De même que pour la stratégie précédente, le reste de la topologie est parcourue de manière similaire (5, 6, 7).

5.3.2.3 Résultats préliminaires : comparaison des stratégies

Lorsque nous avons proposé ce travail a EuroPar 2016, nous avions implémenté ces stratégies dans XKaapi [2] et effectué nos expériences sur idchire. La configuration de la machine et de son environnement était significativement différente de celle présentée dans cette thèse : à la fois au niveau matériel étant donné que la machine était configurée pour utiliser des *huge pages*, et à la fois au niveau logiciel, les KASTORS et les supports exécutifs ayant subit plusieurs changements depuis. Une comparaison directe avec nos résultats récents de la section 5.4 serait trompeuse, nous faisons donc un point dès maintenant sur comment nos choix ont été guidés, avec les mesures que nous avions fait pour l'article en 2016. À titre indicatif, le pic de performance obtenu actuellement est légèrement au delà de 2300 GFlops, contre à peine 2000 GFlops en 2016.

La figure 5.5 regroupe les performances de ces stratégies sur un exemple représentatif de leur comportement : une factorisation de Cholesky sur une matrice de taille 32768 avec une taille de bloc de 512. Les meilleures performances de libGOMP via GCC (version 5.2.0) sont indiquées comme repère.

Parmi les stratégies choisies il y a :

- Deux stratégies «naïves» avec vol de travail aléatoire, soit au niveau des cœurs (1), soit au niveau des nœuds (2) ;

- Une stratégie prenant en compte l'affinité, mais n'effectuant la gestion des tâches qu'au niveau des nœuds (3) ;

- Quatre stratégies prenant en compte l'affinité et favorisant un placement des tâches à deux niveaux (dans la file du nœud NUMA si l'affinité pointe sur un nœud distant, ou dans la file du cœur courant si l'affinité pointe vers le nœud local). La différence entre les stratégies 4, 5, 6, et 7 réside dans la stratégie de sélection des victimes lors du vol de travail.

[2] https://scm.gforge.inria.fr/anonscm/git/kaapi/xkaapi.git, branche 'public/europar2016'

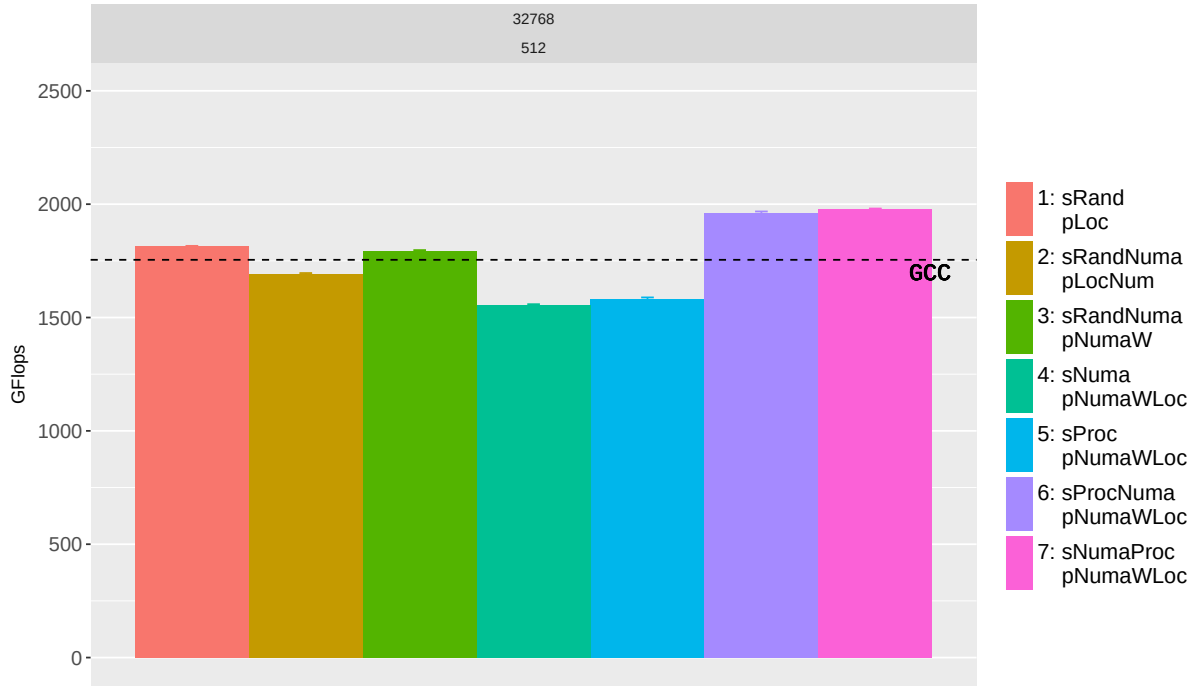

Figure 5.5: Performances des différentes stratégies pour Cholesky (N=32768, BS=512)

La première chose à remarquer est que les stratégies a priori naïves offre des performances tout à fait acceptables (GCC, 1, et 2) !

Deux stratégies sortent clairement du lot : 6 et 7. Leur spécificité commune est qu'elles utilisent complètement les deux niveaux de hiérarchie, tant lors du placement des tâches que lors de la sélection d'une victime à voler. Leur différence est uniquement l'ordre de parcours des files de tâches : 7 commence par essayer de voler les nœuds puis ensuite les cœurs, 6 fait l'inverse (une description détaillée avec des schémas est donnée dans la section 5.3.2).

Conserver un placement hiérarchique mais avoir une sélection principalement sur un seul niveau de hiérarchie n'est pas concluant (4 et 5). Prendre en compte l'affinité mais ne faire du placement ou de la sélection que parmi les files des nœuds NUMA (3) offre des performances juste équivalente à une stratégie basique de vol de travail (1).

Nous avons donc conclu que le plus bénéfique était de prendre complètement en compte la hiérarchie de la machine ainsi que le placement des données, et la stratégie 7 est celle que nous avons implémentée lorsque nous avons porté nos idées dans le support exécutif libOMP (décrit en section 5.4.1).

Conclusion

L'ensemble de ces modifications proposent un large choix de paramètres à ajuster pour le support exécutif, et ce n'est pas évident de voir a priori quelle(s) stratégie(s) devraient être privilégiée(s).

Y en a-t-il des meilleures que d'autres quelques soient les circonstances ? Est-ce que que cela dépend du type d'application ? Du type d'architecture ? La section suivante

propose une évaluation détaillée de l'impact des différents points, et vise à dégager des conseils généraux vis à vis du choix des stratégies.

5.4 Évaluation des extensions proposées

Nous avons évalué les différentes améliorations proposées dans les sections 5.2 et 5.3 sur les machines idchire et brunch, décrites en détails dans la section 4.2.

Pendant le déroulement de cette thèse, les développeurs de Clang ont décidé d'adopter et d'intégrer officiellement le support exécutif d'Intel open source pour leur support d'OpenMP, et l'ont nommé libOMP. Ce support exécutif dispose d'un support complet et robuste de la norme OpenMP 4.0, et utilise du vol de travail décentralisé (contrairement à libGOMP), avec également une découverte de la hiérarchie de la machine via hwloc. La section suivante décrit l'implémentation de nos idées dans le support exécutif libOMP, la section 5.4.2 décrit le reste des logiciels que nous avons utilisés ; et enfin la section 5.4.3 aborde point par point l'impact sur les performances des extensions proposées.

5.4.1 Portage dans libOMP

Comme indiqué dans la section 5.3.2.3, nous avions initialement implémenté nos idées dans XKaapi. La structure de libOMP nous a semblé être une base favorable pour intégrer nos travaux et favoriser leur diffusion. Cela nous permettait également d'ajouter des extensions directement dans Clang, tout en profitant au passage de la robustesse de son support d'OpenMP. Nous avons donc étendu ce support exécutif avec la plupart des mécanismes présents dans XKaapi : file de tâches non bornées (T.H.E), file de tâches hiérarchiques, et outils de génération de traces. Nous l'avons renommé libKOMP. Il s'agit là de la seconde version, bien différente de la version initiale publiée en 2012 par Broquedis et al. [Broquedis 2012].

5.4.1.1 Extensions et options ajoutées

Le support exécutif libOMP fonctionne, pour les tâches, par vol de travail. Chaque threads dispose d'une file de tâches, et les stratégies d'ordonnancement pour le placement et la sélection des files sont les suivantes, respectivement : les tâches prêtes sont placées dans la file du thread courant ; lorsqu'un thread a besoin de voler du travail, il sélectionne aléatoirement une file de tâche d'un autre thread, s'il réussit un vol il reviendra voler cette victime la prochaine fois qu'il aura besoin de voler du travail.

Dans un premier temps, nous avons donc commencé par ajouter un ensemble de structures de données : pour chaque nœud NUMA utilisé dans la *team* OpenMP courante, nous avons ajouté une file de tâches, permettant ainsi d'exposer un second niveau de hiérarchie dans l'ordonnancement. Nous avons également ajouté une file de tâche privée par thread et par nœud, pour implémenter la notion d'affinité stricte.

Dans un second temps, nous avons modifié les fonctions de placement des tâches prêtes et de sélection d'une victime à voler. Nous avons directement implémenté la combinaison de stratégies qui obtenait le meilleur niveau de performances parmi celles présentées dans la section 5.3.2, à savoir :

- pour l'affinité entre une tâche et une donnée, le placement de la tâche est effectué selon la stratégie *pNumaWLoc*, définie dans la section 5.3.2.1.

- la sélection d'une file de tâche à voler se fait hiérarchiquement, dans l'ordre indiqué par la stratégie *sNumaProc*, défini dans la section 5.3.2.2.

Dans la suite des expériences, nous ferons référence à cette deuxième version de libKOMP en l'appelant tout simplement *libKOMP*.

5.4.2 Logiciels

Les logiciels utilisés pour nos expériences peuvent être divisé en trois catégories :

- Les applications, qui proviennent des KASTORS ;

- Les supports exécutifs : libGOMP, libOMP, libKOMP ;

- Les bibliothèques externes : BLAS, hwloc, et numactl.

Certaines applications et supports exécutifs utilisés ont dû subir quelques modifications afin d'incorporer les modifications d'OpenMP proposées. Ces changements sont décrits ci-dessous. Nous n'avons effectué aucune modification aux bibliothèques externes, mais il est important de faire un point dessus, étant donné qu'une partie des performances peut en dépendre.

5.4.2.1 Applications utilisées

Les applications utilisées pour ces expériences proviennent des KASTORS[3]. Nous avons ajouté une clause *affinity* dans certaines applications : dans le cas des applications d'algèbre linéaire, les tâches de calculs dépendent d'un ou plusieurs blocs de données. Compte tenu des premiers résultats que nous avons observé précédemment [Virouleau 2016a], nous avons estimé qu'ajouter une affinité entre chaque tâche et les données qu'elle écrit serait avantageux. Dans le cas des applications de type stencil, nous avons ajouté une affinité vers un cœur précis pour les tâches successives.

5.4.2.2 Supports exécutifs

Nous avons pris comme base de comparaison les supports exécutifs fournis avec les compilateurs existants au moment de nos propositions.

GCC/libGOMP : nous avons utilisé la version 7.2.0 comme référence, sans y apporter de modification.

Clang/libOMP : nous avons utilisé la version 3.9. Bien que le support exécutif (libOMP) n'ait pas été modifié, le compilateur (Clang) a subit des modifications afin de supporter les clauses décrites dans la section 5.2.

[3] https://gitlab.inria.fr/openmp/kastors, branche 'affinity'

5.4.2.3 Bibliothèques externes

BLAS : les applications d'algèbre linéaire des KASTORS dépendent de la bibliothèque BLAS. Pour les expériences effectuées dans la section 5.4.3, nous avons utilisé la bibliothèque OpenBLAS 2.19 pour fournir les noyaux de calculs de base.

hwloc : cette bibliothèque fournit les informations sur la hiérarchie de la machine, ainsi que des fonctions d'allocation de mémoire selon différentes politiques (voir section 2.2.2). Nous avons utilisé la version 1.11.0.

numactl : nous avons utilisé *numactl* pour certaines courbes de référence. *numactl* est fournie par libNUMA ; nous avons utilisé la version par défaut fourni par le fabricant de la machine.

5.4.3 Résultats

Les résultats obtenus sont illustrés ci-dessous, dans trois sections abordant des points importants : la distribution des données, la prise en compte de la localité des données lors de l'exécution, et la possibilité de restreindre le vol de travail.

5.4.3.1 Impact de la distribution des données

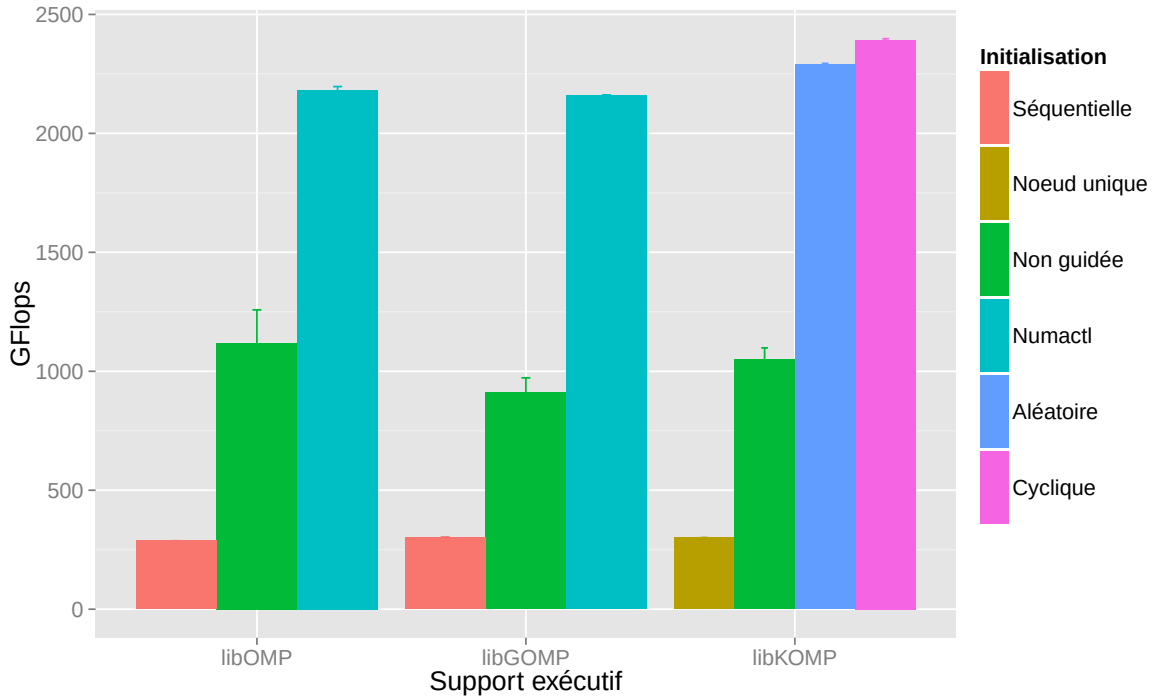

Figure 5.6: Performances, sur les 192 cœurs d'idchire, des différentes stratégies pour Cholesky en fonction de la distribution de données. Taille de matrice : 32768, taille de bloc : 512

La figure 5.6 montre un exemple représentatif du comportement de la factorisation de Cholesky par bloc, en fonction du support exécutif et du type de distribution

de données. La taille de matrice utilisée est 32768, et la taille de bloc est 512, et la performance affichée est celle obtenue avec l'ensemble des 192 cœurs d'idchire.

La distribution *Séquentielle* correspond à l'exécution du programme original. La distribution *Numactl* correspond à une distribution des pages des données effectuée de manière cyclique sur les nœuds de la machine à l'aide de *numactl*. Toutes les autres stratégies utilisent une initialisation parallèle dans laquelle chaque tâche est responsable de l'initialisation d'un bloc de données.

Parmi celles-ci, la distribution *Non guidée* correspond à une distribution des tâches (et donc du placement des données, comme expliqué dans la section 5.2.2) sur laquelle aucun contrôle n'a été effectué. Néanmoins il y a une certaine distribution des tâches qui est naturellement effectuée : compte tenu des tailles utilisées (64 blocs de largeur) et de la symétrie de la matrice, il y a 2080 blocs à initialiser et 24 nœuds sur la machine. En fonction de la rapidité d'initialisation du support exécutif, de la taille des files (si bornées), et de l'ordre dans lesquels les threads commencent à travailler, cette distribution non guidée peut varier.

La distribution *Nœud unique* correspond au placement de toutes les tâches d'initialisation sur un seul nœud (ce qui revient à faire une initialisation séquentielle). Les deux autres distributions, *Cyclique* et *Aléatoire* correspondent aux distributions que nous avons implémentées, et distribuent les tâche de manière cyclique ou aléatoire sur les nœuds de la machine.

La première chose à remarquer est que l'initialisation séquentielle de base (ou l'initialisation sur un nœud unique) propose sans surprise des performances désastreuses.

L'initialisation parallèle *Non guidée* se distingue par sa variabilité : les barres d'erreurs illustrent cette variabilité, due à l'ordre dans lesquels les threads viennent voler les tâches d'initialisation.

Au final l'utilisation de *numactl* permet de contrebalancer les effets de l'initialisation séquentielle et d'atteindre des performances raisonnables.

La différence entre *Numactl* et une distribution *Aléatoire* peut être expliquée par le fait que *numactl* distribue les *pages* de manière aléatoire, alors qu'une distribution aléatoire des tâches d'initialisation distribue des *blocs de données*, composés de plusieurs pages, ici 512 pages par bloc.

Ajouter une distribution de données spécifique assure que la distribution des tâches ne repose pas sur l'ordonnancement par défaut du support exécutif, et l'ordre parfois non contrôlé dans lequel les threads viennent voler du travail. Bien que la différence ne soit que de quelques dizaines de GFlops, la distribution *Cyclique* sur l'ensemble des nœuds NUMA semble être plus avantageuse que la distribution *Aléatoire* sur l'ensemble des stratégies d'ordonnancement testé, et c'est celle qu'on a retenu pour les expériences des sections suivantes.

5.4.3.2 Étude de l'impact de l'affinité

Dans cette section, les résultats de libKOMP ont été obtenus avec une distribution *Cyclique* et avec une combinaison de stratégies prenant en compte l'affinité et les deux niveaux de hiérarchie des machines. Nous avons étudié l'impact de l'affinité sous plusieurs angles :

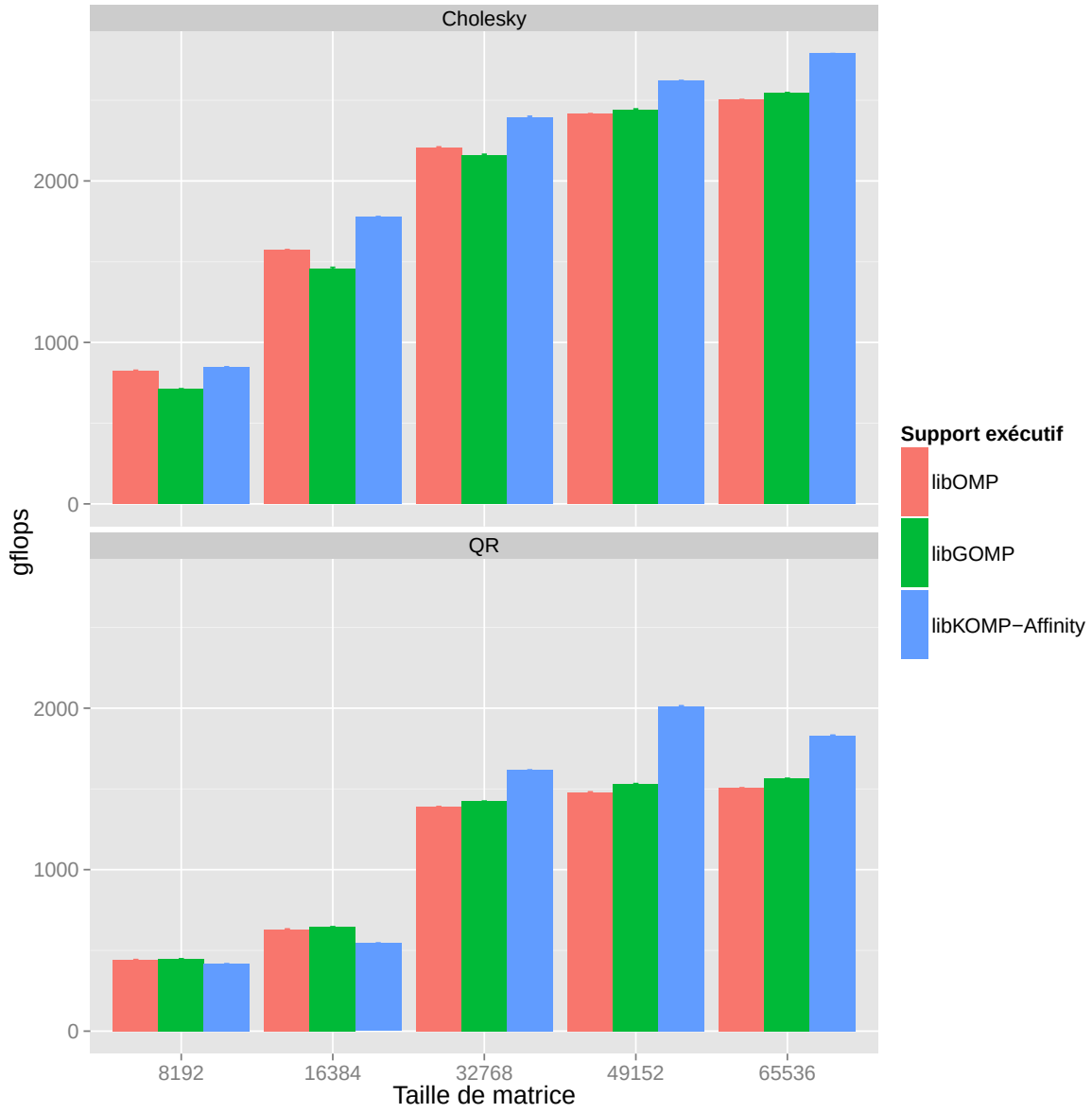

Figure 5.7: Comparaison des supports exécutifs sur Cholesky et QR sur idchire, en fonction de la taille de matrice (tailles de bloc données dans le tableau 5.1)

Performances brutes Pour avoir un aperçu général des stratégies que nous avons évaluées sur les factorisations Cholesky et QR, avec des tailles de matrice variant entre 16384 et 65536 ; les tailles de blocs correspondantes sont détaillées dans le tableau 5.1.

Taille de matrice	8192	16384	32768	49152	65536
Taille de bloc	256	256	512	512	512

Table 5.1: Tableau de correspondance entre taille de matrice et taille de bloc

La figure 5.7 montre les résultats obtenus sur idchire, et la figure 5.8 montre les résultats obtenus sur brunch.

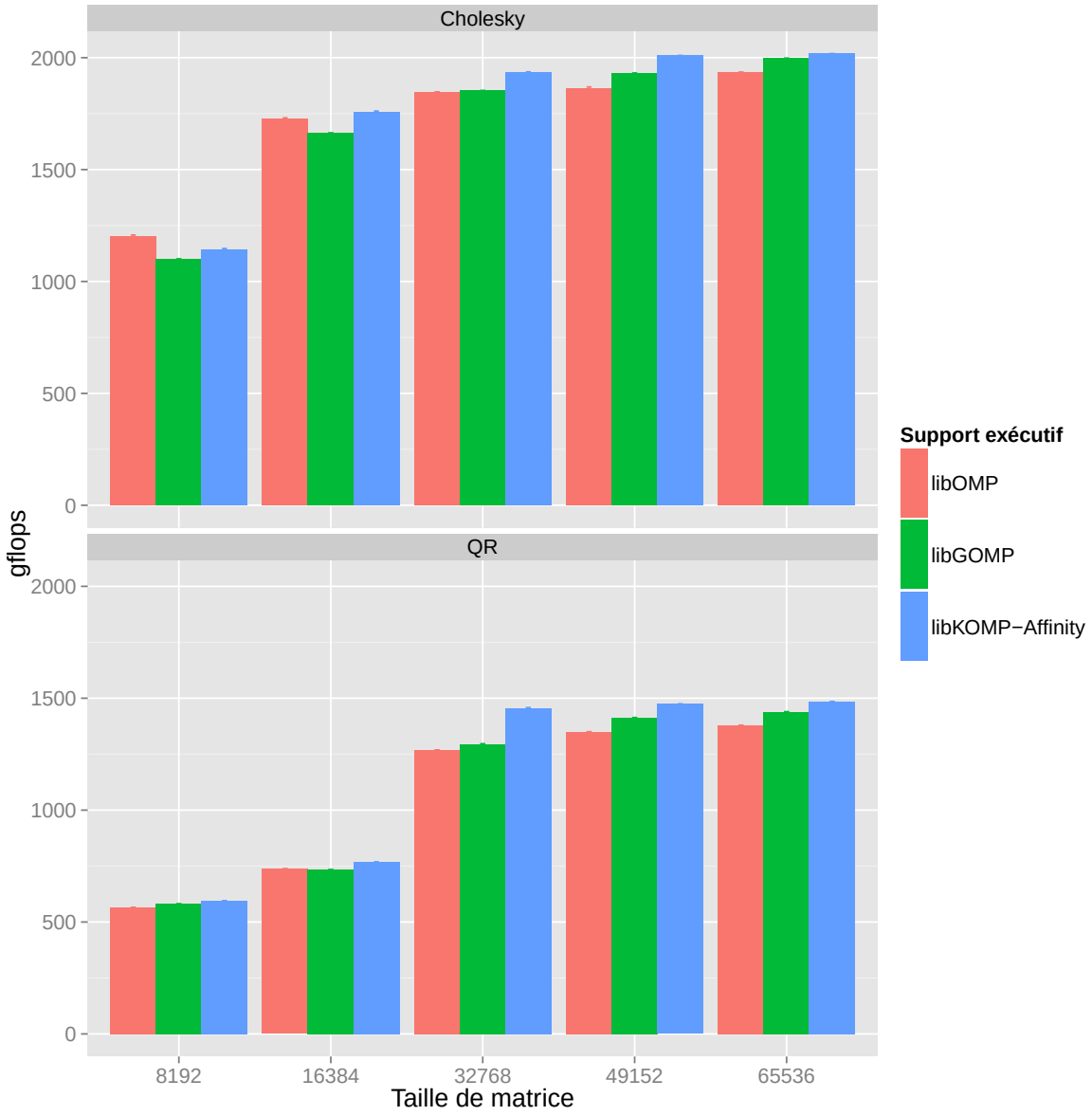

Figure 5.8: Comparaison des supports exécutifs sur Cholesky et QR sur brunch, en fonction de la taille de matrice (tailles de bloc données dans le tableau 5.1)

Les machines brunch et idchire ont un facteur NUMA différent : le coût d'accès à la mémoire distante par rapport à la mémoire locale est beaucoup plus important sur idchire que sur brunch. Malgré cela, l'impact de l'affinité est bien visible sur les deux machines, même s'il est proportionnellement plus important sur idchire.

On peut observer que pour les faibles tailles de matrices, l'intérêt de l'affinité semble limité. Plutôt que de regarder la taille de matrice, il faut en fait regarder la taille de bloc pour trouver l'explication de la différence : pour les tailles de matrice 8192 et 16384 la taille de bloc est de 256, pour les autres elle de 512. Les tailles de blocs induisent une différence claire vis à vis du cache L3 : dans le cas des blocs de taille 256 l'ensemble des données de calcul pour chaque noyau de l'application tient dans le cache L3, ce qui n'est pas le cas pour les blocs de taille 512. Compte tenu des résul-

tats montrés dans la section 4.3.3.3, qui ont montré la dégradation très importante des performances lorsque les données ne tenant pas dans le cache sont à distance, il est donc logique d'observer l'amélioration significative des performances pour de grande tailles de blocs (nécessaires pour les grandes tailles de matrices), et une différence relativement faible concernant les tailles de blocs plus petites.

Évolution en fonction du nombre de cœurs La figure 5.9 illustre l'évolution de l'impact de l'affinité en fonction du nombre de cœurs et de la taille de matrice sur idchire.

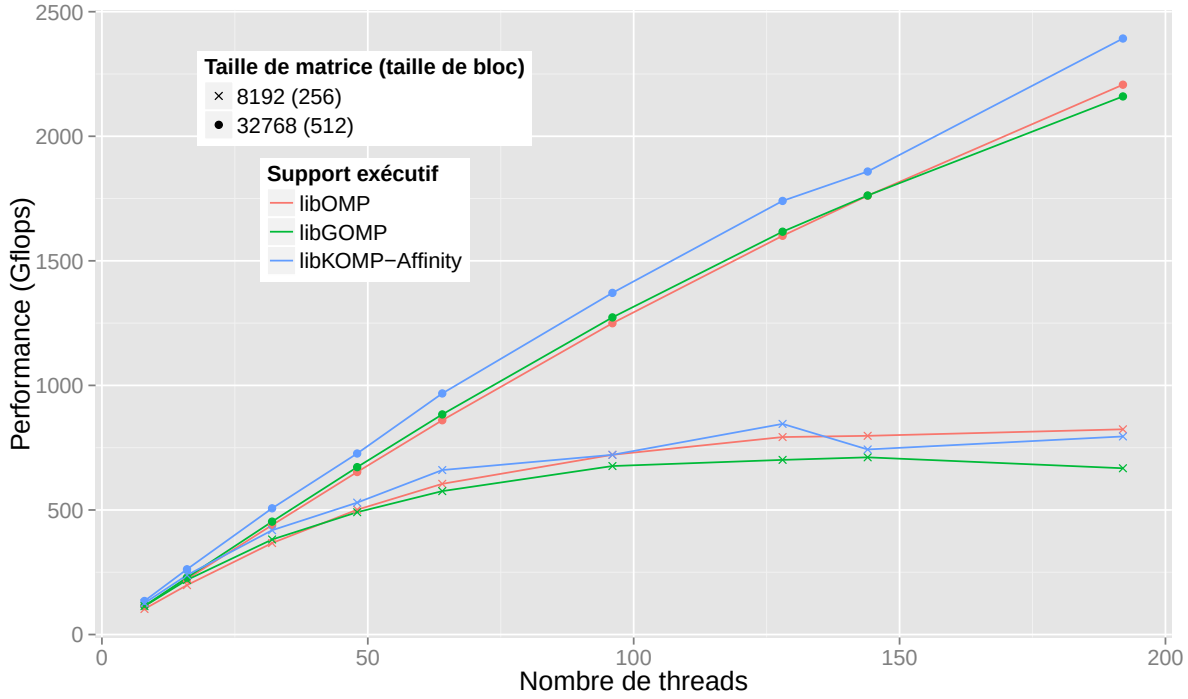

Figure 5.9: Performance en fonction du nombre de cœurs et de la taille de matrice sur idchire

Comme pour les expériences précédentes, les tailles de blocs correspondantes sont 256 pour la taille 8192, et 512 pour la taille 32768. Les deux tailles de matrice choisies représentent les deux situations décrites précédemment : dans un cas les données des noyaux de Cholesky tiennent dans le cache L3, dans l'autre non. Dans le cas d'une petite taille de bloc on peut constater que la courbe de libKOMP se situe entre libOMP et libGOMP, ne donnant donc aucun résultat concluant, voire même étant coûteux par rapport à un simple vol de travail aléatoire comme dans libOMP ! Pour une grosse taille de bloc en revanche c'est clair et net : après avoir suivi le même départ, les courbes divergent. Les supports exécutif libOMP et libGOMP affichent les même performances, tandis que l'affinité offre un net gain.

Les figures 5.10 et 5.11 montrent les résultats des factorisations Cholesky et QR (taille de matrice 32768, taille de bloc 512) en fonction du nombre de threads sur idchire et brunch, respectivement. Comparer les mêmes instances sur les deux machines donne des résultats intéressants : leurs caractéristiques ne changent pas l'allure

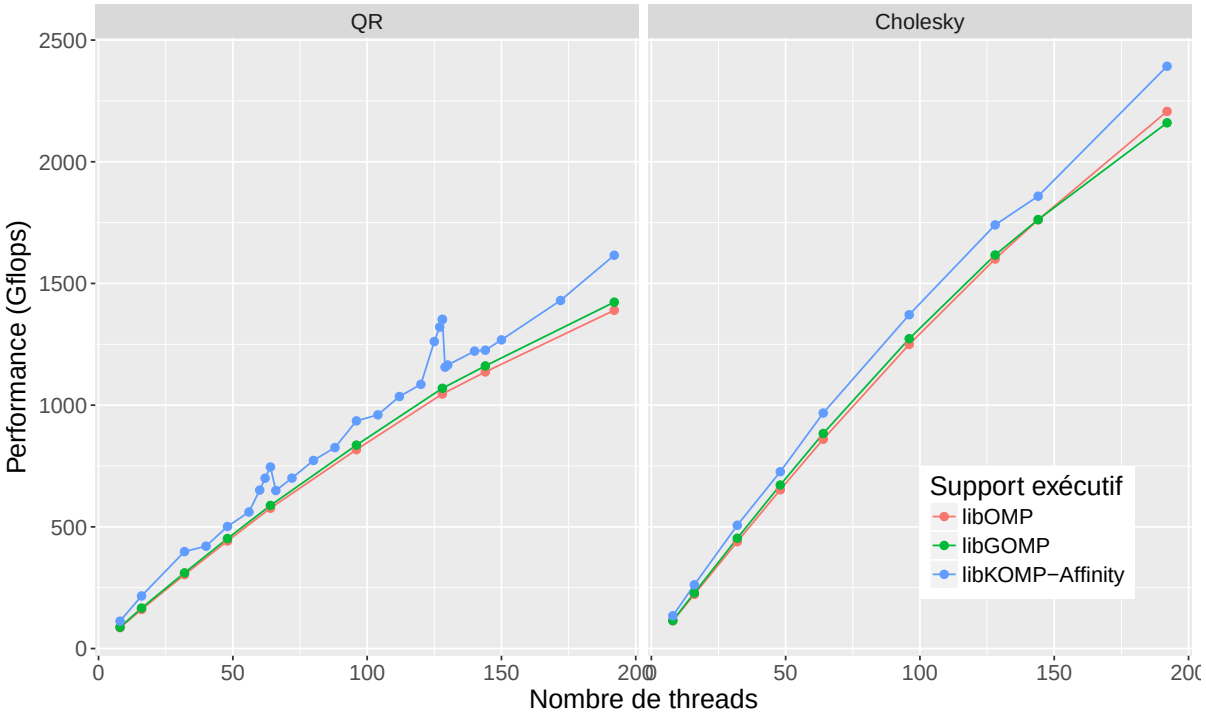

Figure 5.10: Performance de Cholesky et QR en fonction du nombre de cœurs sur idchire. Taille de matrice : 32768, taille de bloc : 512

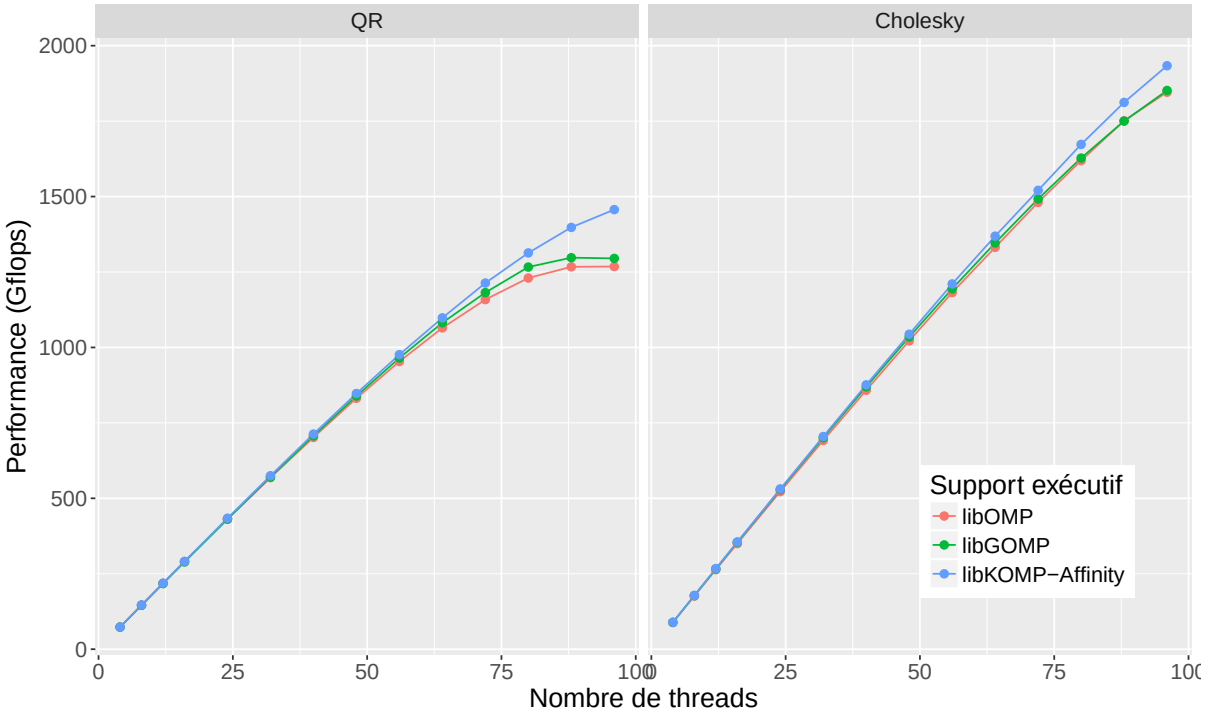

Figure 5.11: Performance de Cholesky et QR en fonction du nombre de cœurs sur brunch. Taille de matrice : 32768, taille de bloc : 512

général des courbes, en revanche le faible facteur NUMA de brunch fait bien sentir que l'affinité n'est rentable que lorsque la machine est chargée à partir d'un seuil élevé (environ 75%). Alors que sur idchire la différence se voit dès une trentaine de cœur (qui correspond d'ailleurs à 4 nœuds NUMA).

En faisant ces expériences nous avons constaté un comportement assez atypique de QR avec l'affinité : il y a 3 petits pics de performances à 32, 64, et 128 cœurs (soit 4, 8, et 16 nœud NUMA) ! Nous n'avons pas pu faire une étude aussi approfondie de QR que de Cholesky, il est donc dur d'expliquer ce comportement qui doit être dû à certaines caractéristiques de l'application.

5.4.3.3 Affinité stricte

La section 5.2.3 introduit la clause affinité en précisant que l'utilisateur peut spécifier une affinité *stricte*, restreignant ainsi les décisions d'ordonnancement du support exécutif. Cette fonctionnalité n'a pas été utilisée pour les applications d'algèbre linéaire, car dans les cas que nous avons étudiés il était plus rentable de payer le coût de transfert des blocs de données plutôt que de se priver du parallélisme disponible. Ce n'est évidemment pas le cas pour toutes les applications, et les applications stencil, comme Jacobi, peuvent grandement bénéficier d'une restriction d'affinité aux ressources proches, du fait de leur faible intensité opérationnelle (dans le cas de Jacobi, elle est en $O(1)$).

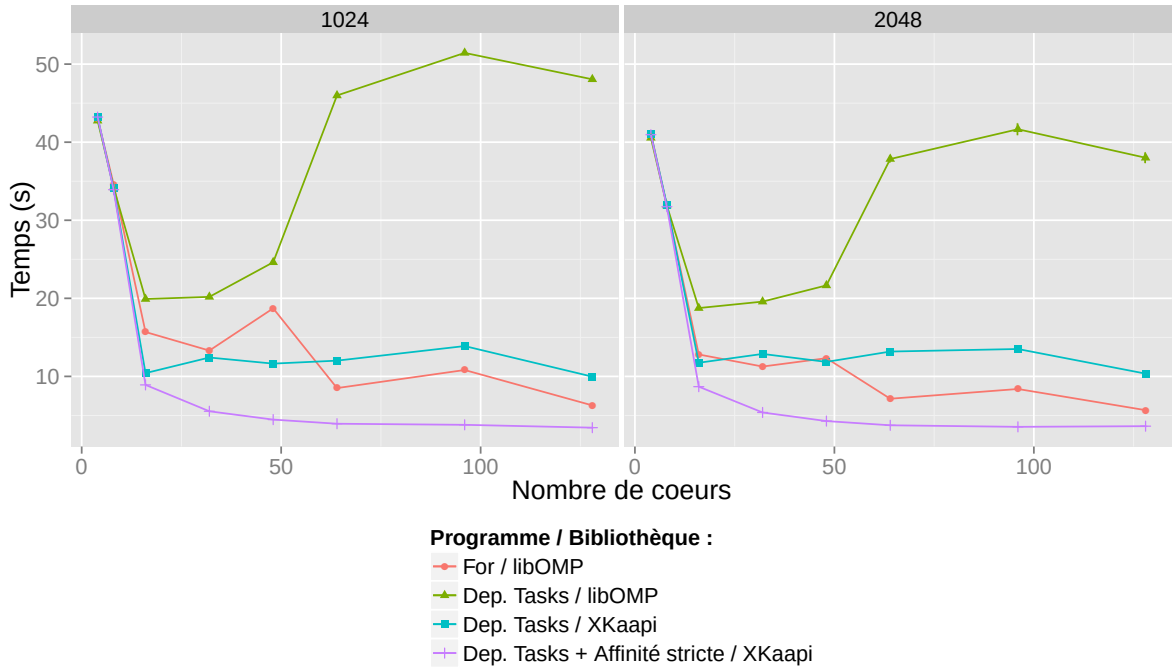

Figure 5.12: Performances de Jacobi en fonction de la version et du support exécutif, avec une taille de matrice de 49152, sur idchire

La figure 5.12 montre les performances de Jacobi sur deux tailles de blocs différentes. Les mesures avec l'affinité ont été effectuées avant le portage dans libOMP,

et le support exécutif sous-jacent est donc XKaapi. La version à base d'OpenMP `for` utilise un ordonnancement statique, sans contrôle particulier sur la distribution des données. Les versions à base de tâches avec dépendances n'utilisent aucun contrôle sur le placement des tâches ou des données. Pour la version avec affinité, une distribution de données a été rajoutée dans l'application, avec également une affinité stricte suivant cette distribution de données sur les itérations successives. Cette distribution affecte les blocs de données sur une grille la plus carrée possible en fonction des nœuds utilisés. L'affinité stricte ajoutée permet de garantir la réutilisation des données présentes dans le cache L2 (point critique pour cette application), et éviter des migrations de tâches inappropriées, ce qui est d'autant plus important que l'intensité opérationnelle est faible. Les performances obtenues avec les versions à base de boucles sont faibles comparativement aux résultats de l'affinité, néanmoins cela vient du fait qu'il n'y a pas eu d'effort particulier de fait pour que le découpage des itérations correspondent à celui des distributions des données. Les performances des deux versions devraient théoriquement être équivalentes : l'affinité stricte est là pour restreindre la portée du vol de travail qui dégradait les performances, phénomène qui ne devrait pas être présent dans une version à base de boucles.

Discussions et conclusion

Une analyse préliminaire des architectures et applications a permis de trouver et quantifier l'impact d'un paramètre majeur sur les performances des parties critiques d'application : la localité des données. Ce chapitre a montré qu'il était possible d'étendre un modèle de programmation et ses supports exécutifs dans but double : enrichir le graphe de tâches d'une application avec des informations sur l'affinité entre une tâches et ses données ; et utiliser efficacement ces informations lors de l'ordonnancement de l'application.

Parmi les améliorations possibles, nous avons pu constater que l'impact de la localité des données était directement liée à la quantité de données manipulées. On peut également supposer que le type d'opération effectuée sur ces données pourrait avoir un impact. De plus dans certains cas le compilateur pourrait fournir ce type d'information au support exécutif. Il y a donc une opportunité pour des futurs travaux sur l'amélioration de l'intéraction entre le compilateur et les supports exécutifs.

Se pose aussi la question de la performance du support exécutif : compte tenu des performances de référence des noyaux, est-ce que la performance obtenue via un ordonnancement par vol de travail est proche du maximum atteignable ? Existe-t-il un meilleur ordonnancement ? Nous avons commencé à développer un simulateur, présenté dans le chapitre 6, dont l'un des objectifs serait de répondre à ces questions.

6

Vers une amélioration possible du support exécutif à travers la simulation

Le chapitre 4 a montré qu'il était possible de caractériser précisément à la fois les machines et les parties critiques d'applications vis à vis de ces machines. Cela a pu confirmer et chiffrer l'importance de la localité des données sur les architectures NUMA. Le chapitre précédent a montré comment nous avons étendu un modèle de programmation et les supports exécutifs pour mieux prendre en compte la localité des données, et globalement améliorer l'ordonnancement de l'application.

À partir de là, certaines questions peuvent se poser : compte tenu des caractéristiques de l'application et des machines, peut on faire mieux ? Si oui : quelle marge reste-t'il à gagner par rapport aux ordonnancements théoriques connus ? Quelles caractéristiques faudrait il prendre en compte ?

Le coût en temps d'un développement de nouvelles analyses ou stratégies dans les compilateurs et supports exécutifs peut être important, il est donc logique de

se tourner vers la simulation. Pour des architectures hétérogènes ou distribuées, plusieurs simulateurs existent déjà : par exemple SimGrid [Casanova 2001], qui dispose d'une base d'utilisateurs importante, ou TaskSim [Rico 2010]. Stanisic et al. [Stanisic 2015] ont étudié la simulation de supports exécutifs dynamiques à base de tâches sur des architectures hétérogènes. Ils expliquent leur démarche pour porter StarPU par dessus SimGrid (et donc pouvoir émuler l'exécution du code réel), et présentent leurs résultats à la fois sur des architectures hétérogènes et sur des architectures NUMA. Leur modélisation précise des communications vers les GPUs leur permettent d'obtenir des simulations très précises pour les architectures hétérogènes. En revanche ils expliquent ne pas tenir compte des phénomènes NUMA pour des raisons de difficultés de modélisation : ils montrent donc des simulations précises pour la taille d'un nœud, mais qui s'éloignent complètement de l'exécution réelle lors de l'utilisation de plusieurs nœuds. À notre connaissance il n'y a pas eu plus de travaux sur SimGrid ou TaskSim concernant spécifiquement les architectures NUMA. Nous avons donc développé un prototype de simulateur, dans le but de pouvoir apporter une réponse aux questions ci-dessus, sans pour autant impliquer de lourds développements logiciel.

Ce chapitre est organisé de la façon suivante : la section 6.1 décrit l'architecture générale du simulateur. La section 6.2 fait un point sur les différents modèles envisagés, et la section 6.3 montre des résultats préliminaires obtenus avec ces modèles, en apportant des premières pistes de réponses aux questions posées. Enfin la section 6.4 décrit quels améliorations du simulateur seraient possible pour améliorer son réalisme.

6.1 Fonctionnement du simulateur

La figure 6.1 illustre la place du simulateur par rapport aux autres travaux de cette thèse. L'expression d'une application dans le simulateur se fait à travers une API dédiée. Elle peut soit se faire directement à l'aide d'une trace d'une exécution réelle (obtenue via libKOMP), soit manuellement en décrivant l'application de manière synthétique. Les coûts associés aux tâches peuvent être fournis au simulateur à l'aide de CarToN. La simulation a lieu en fonction du modèle d'architecture choisi et de la stratégie d'ordonnancement sélectionnée.

Le simulateur implémente un certain nombre de concepts que nous avons déjà abordés, détaillés ci-après.

Données : elles sont représentées comme des blocs de taille fixe, qui sont associés à un cœur lors de leur création (pour simuler la politique *first-touch* du système d'exploitation).

Tâches : elles sont représentées de manière similaire à ce qui se fait dans les supports exécutifs. Chaque tâche est composée d'une série d'actions séquentielles, pouvant être de trois types différents :

- *READ* : lecture d'un bloc de données
- *WRITE* : écriture d'un bloc de données

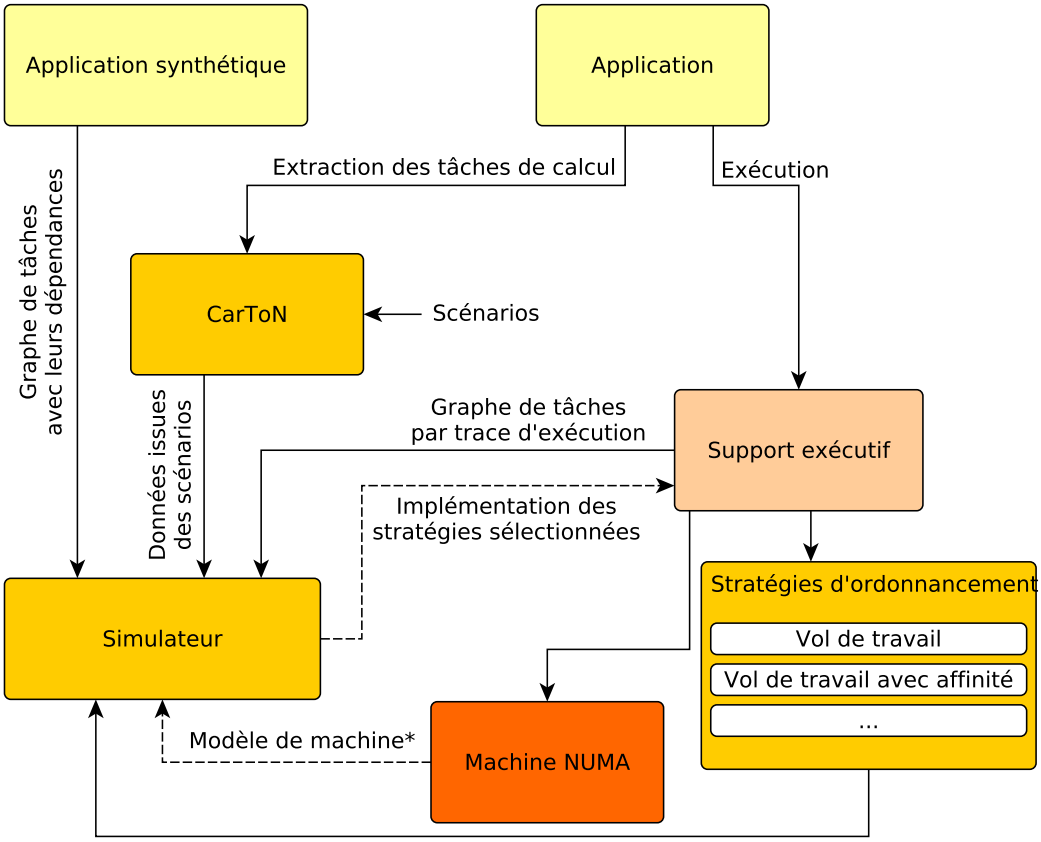

Figure 6.1: Schéma d'interaction des différents programmes.
* : partie envisagée mais encore non implémentée.

- *COMPUTE* : calcul avec un certain nombre d'instructions.

De plus des dépendances entre tâches peuvent être ajoutées en attachant un ensemble de prédécesseurs à une tâche donnée.

Topologie : De manière similaire à ce que nous avons utilisé lors de nos travaux, nous avons considéré ici deux niveaux de hiérarchie, avec une file de tâche prêtes par cœur, et une file de tâche prêtes par nœud.

Ordonnancement : L'objectif était de simuler le comportement d'un support exécutif similaire à ceux que nous avons utilisés dans le chapitre 5, nous avons donc basé l'ordonnancement au sein du simulateur sur du vol de travail. Le moteur d'exécution repose sur deux fonctions *steal* et *push*, devant implémenter les processus de vol et de placement d'une tâche, respectivement. Pour le besoin des premières comparaisons, nous avons implémenté deux heuristiques différentes, similaires à celles utilisées dans les résultats de la section 5.4 :

- *RandLoc* : elle implémente un vol de travail aléatoire et un placement des tâches prêtes localement ; elle est équivalente à la combinaison de stratégies *sRand/pLoc* décrites dans la section 5.3.2.2.

• *Affinity* : elle implémente un vol de travail hiérarchique et un placement des tâches conformément à l'affinité des données ; elle est équivalente à la combinaison *sNumaProc/pNumaWLoc* décrite dans la section 5.3.2.2.

Modèle de l'architecture : Un modèle fournit des informations cruciales : il définit la topologie de la machine (nombre de cœurs, de nœuds, ainsi que les associations cœur/nœud), il défini également le coût de chaque opération (exprimé en secondes) en fonction de son type et du type de la tâche effectuant l'opération.

6.2 Modèles de coût de tâches envisagés

Le simulateur décompose le temps de chaque tâche de la manière suivante :

$$T_{tache} = \sum_i T_{lecture}(P_i) + T_{execution} + \sum_i T_{ecriture}(P_i)$$

Où les P_i sont les paramètres en lecture et écriture de la tâche.

Avant de modéliser précisément l'architecture, nous avons dans un premier temps décidé de considérer les tâches dans leur ensemble. À travers CarToN il est possible d'obtenir des temps d'exécution des tâches dans lesquels la totalité des opérations est pris en compte ; en jouant sur les conditions d'exécution il est donc possible d'obtenir les T_{min} et T_{max} observés, et borner le temps d'exécution d'une tâche :

$$T_{min} \leq T_{tache} \leq T_{max}$$

Nous avons choisi de commencer à étudier l'application qui nous a servi de cas d'étude pour le chapitre 4 : Cholesky. Les données récoltées à l'aide de CarToN nous ont permis de comprendre précisément le comportement de chacun des quatre noyaux impliqués dans la factorisation de Cholesky. En particulier nous avons les T_{min} et T_{max} de chaque noyau en fonction de la charge de la machine, et le T_{tache} moyen en fonction du type d'accès aux données (local ou distant).

Nous avons implémenté un Cholesky synthétique correspondant à l'algorithme de l'application dans le simulateur, et considéré plusieurs modèles de coûts afin d'évaluer le simulateur et éventuellement les bornes de l'application en terme de performances.

Afin de pouvoir illustrer les différents modèles de coût que nous avons testés, le tableau 6.1 regroupe quelques exemples de performance pour les noyaux de Cholesky, pour une taille de matrice de 512, en fonction de la charge de la machine (comme décrit dans la section 4.3.3.1) et du type d'accès.

Cet aperçu des performances permet de montrer que si la localité des données n'a pas d'impact positif pour un unique thread, à partir du moment où tous les threads d'un nœud sont utilisés (et donc a fortiori en pleine charge) la différence de performance est significative.

Nous avons donc dégagé plusieurs modèles de coûts des tâches. Ces modèles se basent sur les expériences que nous avons pu faire avec CarToN. Au cours de ces expériences, nous avons fait varier deux paramètres qui peuvent avoir un impact majeur sur les performances : l'emplacement des données du noyau, et la charge de la machine. En faisant varier ces deux paramètres de manière extrême, nous avons pu obtenir un profil a priori complet des noyaux, pour nourrir les modèles de coût ci-dessous.

Noyau	Nombre d'exécution concurrentes (threads)	Performance moyenne par cœur (GFLOPS)	
		Données locales	Données distantes
POTRF	1	11.48	11.66
	8	11.08	11.11
	192	9.30	8.81
TRSM	1	14.20	14.38
	8	13.19	12.22
	192	10.00	9.15
SYRK	1	16.14	16.41
	8	13.97	13.59
	192	10.10	7.51
GEMM	1	16.92	17.10
	8	16.32	14.11
	192	14.45	12.82

Table 6.1: Tableau illustrant les performances réelles (en GFLOPS) des noyaux de Cholesky sur des matrices de taille 512, sur idchire

Modèle "Distant" : ce modèle a pour but d'estimer la performance observée que l'on est a priori en droit d'attendre d'un support exécutif implémentant une stratégie de vol de travail naïve sur une architecture NUMA. L'idée est de prendre un cas extrême pour l'exécution de chaque noyau, avec un cas extrême pour l'emplacement des données (toutes les données à distance), et une charge complète de la machine. En terme de scénario, sur idchire, cela revient à exécuter 192 noyaux simultanément, avec leurs données placées sur un nœud distant, de manière similaire à ce que nous avons effectué pour la figure 4.14. Pour les différentes tailles de blocs, en pleine charge de la machine, nous avons pris le minimum observé parmi les placements de données effectués.

Modèle "Maximum" : à l'inverse l'objectif de ce modèle est de donner une borne supérieure pour les performances du support exécutif. Nous avons donc considéré seulement la performance a priori maximale de chacun des noyaux pour le coût de chaque tâche, et va donc permettre d'obtenir une borne supérieure pour les performances globales de l'application. Cela correspond à l'autre extrême en terme de distribution de données : toutes les données locale, et à une exécution non perturbée par d'autres noyaux.

Modèle "DeuxNiveaux" : l'objectif de ce modèle est d'essayer de simuler au plus près le comportement des noyaux, en utilisant deux niveaux de coût : soit celui correspondant à l'exécution avec des données locales, soit celui correspondant à l'exécution avec des données distantes. La simulation étant lancée sur un nombre de cœurs connus, nous avons utilisé les performances de référence des noyaux correspondant à la charge de la machine simulée, que nous avons obtenues à travers les scénarios exécutés par CarToN. Lors de la simulation nous déterminons si les accès en écriture de la tâche sont locaux ou distant, et le coût de la tâche correspondante est utilisé.

6.3 Résultats préliminaires

Nous avons commencé par comparer les modèles entre eux. Pour ce faire nous avons choisi un cas réel ou l'affinité avait un impact significatif : une taille de bloc de 512 pour une taille de matrice de 32768, avec des blocs de données répartis de manière cyclique sur la machine. Les résultats des simulations avec les différents modèles sont montrés sur la figure 6.2.

Figure 6.2: Comparaison des différents modèles et stratégies, basés sur les références d'idchire. Taille de bloc : 512, taille de matrice : 32768

Les premières observations à faire sur cette figure sont plutôt positives : les performances affichées semblent réalistes, le modèle *DeuxNiveaux* est correctement encadré par les modèles *Distant* et *Maximum*, et au sein du modèle *DeuxNiveaux*, il y a bien une différence claire entre un vol de travail «naïf» — *RandLoc* — et un vol de travail hiérarchique et sensible à l'affinité — *Affinity*.

Cela est confirmé en regardant le nombre de lectures et écritures locales ou distantes, rapportées dans le tableau 6.2. Ce tableau montre que le nombre d'accès dis-

Stratégie	Lectures		Écritures	
	Locales	Distantes	Locales	Distantes
RandLoc	4 380	117 066	1902	43 858
Affinity	15 343	88 045	22 556	23 204

Table 6.2: Nombre de lectures et écritures de blocs locaux ou distants en fonction de la stratégie de vol

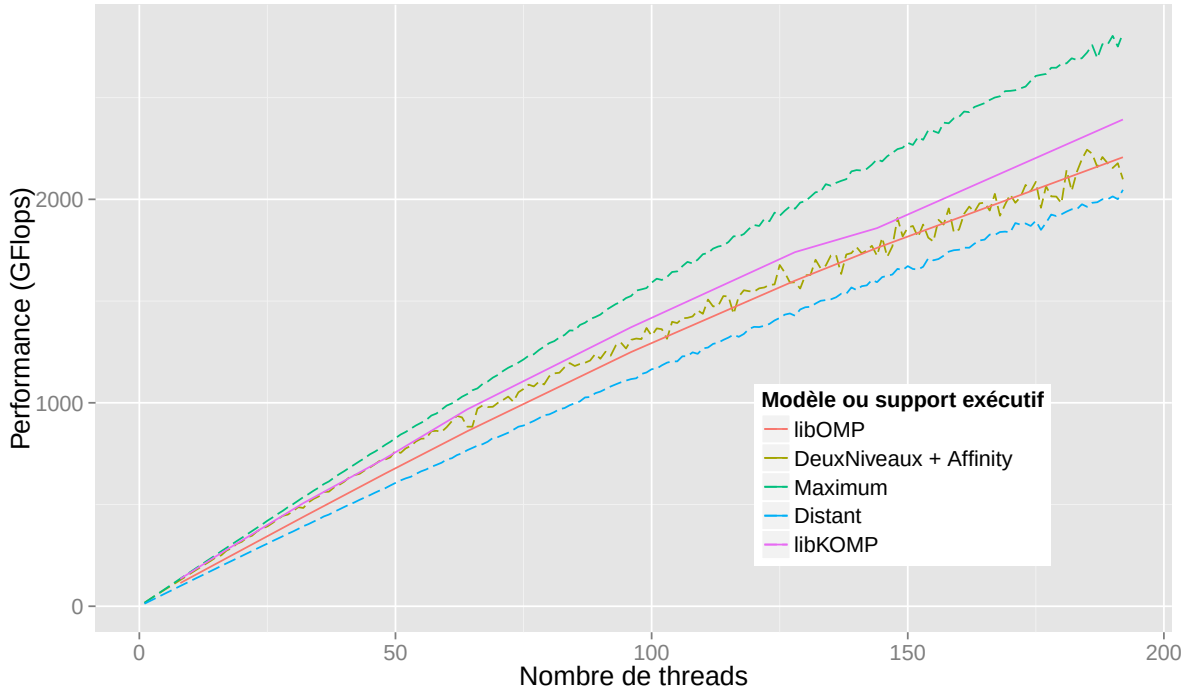

Figure 6.3: Comparaison des certains modèles aux supports exécutifs sur idchire, pour une taille de matrice de 32768 et une taille de bloc de 512

tants est significativement diminué par l'utilisation d'une stratégie de vol de travail sensible à l'affinité : le nombre d'écritures distantes est réduit de moitié, tandis que le nombre de lectures distantes est réduit d'environ 25%.

Ces figures sont bien sur issues de simulation ; que valent elles en comparaison aux chiffres obtenus à travers les expériences ? La figure 6.3 compare les modèles *Distant* et *DeuxNiveaux* (avec vol hiérarchique) aux supports exécutifs libOMP (qui utilise un vol aléatoire) et libKOMP (qui utilise un vol hiérarchique et l'affinité). Comme on peut le constater, les performances des deux supports exécutifs sont effectivement supérieures au minimum simulé. En revanche les performances simulées de l'affinité semblent légèrement en retrait compte tenu des performances réelles.

Cette différence pourrait sembler étonnante, mais pourrait être en partie expliquée par le fait que les performances de référence ont été obtenues lorsque les données étaient soit *toutes* locales ou *toutes* distantes. Alors que dans la réalité il peut évidem-

ment y avoir plusieurs autres cas quand plusieurs blocs sont en paramètre des noyaux. Cela donnerait donc une version «minimum» des performances plus pessimistes que la réalité. Ces travaux étant en cours, plusieurs pistes d'amélioration sont envisagées, et sont décrites dans la section 6.4.

Les résultats obtenus sur les cas favorable à l'utilisation de l'affinité sont assez encourageants. Nous avons également appliqué la simulation sur des tailles de bloc plus petites, typiquement pour une taille de matrice de 8192 avec des blocs de 256.

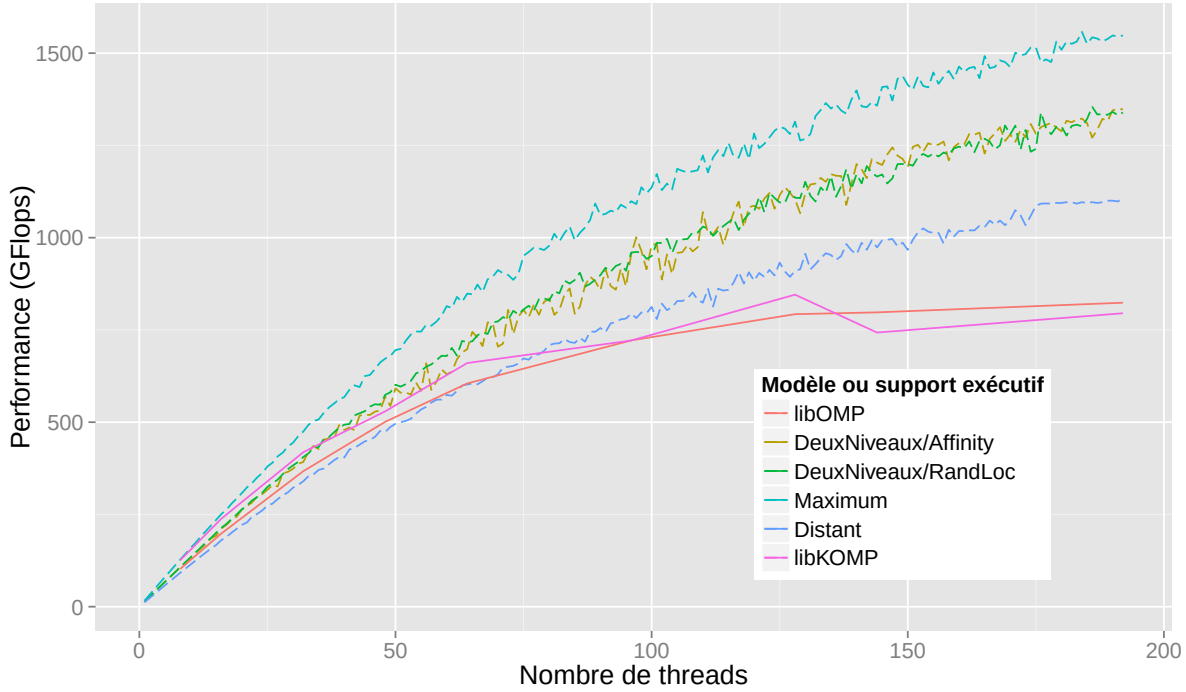

Figure 6.4: Comparaison du modèle *DeuxNiveaux* aux supports exécutifs sur idchire, pour une taille de matrice de 8192 et une taille de bloc de 256

Les résultats obtenus sont présentés sur la figure 6.4. Comme on a pu le constater sur les expériences réelles, la simulation ne montre également aucune différence de performances avec ou sans l'utilisation de l'affinité.

En revanche, passé un certain nombre de cœurs, les performances des deux supports exécutifs sont assez loin de celles a priori atteignables ! Le nombre de tâches dans une telle configuration peut vite limiter le parallélisme exposé, ce qui va donc naturellement entrainer une augmentation importante du nombre de requêtes de vol émises par les threads inactifs.

Ce phénomène est illustré sur la figure 6.5, où l'on voit l'évolution du nombre de requêtes de vol divisé par le nombre de tâches à exécuter, au sein du simulateur. Pour un cas exposant un fort parallélisme, une taille de matrice de 32768 avec une taille de bloc de 512 et donc plus de 45000 tâches, on peut constater que le nombre de requêtes par tâche reste raisonnable. En revanche pour le cas étudié dans la figure 6.4, une taille de matrice de 8192 avec une taille de bloc de 256 et donc un peu moins de 6000 tâches, on peut constater que l'évolution du nombre de requêtes de vol par tâche est exponentielle, ce qui pourrait avoir un impact sur les performances.

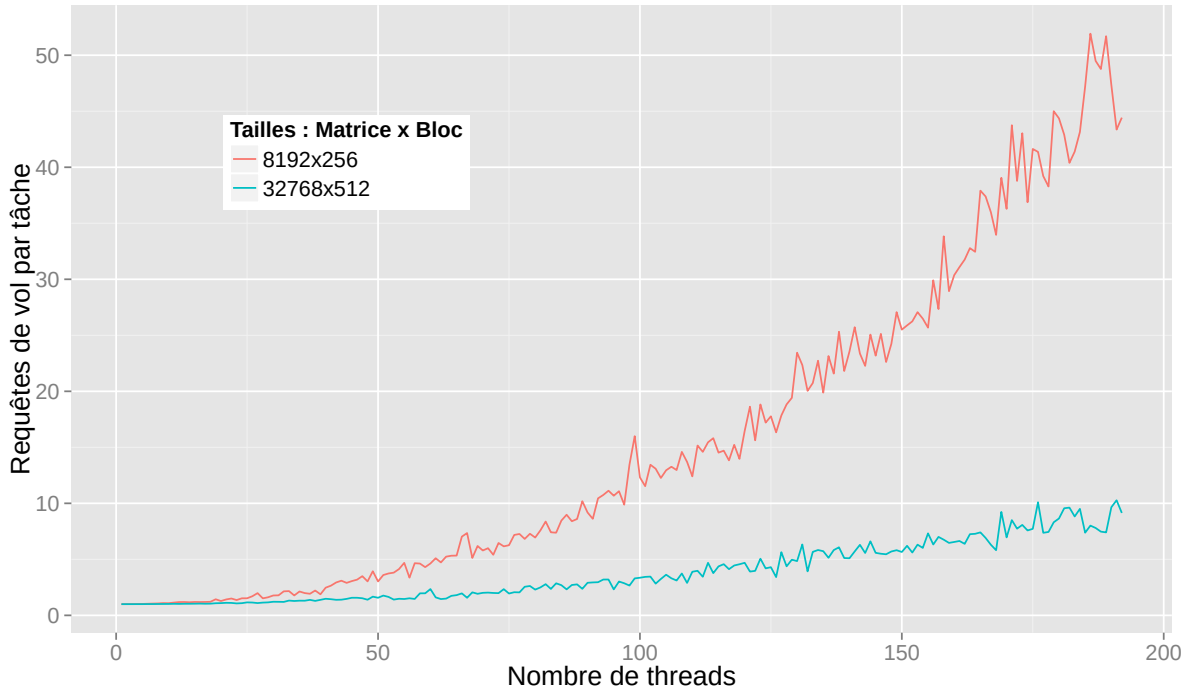

Figure 6.5: Comparaison de l'évolution du nombre de requêtes de vol par tâche dans le simulateur, en fonction de la taille de matrice

Noyau	Performance minimum attendue (GFlops)		Performance moyenne observée (GFlops)	
	8192 (256)	32768 (512)	8192 (256)	32768 (512)
POTRF	6.55	8.79	5.07	8.74
TRSM	7.82	9.15	5.55	9.76
SYRK	9.25	7.51	5.80	11.85
GEMM	12.19	12.82	9.41	14.38

Table 6.3: Comparaison des performances minimum attendues aux performances moyennes observées pour l'exécution de Cholesky sur 192 cœurs, avec une taille de matrice de 8192 et des blocs de taille 256

Pour une taille de matrice de 8192 (taille de bloc 256) et avec 192 cœurs, nous avons observés les temps moyens passés dans chaque tâche, rapportés par les traces de libKOMP. Nous les avons comparé aux performances minimum considérées par le simulateur dans le tableau 6.3.

Nous avons constaté une forte différence entre les performances réelles et les coûts de tâches considérés par le simulateur. La différence observée sur la figure 6.4 ne semble donc pas venir d'un surcoût d'exécution du support exécutif lié à la quantité de requêtes de vol exécutées. En revanche les requêtes de vol peuvent générer du trafic sur les bus mémoires : non seulement les threads ne participent plus activement

à l'exécution du programme, mais en plus ils peuvent participer à la saturation de la bande passante et donc dégrader les performances des threads actifs. Cela pourrait expliquer en partie la différence observée dans le tableau 6.3, et pourrait être prise en compte lors de la modélisation des coûts de communication.

Évaluation du temps de simulation

Un autre avantage de la simulation est qu'il n'y a pas besoin d'exécuter réellement les noyaux d'exécution, seulement de considérer le temps qu'ils prendraient. Bien qu'une étude détaillée n'ait pas été réalisée, le temps nécessaire pour simuler les plus gros cas (comme la figure 6.3) est de seulement quelques secondes, contre plusieurs heures qui seraient nécessaires pour générer les même courbes.

6.4 Discussions et améliorations possibles

Nos travaux sur la simulation sont très récents et encore en cours de développement. Afin de généraliser la portée du simulateur et améliorer sa précision, nous évoquons ici les différentes pistes en cours d'étude.

6.4.1 Modèle *Minimum*

Pour venir en complément de notre modèle *Maximum*, nous souhaiterions construire un modèle de coûts *Minimum* qui nous permettrait de donner une borne inférieure pour les performances de notre applications. Pour construire notre modèle de coûts *Distant*, nous avons considéré une charge de la machine avec des noyaux s'exécutant indépendamment, qui n'est pas assez représentatif des «pires» conditions réelles dans lesquelles peut se retrouver un noyau. Il faudrait donc utiliser CarToN pour venir ajouter des perturbations réalistes sur les ressources utilisées pendant l'exécution d'un noyau, afin de retrouver un pire cas d'exécution tel que constaté dans le tableau 6.3.

6.4.2 Modélisation du cache L3

La version actuelle du simulateur inclu un cache par cœur de taille infinie. Ce cache est modélisé comme une liste de blocs, dans une *version* particulière. À chaque écriture du bloc, la version du bloc est incrémentée. Si une tâche effectue une lecture de ce bloc et que sa version correspond à la version du cache, alors la lecture ne coûte rien, sinon le cout de lecture est demandé au modèle.

Comme indiqué dans les sections précédentes, les modèles envisagés pour l'instant se basent uniquement sur le cout global d'une tâche, plutôt que sur des actions séparées.

L'une des améliorations nécessaires pour augmenter la précision du modèle serait l'introduction d'un cache partagé entre les cœurs d'un même nœud, disposant d'une capacité définie par le modèle, et implémentant une politique d'éviction relativement proche de celle utilisée par le matériel : *LRU*.

6.4.3 Modélisation de la bande passante

Le chapitre 4 nous a permis d'étudier en détail le comportement de nos deux machines expérimentales. Les figures 4.3 et 4.4 permettent de conclure par exemple que la bande passante d'idchire se comporte d'une manière assez proche d'un modèle de réseau limité. L'objectif serait donc d'introduire un coût d'accès aux blocs mémoires suivant ce modèle.

6.4.4 Modélisation de l'impact des requêtes de vol

Nous avons observé une différence très importante entre la simulation et l'exécution réelle sur la figure 6.4. La figure 6.5 et le tableau 6.3 ont permis de déduire que si les requêtes de vol n'entrainent pas de surcoût dans le support exécutif, elles pourraient néanmoins avoir un impact sur les performances des noyaux. Ces requêtes sont due au parallélisme limité des cas étudiés, et elles peuvent contribuer à la saturation de la bande passante.

Il faudrait donc pouvoir estimer l'impact des requêtes de vol sur le trafic mémoire. Cela semble assez compliqué à exprimer à l'aide d'un scénario de CarToN, mais cela pourrait sûrement se faire au sein de libKOMP : il devrait être possible de faire exécuter des tâches sur les threads d'un unique nœud via une affinité stricte, et d'augmenter progressivement le nombre de requêtes de vols en ajoutant des threads en dehors de ce nœud (qui ne pourrait pas voler de tâche avec succès à cause de l'affinité stricte).

6.4.5 Optimisation de la distribution

Le simulateur permet d'obtenir des informations précises sur la distribution de données et son impact sur les accès locaux et distants. Comme on l'a vu dans le tableau 6.2, le vol de travail respectant l'affinité permet de réduire considérablement les accès distants. Il devrait donc être possible d'utiliser le simulateur pour automatiquement trouver la «meilleure» distribution de données (c'est à dire celle qui minimise les accès distants), en se basant sur une distribution cyclique et en introduisant des paramètres tels que le premier nœud considéré, le nombre de blocs alloués en même temps, et le pas lors du parcours des nœuds.

« That's all folks! »

7 Conclusion et perspectives

Le calcul haute performance répond aux besoins toujours plus grand de la simulation, et les supercalculateurs embarquent un nombre très important de machines à mémoire partagée. La tendance dans ces machines est à la multiplication du nombre de cœurs de calcul plutôt qu'à l'augmentation de leur fréquence. Les architectures ont donc évolué, et pour répondre aux besoins en terme d'accès à la mémoire elles sont maintenant découpées en plusieurs nœuds, regroupant des cœurs de calcul et de la mémoire. Cela introduit du même coup un temps d'accès à la mémoire non uniforme pour les différents cœurs, d'où le qualificatif NUMA — Non Uniform Memory Access — pour ce type d'architectures.

Pour en faire une exploitation efficace, les applications scientifiques devraient prendre en compte la hiérarchie mémoire imposée par le matériel. Mais cela n'est possible que si les modèles de programmation et les supports exécutifs utilisés par ces applications évoluent pour supporter ces architectures. Les modèles de programmation à base de tâche sont particulièrement adaptés pour cibler les architectures NUMA : l'aspect dynamique de l'ordonnancement d'un graphe de tâches permet d'assurer un équilibrage de charge efficace même lors d'irrégularités lors des calculs. De plus cela permet d'exprimer un parallélisme à grain fin, nécessaire pour fournir du travail au nombre important de cœurs de calcul.

OpenMP est le standard *de-facto* pour les architectures à mémoire partagée. Les évolutions du langage ont montré qu'il pouvait s'adapter au nouveau matériel, et il propose maintenant un ensemble de fonctionnalités autour du paradigme de la programmation par tâches. Néanmoins les concepts spécifiques aux architectures NUMA sont encore peu présents : bien que des efforts récents aient été faits pour contrôler le placement des threads de calcul sur la topologie de l'architecture, il n'y a rien pour aider le support exécutif à conserver la localité des données au cours de l'ordonnancement.

Contributions de cette thèse

Les travaux de cette thèse ont porté sur plusieurs facettes liés au développement et à l'exécution d'applications scientifiques sur les architectures NUMA.

Dans un premier temps nous avons introduit CarToN, un outil permettant de faciliter la caractérisation de machines et d'applications scientifiques, à travers des scénarios définis par l'utilisateur. Nous l'avons utilisé pour évaluer la hiérarchie mémoire de nos machines d'expérimentation, idchire et brunch. Nous l'avons également utilisé dans le contexte d'une étude de cas sur la factorisation de Cholesky, où il nous a permis d'évaluer précisément le comportement des quatre noyaux sur lesquels la factorisation repose. Nous avons confirmé et mesuré l'impact de la localité des données sur ces noyaux, et identifié un axe possible d'amélioration pour l'ordonnancement de ce type d'application.

Dans un second temps nous avons donc introduit des extensions au modèle de programmation utilisé pour cette application, OpenMP, permettant aux programmeurs : de contrôler la distribution des données lors de l'initialisation de l'application, et d'exprimer une *affinité* entre une tâche et une donnée ou un élément de la topologie.

Nous avons implémenté ces propositions dans le compilateur Clang. Ces changements ont été accompagnés de l'extension de deux supports exécutifs : XKaapi, un support exécutif expérimental, et libOMP, le support exécutif grand public de l'infrastructure LLVM. L'objectif des extensions a été à la fois de pouvoir utiliser l'affinité exprimée dans le programme, mais aussi d'exploiter les plusieurs niveaux de hiérarchie présentés par la machine lors du vol de travail. Les résultats des expériences que nous avons menées ont confirmé et ont permis d'apprécier le gain de performances lié à l'affinité sur un ensemble d'applications. Ils ont aussi permis de mettre en avant que l'intérêt de l'affinité était principalement lié à la taille des données manipulées par les tâches.

Enfin nous avons pu commencer des développements autour d'un simulateur, avec deux objectifs principaux : pouvoir tester de nouvelles heuristiques sans avoir de lourds coûts de développement à payer dans les supports exécutifs ; et pouvoir comparer les performances réelles des supports exécutifs par rapport à ce qu'il serait théoriquement possible d'atteindre. Les résultats préliminaires obtenus sont encourageants et confirment l'intérêt d'une telle approche.

Perspectives

Au delà de la simulation mentionnée dans la section précédente, cette thèse ouvre plusieurs perspectives.

La première et la plus importante semble être **l'amélioration de la collaboration entre le compilateur et le support exécutif**. Le compilateur peut dans certains cas avoir accès à des informations critiques concernant les différentes tâches. En supposant qu'il ait accès à toute l'application, il peut alors connaître la taille des données manipulées en fonction de l'instance de l'application considérée, ainsi que le type d'opérations effectuées sur ces données. Transmettre ces informations au support exécutif pourrait lui permettre de prendre de meilleures décisions quand au placement des tâches. Une piste que nous n'avons pas eu le temps d'explorer concerne l'*intensité*

opérationnelle des tâches (le nombre d'instructions effectuées par rapport aux données utilisées) : a priori une tâche avec une grande intensité serait plus «rentable» à voler qu'une tâche de faible intensité, où la performance est complètement dirigée par le temps d'accès à ses données.

Une deuxième perspective serait **d'appliquer des idées similaires dans un contexte hétérogène**. L'arrivée de constructions pour supporter les accélérateurs dans OpenMP offre une porte d'entrée standard pour attaquer des supports exécutifs hétérogènes. Les idées développées au cours de cette thèse pourraient être étendues pour attacher des informations aux constructions *target*, et permettre au support exécutif de faire des choix pertinents vis à vis des tâches à exécuter sur le ou les accélérateurs.

Enfin les architectures vont continuer à évoluer, et les fabricants eux même ne sont pas toujours sûr de la manière de se préparer à l'exascale. Après avoir sorti en 2012 le *Xeon Phi*, un processeur multicœurs sous forme d'accélérateur avec une soixantaine de cœurs physiques, Intel a rapidement annoncé une deuxième mouture du processeur montée sur socket l'année suivante, et a finalement annoncé l'abandon des nouvelles versions en novembre 2017, au profit d'une nouvelle microarchitecture adaptée à l'exascale. Le simulateur que nous avons développé peut donc ici répondre à un besoin de tester les nouvelles architectures : il peut permettre de donner un aperçu des performances des applications compte tenu des algorithmes d'ordonnancement connu, et peut également servir pour prototyper de nouveaux algorithmes d'ordonnancement.

Compte tenu des changements fréquent en terme de topologie des machines, les propositions qui ont été faites au cours de cette thèse constituent une bonne base pour permettre aux supports exécutifs de s'adapter au nouveau matériel.

Bibliographie

[Abel 2014] Andreas Abel et Jan Reineke. *Reverse engineering of cache replacement policies in intel microprocessors and their evaluation*. Dans Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on, pages 141–142. IEEE, 2014. *cité page 24*

[Agullo 2016] Emmanuel Agullo, Olivier Beaumont, Lionel Eyraud-Dubois et Suraj Kumar. *Are Static Schedules so Bad? A Case Study on Cholesky Factorization*. Proceedings - 2016 IEEE 30th International Parallel and Distributed Processing Symposium, IPDPS 2016, pages 1021–1030, 2016. *cité page 44*

[Al-Omairy 2015] Rabab Al-Omairy, Guillermo Miranda, Hatem Ltaief, Rosa M. Badia, Xavier Martorell, Jesus Labarta et David Keyes. *Dense Matrix Computations on NUMA Architectures with Distance-Aware Work Stealing*. Supercomputing Frontiers and Innovations, vol. 2, no. 1, pages 49–72, 2015. *cité page 58*

[Al-Zoubi 2004] Hussein Al-Zoubi, Aleksandar Milenkovic et Milena Milenkovic. *Performance Evaluation of Cache Replacement Policies for the SPEC CPU2000 Benchmark Suite*. Dans Proceedings of the 42Nd Annual Southeast Regional Conference, ACM-SE 42, pages 267–272, New York, NY, USA, 2004. ACM. *cité page 24*

[ARM 2010] ARM. *ARM Cortex-R series processors manual*, 2010. *cité page 25*

[Aslot 2001] Vishal Aslot, Max Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B Jones et Bodo Parady. *SPEComp: A new benchmark suite for measuring parallel computer performance*. Dans International Workshop on OpenMP Applications and Tools, pages 1–10. Springer, 2001. *cité page 98*

[Augonnet 2011] Cédric Augonnet, Samuel Thibault, Raymond Namyst et Pierre-André Wacrenier. *StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures*. Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009, vol. 23, pages 187–198, Février 2011. *3 citations pages 42, 58, et 64*

[Bailey 1994] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan et S. Weeratunga. *The NAS Parallel Benchmarks*. Report RNR-94-007, Department of Mathematics and Computer Science, Emory University, March 1994. *cité page 98*

[Ben-Kiki 2009] Oren Ben-Kiki, Clark Evans et Ingy döt Net. *YAML Ain't Markup Language*, 2009. *cité page 71*

[Besseron 2009] Xavier Besseron, Christophe Laferriere, Daouda Traore et Thierry Gautier. *X-Kaapi : Une nouvelle implémentation eXtrême du vol de travail*. Dans Proceedings des Rencontres Francophones du Parallélisme, RenPar'19, Toulouse, France, France, 2009. *cité page 61*

[Bienia 2008] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh et Kai Li. *The PARSEC benchmark suite: Characterization and architectural implications*. Dans Proceedings of the 17th international conference on Parallel architectures and compilation techniques, pages 72–81. ACM, 2008. *cité page 98*

[Blackford 1999] Susan Blackford et Jack Dongarra. *LAPACK Working Note 41, Installation Guide for LAPACK*. http://www.netlib.org/lapack/lawnspdf/lawn41.pdf, 1999. [Online; accessed 10-January-2018]. *cité page 86*

[Bleuse 2014] Raphaël Bleuse, Thierry Gautier, João V. F. Lima, Grégory Mounié et Denis Trystram. Euro-par 2014 parallel processing: 20th international conference, porto, portugal, august 25-29, 2014. proceedings, chapitre Scheduling Data Flow Program in XKaapi: A New Affinity Based Algorithm for Heterogeneous Architectures, pages 560–571. Springer International Publishing, Cham, 2014. *2 citations pages 58 et 62*

[Blumofe 1996] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall et Yuli Zhou. *Cilk: An efficient multithreaded runtime system*. Journal of parallel and distributed computing, vol. 37, no. 1, pages 55–69, 1996. *cité page 43*

[Broquedis 2009] François Broquedis, Nathalie Furmento, Brice Goglin, Raymond Namyst et Pierre-André Wacrenier. *Dynamic task and data placement over NUMA architectures: an OpenMP runtime perspective*. Dans International Workshop on OpenMP, pages 79–92. Springer, 2009. *cité page 31*

[Broquedis 2010a] François Broquedis, Nathalie Furmento, Brice Goglin, Pierre-André Wacrenier et Raymond Namyst. *ForestGOMP: an efficient OpenMP environment for NUMA architectures*. International Journal on Parallel Programming, Special Issue on OpenMP; Guest Editors: Matthias S. Müller and Eduard Ayguade, vol. 38, no. 5, pages 418–439, 2010. *2 citations pages 59 et 105*

[Broquedis 2010b] François Broquedis, Jérôme Clet-Ortega, Stephanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault et Raymond Namyst. *hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications*. Dans Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, PDP 2010, Pisa, Italy, February 17-19, 2010, pages 180–186, 2010. *cité page 31*

[Broquedis 2012] François Broquedis, Thierry Gautier et Vincent Danjean. *LIBKOMP, an Efficient openMP Runtime System for Both Fork-join and Data Flow Paradigms*.

Dans Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World, IWOMP'12, pages 102–115, Berlin, Heidelberg, 2012. Springer-Verlag. *2 citations pages 62 et 114*

[BSC 2008] BSC. *SMP Superscalar (SMPSs) User's Manual*, 2008. *2 citations pages 42 et 98*

[Casanova 2001] Henri Casanova. *Simgrid: A toolkit for the simulation of application scheduling*. Dans Cluster computing and the grid, 2001. proceedings. first ieee/acm international symposium on, pages 430–437. IEEE, 2001. *cité page 126*

[Charles 2005] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun et Vivek Sarkar. *X10: an object-oriented approach to non-uniform cluster computing*. Dans Acm Sigplan Notices, volume 40, pages 519–538. ACM, 2005. *cité page 38*

[Che 2010] Shuai Che, J.W. Sheaffer, M. Boyer, L.G. Szafaryn, Liang Wang et K. Skadron. *A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads*. Dans Workload Characterization (IISWC), 2010 IEEE International Symposium on, pages 1–11, Dec 2010. *cité page 98*

[Clauss 2010] Pierre-Nicolas Clauss et Jens Gustedt. *Iterative Computations with Ordered Read-Write Locks*. Journal of Parallel and Distributed Computing, vol. 70, no. 5, pages 496–504, 2010. *cité page 57*

[Clet-Ortega 2014] Jérôme Clet-Ortega, Patrick Carribault et Marc Pérache. *Evaluation of OpenMP Task Scheduling Algorithms for Large NUMA Architectures*. Dans Euro-Par 2014 Parallel Processing - 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings, pages 596–607, 2014. *cité page 55*

[Cook 1971] Stephen A. Cook. *The Complexity of Theorem-proving Procedures*. Dans Proceedings of the Third Annual ACM Symposium on Theory of Computing, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM. *cité page 42*

[Diener 2014] Matthias Diener, Eduardo H.M. Cruz, Philippe O.A. Navaux, Anselm Busse et Hans-Ulrich Heiß. *kMAF: Automatic Kernel-level Management of Thread and Data Affinity*. Dans Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, PACT '14, pages 277–288, New York, NY, USA, 2014. ACM. *cité page 60*

[Dobson 2003] M. Dobson, P. Gaughen, M. Hohnbaum et E. Focht. *Linux Support for NUMA Hardware*. Dans Ottawa Linux Symposium 2003, 2003. *cité page 29*

[Drebes 2014] Andi Drebes, Karine Heydemann, Nathalie Drach, Antoniu Pop et Albert Cohen. *Topology-Aware and Dependence-Aware Scheduling and Memory Allocation for Task-Parallel Languages*. ACM Transactions on Architecture and Code Optimization, vol. 11, no. 3, pages 1–25, 2014. *cité page 59*

[Duran 2009] A. Duran, X. Teruel, R. Ferrer, X. Martorell et E. Ayguade. *Barcelona OpenMP Tasks Suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP*. Dans Parallel Processing, 2009. ICPP'09. International Conference on, pages 124–131. IEEE, 2009. *cité page 98*

[Duran 2011] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell et Judit Planas. *Ompss: a proposal for programming heterogeneous multi-core architectures*. Parallel Processing Letters, vol. 21, no. 02, pages 173–193, 2011. *2 citations pages 38 et 63*

[Durand 2013] Marie Durand, François Broquedis, Thierry Gautier et Bruno Raffin. Dans Proceedings of the 9th International Conference on OpenMP in the Era of Low Power Devices and Accelerators, pages 141–155, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. *3 citations pages 44, 57, et 62*

[Flynn 1966] M. J. Flynn. *Very high-speed computing systems*. Proceedings of the IEEE, vol. 54, no. 12, pages 1901–1909, Dec 1966. *cité page 25*

[Frigo 1998] Matteo Frigo, Charles E. Leiserson et Keith H. Randall. *The Implementation of the Cilk-5 Multithreaded Language*. Dans Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM. *2 citations pages 36 et 61*

[Gautier 2007] T. Gautier, X. Besseron et L. Pigeon. *Kaapi: A Thread Scheduling Runtime System for Data Flow Computations on Cluster of Multi-Processors*. Dans PASCO'07, 2007. *3 citations pages 44, 57, et 61*

[Gautier 2013] Thierry Gautier, Joao VF Lima, Nicolas Maillard et Bruno Raffin. *Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures*. Dans Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on, pages 1299–1308. IEEE, 2013. *2 citations pages 44 et 61*

[Graham 1966] Ronald L Graham. *Bounds for certain multiprocessing anomalies*. Bell Labs Technical Journal, vol. 45, no. 9, pages 1563–1581, 1966. *cité page 43*

[Gustedt 2017] Jens Gustedt, Emmanuel Jeannot et Farouk Mansouri. *Automatic, Abstracted and Portable Topology-Aware Thread Placement*. Dans IEEE Cluster, Cluster Computing (CLUSTER), 2017 IEEE International Conference on, pages 389 – 399, Hawaï, United States, Septembre 2017. *cité page 57*

[Hermann 2010] Everton Hermann, Bruno Raffin, François Faure, Thierry Gautier et Jérémie Allard. *Multi-GPU and multi-CPU parallelization for interactive physics simulations*. Dans European Conference on Parallel Processing, pages 235–246. Springer, 2010. *cité page 58*

[Hill 1989] Mark D Hill et Alan Jay Smith. *Evaluating associativity in CPU caches*. IEEE Transactions on Computers, vol. 38, no. 12, pages 1612–1630, 1989. *cité page 23*

[Intel 2014] Intel. *Intel® Xeon Phi™ Coprocessor System Software Developers Guide*, 2014. *cité page 24*

[Jaleel 2010] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr. et Joel Emer. *High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP)*. Dans Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA '10, pages 60–71, New York, NY, USA, 2010. ACM. *cité page 25*

[Jeffers 2016] James Jeffers, James Reinders et Avinash Sodani. Intel xeon phi processor high performance programming: Knights landing edition 2nd edition. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2016. *cité page 26*

[Kaiser 2014] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio et Dietmar Fey. *Hpx: A task based programming model in a global address space*. Dans Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, page 6. ACM, 2014. *cité page 38*

[Kale 1993] Laxmikant V. Kale et Sanjeev Krishnan. *CHARM++: A Portable Concurrent Object Oriented System Based on C++*. Dans Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications, OOPSLA '93, pages 91–108, New York, NY, USA, 1993. ACM. *cité page 56*

[Karp 1992] Richard M. Karp. *On-Line Algorithms Versus Off-Line Algorithms: How Much is It Worth to Know the Future?* Dans Proceedings of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture - Information Processing '92, Volume 1 - Volume I, pages 416–429, Amsterdam, The Netherlands, The Netherlands, 1992. North-Holland Publishing Co. *cité page 41*

[Keltcher 2003] C. N. Keltcher, K. J. McGrath, A. Ahmed et P. Conway. *The AMD Opteron processor for multiprocessor servers*. IEEE Micro, vol. 23, no. 2, pages 66–76, March 2003. *cité page 27*

[Kleen 2004] A. Kleen. *A NUMA API for Linux.*, 2004. *cité page 31*

[Koelbel 1994] Charles H Koelbel. The high performance fortran handbook. MIT press, 1994. *cité page 56*

[Kurzak 2008] Jakub Kurzak, Alfredo Buttari et Jack Dongarra. *Solving systems of linear equations on the CELL processor using Cholesky factorization*. IEEE Transactions on Parallel and Distributed Systems, vol. 19, no. 9, pages 1175–1186, 2008. *cité page 42*

[Kurzak 2009] Jakub Kurzak et Jack Dongarra. *QR Factorization for the CELL Processor*. Scientific Programming (to appear), 2009. *cité page 42*

[Kurzak 2010] Jakub Kurzak, Hatem Ltaief, Jack Dongarra et Rosa M. Badia. *Scheduling Dense Linear Algebra Operations on Multicore Processors*. Concurr. Comput. : Pract. Exper., vol. 22, no. 1, pages 15–44, Janvier 2010. *2 citations pages 42 et 98*

[Kurzak 2013] Jakub Kurzak, Piotr Luszczek, Asim YarKhan, Mathieu Faverge, Julien Langou, Henricus Bouwmeester et Jack Dongarra. Multithreading in the plasma library, pages 119–141. Chapman and Hall/CRC, 2013. *3 citations pages 64, 83, et 98*

[Kędzierski 2010] K. Kędzierski, M. Moreto, F. J. Cazorla et M. Valero. *Adapting cache partitioning algorithms to pseudo-LRU replacement policies*. Dans 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS), pages 1–12, April 2010. *cité page 24*

[Le Mentec 2011] Fabien Le Mentec, Thierry Gautier et Vincent Danjean. *The X-Kaapi's Application Programming Interface. Part I: Data Flow Programming*. Technical Report RT-0418, INRIA, Decembre 2011. *cité page 62*

[libGOMP 2018] libGOMP. *libGOMP's source code*, 2018. *cité page 62*

[Lima 2015] João V. F. Lima, Thierry Gautier, Vincent Danjean, Bruno Raffin et Nicolas Maillard. *Design and analysis of scheduling strategies for multi-CPU and multi-GPU architectures*. Parallel Computing, vol. 44, pages 37–52, 2015. *4 citations pages 42, 58, 61, et 62*

[Locke 1992] C Douglass Locke. *Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives*. Real-Time Systems, vol. 4, no. 1, pages 37–53, mar 1992. *cité page 42*

[McCalpin 1995] John D McCalpin. *A survey of memory bandwidth and machine balance in current high performance computers*. IEEE TCCA Newsletter, vol. 19, page 25, 1995. *cité page 59*

[McVoy 1996] Larry W McVoy, Carl Staelin*et al. lmbench: Portable Tools for Performance Analysis.* Dans USENIX annual technical conference, pages 279–294. San Diego, CA, USA, 1996. *cité page 23*

[Muller 1989] Jean-Michel Muller. Arithmétique des ordinateurs. Masson, 1989. *cité page 25*

[Novillo 2006] Diego Novillo. *OpenMP and automatic parallelization in GCC*. the Proceedings of the GCC Developers Summit, 2006. *cité page 62*

[Olivier 2012] Stephen Olivier, Allan Porterfield, Kyle B. Wheeler, Michael Spiegel et Jan F. Prins. *OpenMP task scheduling strategies for multicore NUMA systems*. IJHPCA, vol. 26, no. 2, pages 110–124, 2012. *2 citations pages 55 et 110*

[Olivier 2013] Stephen L. Olivier, Bronis R. De Supinski, Martin Schulz et Jan F. Prins. *Characterizing and mitigating work time inflation in task parallel programs*. Scientific Programming, vol. 21, no. 3-4, pages 123–136, 2013. *cité page 55*

[OpenMP Architecture Review Board 2013] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 4.0*, Juillet 2013. *cité page 47*

[OpenMP Architecture Review Board 2015] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 4.5*, Novembre 2015. *2 citations pages 36 et 45*

[PGAS 2013] PGAS. *PGAS*, Juillet 2013. *cité page 38*

[Pilla 2014] Laércio L. Pilla. *Topology-Aware Load Balancing for Performance Portability over Parallel High Performance Systems*. Theses, Université de Grenoble ; UFRGS, Avril 2014. *cité page 56*

[PLASMA 2013] PLASMA. *PLASMA version 2.6*, 2013. *cité page 99*

[Pop 2013] Antoniu Pop et Albert Cohen. *OpenStream: Expressiveness and Data-flow Compilation of OpenMP Streaming Programs*. ACM Trans. Archit. Code Optim., vol. 9, no. 4, pages 53:1–53:25, Janvier 2013. *cité page 63*

[Quinlan 2003] Dan Quinlan, Markus Schordan, Qing Yi et Bronis R de Supinski. *A C++ Infrastructure for Automatic Introduction and Translation of OpenMP Directives*. Dans Michael J Voss, editeur, OpenMP Shared Memory Parallel Programming, pages 13–25, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. *cité page 62*

[Reinders 2007] James Reinders. Intel threading building blocks. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007. *cité page 38*

[Reinman 2015] Mickael Reinman. *NUMA-aware scheduling for both memory- and compute-bound tasks*. 2015. *cité page 54*

[Ribeiro 2009] C. P. Ribeiro, J. F. Mehaut, A. Carissimi, M. Castro et L. G. Fernandes. *Memory Affinity for Hierarchical Shared Memory Multiprocessors*. Dans 2009 21st International Symposium on Computer Architecture and High Performance Computing, pages 59–66, Oct 2009. *2 citations pages 56 et 105*

[Rico 2010] Alejandro Rico, Felipe Cabarcas, Antonio Quesada, Milan Pavlovic, Augusto Javier Vega, Carlos Villavieja, Yoav Etsion et Alex Ramirez. *Scalable simulation of decoupled accelerator architectures*. Universitat Politecnica de Catalunya, Tech. Rep. UPC-DACRR-2010-14, 2010. *cité page 126*

[SGI 2012] SGI. *SGI UV 2000 System User Guide*, 2012. *cité page 77*

[Stanisic 2015] Luka Stanisic, Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, Arnaud Legrand, Florent Lopez et Brice Videau. *Fast and Accurate Simulation of Multithreaded Sparse Linear Algebra Solvers*. Dans The 21st IEEE International Conference on Parallel and Distributed Systems, Melbourne, Australia, Decembre 2015. *cité page 126*

[Stone 2010] John E. Stone, David Gohara et Guochun Shi. *OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems*. IEEE Des. Test, vol. 12, no. 3, pages 66–73, Mai 2010. *cité page 50*

[Tahan 2014] Oussama Tahan. *Towards Efficient OpenMP Strategies for Non-Uniform Architectures*. CoRR, vol. abs/1411.7131, 2014. *cité page 56*

[Tchiboukdjian 2010] Marc Tchiboukdjian, Nicolas Gast, Denis Trystram, Jean-Louis Roch et Julien Bernard. *A tighter analysis of work stealing*. Dans International Symposium on Algorithms and Computation, pages 291–302. Springer, 2010. *cité page 61*

[Terboven 2012] Christian Terboven, Dirk Schmidl, Tim Cramer et Dieter an Mey. *Task-Parallel Programming on NUMA Architectures*. Dans Euro-Par 2012 Parallel Processing - 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings, pages 638–649, 2012. *cité page 55*

[Terboven 2016] Christian Terboven, Jonas Hahnfeld, Xavier Teruel, Sergi Mateo, Alejandro Duran, Michael Klemm, Stephen L Olivier et Bronis R de Supinski. *Approaches for Task Affinity in OpenMP*. Dans International Workshop on OpenMP, pages 102–115. Springer, 2016. *cité page 58*

[TF1 2017] TF1. *Jules Desjardin gagne la seconde émission «Les cerveaux»*. https://fr.wikipedia.org/wiki/Les_Cerveaux, 2017. *cité page 5*

[Topcuoglu 2002] H. Topcuoglu, S. Hariri et Min-You Wu. *Performance-effective and low-complexity task scheduling for heterogeneous computing*. IEEE Transactions on Parallel and Distributed Systems, vol. 13, no. 3, pages 260–274, Mar 2002. *cité page 42*

[Videau 2017] Brice Videau, Kevin Pouget, Luigi Genovese, Thierry Deutsch, Dimitri Komatitsch, Frédéric Desprez et Jean-François Méhaut. *BOAST: A metaprogramming framework to produce portable and efficient computing kernels for HPC applications*. International Journal of High Performance Computing Applications, Août 2017. *cité page 70*

[Virouleau 2014] Philippe Virouleau, Pierrick Brunet, François Broquedis, Nathalie Furmento, Samuel Thibault, Olivier Aumage et Thierry Gautier. Evaluation of OpenMP dependent tasks with the KASTORS benchmark suite, volume 8766. 2014. *4 citations pages 14, 49, 98, et 103*

[Virouleau 2016a] Philippe Virouleau, François Broquedis, Thierry Gautier et Fabrice Rastello. Using data dependencies to improve task-based scheduling strategies on numa architectures, volume 9833. 2016. *3 citations pages 14, 105, et 115*

[Virouleau 2016b] Philippe Virouleau, Adrien Roussel, François Broquedis, Thierry Gautier, Fabrice Rastello et Jean-Marc Gratien. Description, Implementation and Evaluation of an Affinity Clause for Task Directives, pages 61–73. Springer International Publishing, Cham, 2016. *4 citations pages 14, 58, 106, et 108*

[Wittmann 2011] Markus Wittmann et Georg Hager. *Optimizing ccNUMA locality for task-parallel execution under OpenMP and TBB on multicore-based systems*. CoRR, vol. abs/1101.0093, 2011. *cité page 57*

[Wulf 1995] Wm. A. Wulf et Sally A. McKee. *Hitting the Memory Wall: Implications of the Obvious*. SIGARCH Comput. Archit. News, vol. 23, no. 1, pages 20–24, Mars 1995. *cité page 21*

[Xinmin 2014] Tian Xinmin et Bronis R. de Supinski. *Explicit Vector Programming with OpenMP 4.0 SIMD Extensions*. http://www.hpctoday.com/hpc-labs/explicit-vector-programming-with-openmp-4-0-simd-extensions/, 2014. [Online; accessed 10-January-2018]. *cité page 49*

[Yang 1994] Tao Yang et A. Gerasoulis. *DSC: scheduling parallel tasks on an unbounded number of processors*. IEEE Transactions on Parallel and Distributed Systems, vol. 5, no. 9, pages 951–967, Sep 1994. *cité page 42*

[YarKhan 2011] A. YarKhan, J. Kurzak et J. Dongarra. *QUARK Users' Guide: QUeueing And Runtime for Kernels*. Rapport technique, Innovative Computing Laboratory, University of Tennessee, 2011. *cité page 99*

[Yu 2017] Seongdae Yu, Seongbeom Park et Woongki Baek. *Design and Implementation of Bandwidth-aware Memory Placement and Migration Policies for Heterogeneous Memory Systems*. Dans Proceedings of the International Conference on Supercomputing, ICS '17, pages 18:1–18:10, New York, NY, USA, 2017. ACM. *cité page 60*

[Ziakas 2010] D. Ziakas, A. Baum, R. A. Maddox et R. J. Safranek. *Intel ® QuickPath Interconnect Architectural Features Supporting Scalable System Architectures*. Dans 2010 18th IEEE Symposium on High Performance Interconnects, pages 1–6, Aug 2010. *cité page 27*

A Scenarios YAML

A.1 Saturation du lien local

Listing A.1: Scénario de saturation du lien groupe entre le nœud 0 et le nœud 1 avec 8 copies simultanées

```yaml
---
scenarii:
  params: {}
  data:
    n_elems:
      type: int
      value: 25000000
    a<i>:
      i: [0, 7, 1]
      type: double*
      value: 0
    b<i>:
      i: [0, 7, 1]
      type: double*
      value: 0
  actions:
  # Initialisation des tableaux a<i> (source), sur le nœud 1
  - for:
    var: i
    limits: [0, 7, 1]
    actions:
    - kernel: init_array
```

```
      core: 8
      params:
      - a<i>
      - n_elems
  # Initialisation des tableaux b<i> (destination), sur le nœud 0
  - for:
    var: i
    limits: [0, 7, 1]
    actions:
    - kernel: init_array
      core: i
      params:
      - b<i>
      - n_elems
  - kernel: barrier
  # Exécution des copies simultanées sur les cœurs : 0, puis 0 et 1,
  # puis 0, 1, 2, etc.
  - for:
    var: i
    limits: [0, 7, 1]
    actions:
    - for:
      var: j
      # Génération des copies
      # simultanées.
      limits: [0, i, 1]
      actions:
      - kernel: copy
        core: 1
        repeat: 10
        params:
        - a<j>
        - b<j>
        - n_elems
    - kernel: barrier
  watchers:
    - name: time
      kernels: copy
```

A.2 Scénario de base d'un GEMM sur un cœur

Listing A.2: Scénario de calcul d'un gemm

```
1 ---
2 scenarii:
3   data:
4     size:
5       type: int
6       value: 256
7     <name>:
8       name: ["a", "b", "c"]
9       type: double*
10  actions:
11  - for:
```

```
    var: name
    values: ["a", "b", "c"]
    actions:
    - kernel: init_blas_bloc
      core: 0
      params:
      - <name>
      - size
  - kernel: dgemm
    core: 0
    repeat: 50
    params:
    - a
    - b
    - c
    - size
watchers:
- name: flops_dgemm
  params:
  - size
  kernels:
  - dgemm
- name: papi:
  params:
  - PAPI_L3_TCM
  - PAPI_L3_DCR
  - PAPI_L3_DCW
  kernels:
  - dgemm
```

A.3 Sénario d'exécution de 8 GEMM indépendants

Listing A.3: Scénario de calcul de 8 gemm indépendants

```
---
scenarii:
  data:
    size:
      type: int
      value: 256
    a<i>:
      i: [0, 7, 1]
      type: double*
    b<i>:
      i: [0, 7, 1]
      type: double*
    c<i>:
      i: [0, 7, 1]
      type: double*
  actions:
  - for:
    var: i
    limits: [0, 7, 1]
    actions:
    - for:
      var: name
      values: ["a", "b", "c"]
      actions:
      - kernel:
        init_blas_bloc
        core: <i>
        params:
        - <name><i>
        - size
  - kernel: barrier
  - for:
    var: i
    limits: [0, 7, 1]
    actions:
    - kernel: dgemm
      core: <i>
      repeat: '50'
      params:
      - a<i>
      - b<i>
      - c<i>
      - size
watchers:
- name: flops_dgemm
  params:
  - size
  kernels:
  - dgemm
- name: papi:
  params:
  - PAPI_L3_TCM
  - PAPI_L3_DCR
  - PAPI_L3_DCW
  kernels:
  - dgemm
```